National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Accelerating SpMM for Multi-head Self-Attention: Kernel-Level Design and Performance Analysis

Diploma Thesis

of

ANASTASIOS LAGOS

Supervisor: Georgios Goumas
Professor, NTUA

Athens, November 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Accelerating SpMM for Multi-head Self-Attention: Kernel-Level Design and Performance Analysis

Diploma Thesis

of

ANASTASIOS LAGOS

Supervisor: Georgios Goumas
Professor, NTUA

Approved by the examination committee on 3rd November 2025.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Georgios Goumas	Nectarios Koziris	Dionisios Pnevmatikatos
Professor, NTUA	Professor, NTUA	Professor, NTUA

Athens, November 2025

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

.....
Anastasios Lagos,
Graduate of School of
Electrical and Computer
Engineering, National
Technical University of
Athens

14th October 2025

Περίληψη

Τα Μεγάλα Γλωσσικά Μοντέλα βασίζονται στην αρχιτεκτονική Transformer για να συλλέξουν σχέσεις μεταξύ των λέξεων εισόδου. Καθώς το μέγεθος των μοντέλων και τα μήκη της ακολουθίας εισόδου αυξάνονται, η δυνατότητα αποτελεσματικής επεξεργασίας γίνεται ολοένα και πιο σημαντικός παράγοντας για την διατήρηση της επίδοσης και της επεκτασιμότητας.

Αραιές (Sparse) προσεγγίσεις του μηχανισμού attention των Transformers έχουν προταθεί για την επίλυση του προβλήματος πολυπλοκότητας. Παρά την θεωρητική τους αποδοτικότητα , συχνά δεν επιτυγχάνουν τους αναμενόμενους χρόνους ολοκλήρωσης σε σύγχρονες κάρτες γραφικών, λόγω των χαρακτηριστικών που εισάγει η αραιότητα των πινάκων που μετέχουν στον υπολογισμό του attention.

Η εργασία αυτή στοχεύει στην ανάλυση συνήθων σημείων συμφόρησης του πυρήνα πολλαπλασιασμού αραιού πίνακα με πίνακα, και επακόλουθη βελτιστοποίηση του, συγκρίνοντας την απόδοση του με την αντίστοιχη υλοποίηση της βιβλιοθήκης cuSPARSE της NVIDIA. Σε μεσαία αραιότητα, η αποδοτικότερη υλοποίηση προσφέρει έως και 57% επιτάχυνση, ενώ σε υψηλές αραιότητες επιβραδύνεται κατά 49%. Ένας εναλλακτικός σχεδιασμός του πυρήνα, επιταχύνεται σε υψηλές αραιότητες έως και 64% σε σχέση με την cuSPARSE, αλλά επιβραδύνεται κατά 250% σε χαμηλές. Συμπεραίνεται ότι, η αντιμετώπιση διαφορετικών συνόλων παραμέτρων ως διακριτά προβλήματα σχεδιασμού αποδεικνύεται πιο αποδοτική, προσφέροντας σημαντικά κέρδη επίδοσης έναντι μιας ενιαίας λύσης.[1]

Λέξεις-Κλειδιά

Νευρωνικά Δίκτυα, Βαθιά Μάθηση, Ενισχυτική Μάθηση, Μεγάλα Γλωσσικά Μοντέλα, Παράλληλα Υπολογιστικά Συστήματα, Πολλαπλασιασμός Αραιού Πίνακα με Πίνακα, Πυρήνες, Μετασχηματιστές, Αραιή Προσοχή

[1] Πηγαίος κώδικας: https://github.com/gosutek/sparse-attention

Abstract

Large Language Models (LLMs) rely on the Transformer architecture to capture dependencies on input tokens. As model sizes and input contexts continue to grow, the ability to efficiently handle extended sequences becomes increasingly critical to maintaining performance and scalability. However, the computational and memory demands of the standard attention mechanism increase by an order of $O(n^2)$ with respect to sequence length, impeding models from scaling further. A proposed solution, the sparse approximation of attention, aims to reduce the overall complexity while maintaining model quality. However, when implemented on accelerator platforms such as GPUs, sparse attention implementations often showcase a performance degradation caused by the emergent properties of sparsity.

This work aims to analyze common bottlenecks in the implementation details of the Sparse matrix-Matrix Multiplication (SpMM) kernel and its consequent optimization, comparing performance to NVIDIA's cuSPARSE library. The best kernel at low sparsity achieves a 57% speedup but for high sparsities, demonstrates a 49% slowdown. A different kernel design achieves a 64% speedup but for low sparsities, a 250% slowdown. Thus, the study concludes that tailoring kernel designs to the specific parameters of each problem achieves significantly better results than applying a catch-all approach.[2]

Keywords

Neural Networks, Deep Learning, Reinforcement Learning, Large Language Models, Transformers, Multi-Head Self-Attention, Sparse Attention, Sparse Matrix–Matrix Multiplication, Parallel Computing Systems, GPU Programming, Kernels, CUDA, Optimization

[2]Source code on: https://github.com/gosutek/sparse-attention

to my family

Acknowledgements

Immense gratitude to my supervising professor Georgios Goumas for the opportunity to work on such a project, uncovering a passion for low-level programming, parallel computing and optimization, I never realized I had, in the process. The successful completion of this work is, in part, due to the continuous endeavour of Ioanna Tasou to provide valuable support whenever that was required and also plenty of breathing room to pursue my academical interests.

Last but not least, I want to thank my parents for the material and immaterial support they have unwaveringly provided throughout this adventure and of course my brothers, for their tenacious belief in me from start to finish.

Athens, November 2025

Anastasios Lagos

Table of Contents

List of Figures

List of Images

List of Tables

1 Εκτενής περίληψη στα Ελληνικά

Τα Μεγάλα Γλωσσικά Μοντέλα βασίζονται στην αρχιτεκτονική Transformer [1], της οποίας, ο κύριος μηχανισμός, Multi-Head Self-Attention (MHSA) επιτρέπει τη μοντελοποίηση σχέσεων μεταξύ των λέξεων εισόδου. Καθώς τα μεγέθη μοντέλου και τα μήκη ακολουθίας εισόδου αυξάνονται, η δυνατότητα αποτελεσματικής επεξεργασίας της εισόδου γίνεται κρίσιμος παράγοντας για τη διατήρηση τόσο της κλιμακωσιμότητας όσο και της απόδοσης. Ωστόσο, τυπικές υλοποιήσεις του μηχανισμού attention εμφανίζουν τετραγωνική υπολογιστική πολυπλοκότητα και πολυπλοκότητα μνήμης citedao2022flashattention σε σχέση με το μήκος της ακολουθίας εισόδου. [2, 3]

Αραιές προσεγγίσεις του attention έχουν προταθεί ως λύση για την επίλυση του προβλήματος πολυπλοκότητας. [2, 4] Αυτές οι μέθοδοι [5, 6] μειώνουν το συνολικό υπολογιστικό κόστος, διατηρώντας παράλληλα συγκρίσιμη ακρίβεια μοντέλου. [7] Παρά την θεωρητική τους αποδοτικότητα, οι μηχανισμοί sparse attention συχνά δεν επιτυγχάνουν τους αναμενόμενους χρόνους ολοκλήρωσης [8] σε σύγχρονες κάρτες γραφικών (GPU). Μερικές από τις αιτίες είναι η ανισόρροπη κατανομή φόρτου εργασίας στα νήματα επεξεργασίας αλλά και η ακανόνιστη πρόσβαση στην μνήμη, χαρακτηριστικά που δημιουργούνται λόγω της αραιότητας.

Συνεπώς, η αποτελεσματική υλοποίηση του πυρήνα Sparse Matrix–Matrix Multiplication (SpMM) ,μια κεντρική λειτουργία στο Sparse Attention , παραμένει κρίσιμο αντικείμενο έρευνας. Αυτή η εργασία αφοσιώνεται στην λεπτομερή ανάλυση της απόδοσης της πράξης SpMM, όταν αυτή υλοποιείται σε κάρτες γραφικών. Πραγματοποιεί ανάπτυξη πυρήνων σε CUDA, και τους συγκρίνει με την αντίστοιχη υλοποίηση της βιβλιοθήκης cuSPARSE της NVIDIA. Κάθε πυρήνας βελτιώνει σταδιακά πτυχές της εκτέλεσης, όπως η συνένωση μνήμης, η χρήση (ή μη) shared memory, warp-level προγραμματισμός, και η κατανομή εργασίας σε μη ομοιόμορφα πρότυπα αραιότητας.

Οι κάρτες γραφικών (GPU) είναι η πλέον δημοφιλής επιλογή όσο αναφορά τον υπολογισμό του attention λόγω της ικανότητάς της να εκτελεί χιλιάδες νήματα παράλληλα. Αντίθετα, ο κεντρικός επεξεργαστής (CPU) έχει βελτιστοποιηθεί για μικρό αριθμό νημάτων και σειριακή εκτέλεση. Η GPU οργανώνεται σε Streaming Multiprocessors (SMs), τα οποία περιλαμβάνουν πολλαπλούς πυρήνες και μονάδες διαχείρισης της μνήμης. Τα νήματα ομαδοποιούνται σε blocks, τα οποία σχηματίζουν ένα grid εκτέλεσης. Κάθε SM εκτελεί ομάδες των 32 νημάτων, τα λεγόμενα warps, τα οποία εκτελούν την ίδια εντολή ταυτόχρονα (μοντέλο Single Instruction Multiple Threads). Όταν τα νήματα ενός warp αποκλίνουν εκτελώντας διαφορετικές εντολές, εμφανίζεται warp divergence, που μειώνει την απόδοση. Η εκτέλεση πραγματοποιείται μέσω πυρήνων (kernels) που ορίζουν πλέγματα νημάτων και blocks. [9]

Η GPU διαθέτει πολυεπίπεδη ιεραρχία μνήμης για ισορροπία μεταξύ ταχύτητας και εύρους ζώνης:

- **Global Memory:** υψηλού εύρους ζώνης αλλά υψηλής καθυστέρησης.

- **L2 Cache:** κοινόχρηστη για όλα τα SMs.
- **L1 Cache / Shared Memory:** ταχεία μνήμη ανά block.
- **Registers:** ιδιωτική μνήμη κάθε νήματος.

Το μοντέλο attention μετασχηματίζει τα διανύσματα εισόδου σε queries (Q), keys (K) και values (V) μέσω γραμμικών προβολών.

$$\mathbf{Q_i} = XW_i^Q \in \mathbb{R}^{N\times d} \tag{1.1}$$
$$\mathbf{K_i} = XW_i^K \in \mathbb{R}^{N\times d} \tag{1.2}$$
$$\mathbf{V_i} = XW_i^V \in \mathbb{R}^{N\times d} \tag{1.3}$$

Η βασική πράξη attention είναι: [1, 8]

$$\mathbf{S} = \mathbf{QK}^\top \in \mathbb{R}^{N\times N}, \qquad \mathbf{A} = \mathrm{softmax}(\mathbf{S}) \in \mathbb{R}^{N\times N}, \qquad \mathbf{O} = \mathbf{AV} \in \mathbb{R}^{N\times d_h} \tag{1.4}$$

Οι έξοδοι από κάθε ¨κεφαλή' (head) συνενώνονται και προβάλλονται εκ νέου σε γραμμική μορφή. Η συνάρτηση softmax κανονικοποιεί τις τιμές των scores ώστε να παράγει πιθανοτικές στάθμες βαρύτητας.

Η πράξη SpMM (Sparse matrix–Matrix Multiplication) αποτελεί το βασικό υπολογιστικό φορτίο του Multi-head Self-Attention (MHSA):

$$C = \alpha A \cdot B + \beta C$$

όπου το A είναι πυκνός πίνακας και ο B αραιός. Οι αραιές πράξεις μειώνουν τη χρησιμοποίηση υπολογιστικών πόρων εξαιτίας μη κανονικών προσπελάσεων στη μνήμη, καθιστώντας τον SpMM μια λειτουργία περιορισμένη από τη μνήμη.

Παρουσιάζονται διάφορες μορφές αποθήκευσης:

- COO (Coordinate List): αποθήκευση συντεταγμένων και τιμών κάθε μη μηδενικού στοιχείου.
- CSR (Compressed Sparse Row): συμπιεσμένη μορφή κατά γραμμές.
- CSC (Compressed Sparse Column): συμπιεσμένη μορφή κατά στήλες.

Προηγμένες μορφές όπως BCSR [10], ELLPACK, DIA, CSF στοχεύουν στη βελτίωση της επαναχρησιμοποίησης δεδομένων σε υπολογιστικά μπλοκ.

Υπολογίζεται το θεωρητικό άνω όριο επιδόσεων με βάση την υπολογιστική ισχύ FP32 (44.10 TFLOPs/s) και το εύρος μνήμης (672.3 GB/s) της GPU. Τα αποτελέσματα δείχνουν ότι ο πολλαπλασιασμός αραιών πινάκων περιορίζεται κυρίως από τις προσπελάσεις μνήμης, με τον χρόνο μεταφοράς δεδομένων να υπερβαίνει τον χρόνο υπολογισμού περίπου κατά 1.3×, φτάνοντας έως και 27× σε υψηλά επίπεδα αραιότητας (98%).

Θα παρουσιάσω έξι διαφορετικούς πυρήνες για τον υπολογισμό του Πολλαπλασιασμού Αραιού Πίνακα με Πίνακα (SpMM), αναλύοντας λεπτομερώς τον σχεδιασμό, τα χαρακτηριστικά και την απόδοσή τους. Ο πρώτος πυρήνας χρησιμοποιεί shared memory για να φέρει τα δεδομένα πιο κοντά στους πυρήνες υπολογισμού, μειώνοντας τις μεταφορές δεδομένων, αλλά πάσχει από μη συνενωμένες (uncoalesced) προσπελάσεις μνήμης και περιττή κατανάλωση bandwidth. Ο δεύτερος πυρήνας αναδιανέμει το έργο ανά μη μηδενικό στοιχείο, επιτυγχάνοντας συνενωμένη πρόσβαση στη μνήμη και καλύτερη εκμετάλλευση των cache αν και εισάγει νέες προκλήσεις όπως η μη ομοιόμορφη κατανομή μη μηδενικών ανά στήλη και η ανάγκη συγχρονισμού μεταξύ νημάτων. Ο τρίτος πυρήνας ελαχιστοποιεί τη χρήση shared memory για να αποκαταστήσει τη λειτουργικότητα της L1 cache, μειώνοντας τη σπατάλη εύρους ζώνης και βελτιώνοντας την επίδοση σε υψηλές αραιότητες. Ο τέταρτος εισάγει διανυσματικές (vectorized) προσπελάσεις μνήμης και προσαρμοστική κατανομή (adaptive tiling), αλλά το μικρό μέγεθος των μητρών περιορίζει τα οφέλη. Ο πέμπτος αξιοποιεί κατανομή στηλών ώστε κάθε block να επεξεργάζεται πολλαπλές στήλες, βελτιώνοντας την επαναχρησιμοποίηση δεδομένων και την απόδοση σε υψηλές αραιότητες. Τέλος, ο έκτος πυρήνας εφαρμόζει κατανομή τύπου block tiling σε στοιχειακό επίπεδο, εστιάζοντας ταυτόχρονα στη διατήρηση της λειτουργικότητας των cache και στη μείωση σπατάλης μνήμης.

Μέσα από συστηματικά πειράματα, η ανάλυση αποκαλύπτει πώς διαφορετικά σχεδιαστικά trade-offs επηρεάζουν τη συνολική ροή και τις επιτευχθείσες floating-point πράξεις ανά δευτερόλεπτο (FLOPs). Τα εμπειρικά αποτελέσματα δείχνουν ότι, διαφορετική σχεδιαστική προσέγγιση ανάλογα με τις παραμέτρους και το επίπεδο της αραιότητας, οδηγεί σε σημαντικά καλύτερα αποτελέσματα σε σύγκριση με μια ενιαία, γενική λύση. Συγκεκριμένα, μέγιστη επιτάχυνση 57% επιτεύχθηκε σε 50% αραιότητα, ενώ ένας εναλλακτικός σχεδιασμός πυρήνα πέτυχε έως και 64% επιτάχυνση σε 98% αραιότητα σε σχέση με την cuSPARSE. Οι μετρήσεις αυτές, υπογραμμίζουν την σημασία προσαρμογής των πυρήνων στην κατανομή αραιότητας και την πιθανότητα σημαντικής βελτίωσης στην απόδοση του μηχανισμού attention.

2 Introduction

The Transformer architecture [1] has become the foundation of modern deep learning models in natural language processing [11] and image generation [12]. At the core of the Transformer lies the Multi-Head Self-Attention (MHSA) mechanism, which enables the model to capture long-range dependencies and contextual relationships between input tokens. As model sizes and sequence lengths continue to increase, the ability to efficiently process extended input contexts becomes a crucial factor in sustaining both scalability and performance. However, the complexity of the standard MHSA scales quadratically [8] with input sequence length, both in terms of computation and memory, which severely limits its applicability to long sequences [2, 4].

To address this issue, numerous approaches have been proposed to introduce sparsity into the attention mechanism [2, 3], thereby reducing the number of pairwise interactions computed during self-attention. Sparse attention models [5, 6], exploit structured sparsity patterns to approximate full attention while maintaining comparable accuracy [7]. These sparse variants significantly reduce computational requirements, making large-scale Transformer models more efficient and scalable. Despite their theoretical efficiency, sparse attention mechanisms often fail to achieve expected speedups on modern GPU accelerators due to the irregular memory access patterns and workload imbalance introduced by sparsity. Consequently, efficient Sparse Matrix–Matrix Multiplication (SpMM) implementations, an operation central to sparse attention, remain a critical research focus.

In this work, I perform an in-depth analysis of the performance bottlenecks inherent in GPU-based SpMM kernels using general sparse formats. Multiple CUDA kernel variants are developed and benchmarked against NVIDIA's cuSPARSE library to evaluate performance across a range of sparsity levels. Each kernel incrementally refines aspects of GPU execution, including memory coalescing, shared memory utilization, warp-level scheduling, and work distribution across non-uniform sparsity patterns. Through experimentation, the analysis reveals how different design trade-offs impact overall throughput and achieved floating-point operations per second (FLOPs).

Empirical results demonstrate that the proposed kernels outperform NVIDIA's cuSPARSE implementation at both low and high sparsity regimes. Specifically, a maximum speedup of 57% was achieved at low sparsities but a 49% slowdown at higher sparsities, while an alternative kernel design achieved up to 64% speedup at extreme sparsities and a 250% slowdown at dense-like sparsities. These findings underscore the importance of adapting kernel execution strategies to the sparsity distribution of input matrices and highlight the potential for significant efficiency gains in sparse attention computation on GPU platforms.

3 Background

I provide some background on fundamental hardware details of the Graphics Processing Unit (GPU) and an overview of Standard Multi-head Self-Attention. I also describe the Sparse matrix-Matrix multiplication, the prevalent operation in attention computation and some general sparse formats (GSFs).

3.1 Graphics Processing Unit

The Graphics Processing Unit (GPU) emerged as a dominant accelerator platform for computing attention scores, owning to its ability to launch and execute thousand of threads (sequences of operation) in parallel. Designed with different goals in mind, the Central Processing Unit (CPU) is optimized for low-latency, single-thread performance and complex control logic (e.g., branch predictors and speculative execution) but can only launch a few tens of concurrent threads. Whereas the GPU sacrifices complex control mechanisms, it comprensates with substantially higher throughput, effectively hiding single-thread latency through massive parallel execution [9]. The GPU architecture is organized around a scalable array of multithreaded Streaming Multiprocessors (SMs). Each SM contains multiple cores capable of executing arithmetic and logic instructions, along with special units for memory and scheduling management. At a higher level of abstraction, threads are organized into *blocks*, which are distributed across the available SMs for execution, while multiple blocks collective compose a *grid*, corresponding to a single execution context.

3.1.1 Single Instruction Multiple Threads (SIMT)

When an SM is given a block of threads to process, it forms groups of 32 (typically) threads called *warps* which serve as the fundamental unit for thread scheduling and execution. Warps are then enumerated in a static way, given increasing *thread IDs* and dispatched to the scheduler for execution. The Single Instruction Mutliple Threads (SIMT) model of execution, dictates that all threads of the same warp will execute the same instruction[1]. From this paradigm arises an important performance consideration: when threads of the same warp attempt to execute different instructions, *warp divergence* occurs and are executed in sequence. Warp divergence can significantly reduce performance and is a key factor to understand in order to achieve full utilization of the GPU.

To enable this massive parallelism, one or more special functions, called *kernels*, can be launched by the CPU, defining a grid of blocks of threads. The blocks are then distributed to SMs and follow the partition, scheduling and execution process discussed previously (Figure 3.1).

[1]Modern GPUs support *Independent Thread Scheduling*, allowing threads of different warps to execute that same instruction.

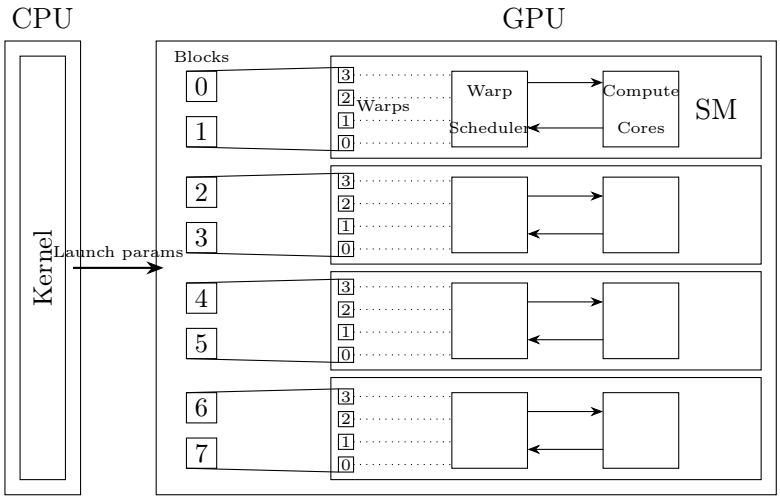

Figure 3.1. *Simple overview of how the GPU works*

3.1.2 Memory hierarchy

The memory hierarchy in a GPU is heterogeneous, designed to balance speed, scope and bandwidth across different levels of storage. Understanding this hierarchy and designing computations to work *with* it is crucial for achieving high performance, as memory access often represents the primary bottleneck in many GPU workloads due to its relatively high latency compared to compute.

Primary memory types used in this work:

- **Global Memory:** Device-level high-bandwidth memory (HBM or GDDR), accessible by all threads. High latency compared to on-chip memory.
- **L2 Cache:** On-chip cache shared across all SMs, serving as an intermediate buffer for global memory accesses. Provides moderate latency.
- **L1 Cache / Shared Memory:** On-chip memory with block-level scope, shared among threads within a block. Shared memory is allocated on the L1 cache, serving as a user-managed cache.
- **Registers:** Private to each thread, providing the fastest storage.

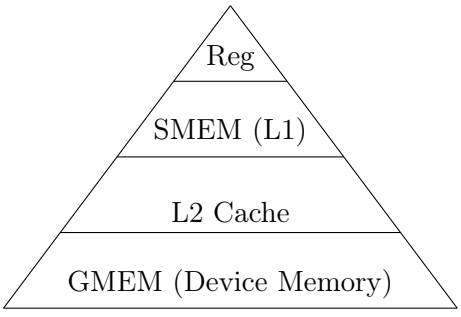

Figure 3.2. ***Memory hierarchy.*** *Base: Larger but slower. Top: Smaller but faster.*

3.2 Standard Multi-head Self-Attention

[1, 8] Given embeddings matrix $X \in \mathbb{R}^{N \times d}$, where N the sequence length, d the head dimension, and h learned linear projections $W_i^Q, W_i^K, W_i^V, W^O \in \mathbb{R}^{d \times d}$, where $0 \leq i \leq h$ and h the number of heads, the transformer network linearly projects the embeddings vectors of the input sequence to queries, keys and values:

$$\mathbf{Q_i} = XW_i^Q \in \mathbb{R}^{N \times d} \tag{3.1}$$

$$\mathbf{K_i} = XW_i^K \in \mathbb{R}^{N \times d} \tag{3.2}$$

$$\mathbf{V_i} = XW_i^V \in \mathbb{R}^{N \times d} \tag{3.3}$$

Each set of $\{Q_i, K_i, V_i\}$ (h in total), is used in the attention operation, with the execution context constituting a *head*. The attention operation computes the following:

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{N \times N}, \qquad \mathbf{A} = \text{softmax}(\mathbf{S}) \in \mathbb{R}^{N \times N}, \qquad \mathbf{O} = \mathbf{A}\mathbf{V} \in \mathbb{R}^{N \times d_h} \tag{3.4}$$

where $d = d_m/h$, d_m is the model dimension and softmax is applied row-wise. Note that, typically, a scaling factor of $\frac{1}{\sqrt{d}}$ and optional mask is applied to $\mathbf{S}$. Finally, each head's attention output $\mathbf{O}_i$ is concatenated column-wise resulting in a $N \times d_m$ matrix and then linearly projected back into usable output:

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Softmax [13] is defined as the function:

$$\sigma \ : \ \mathbb{R}^K \to (0,1)^K$$

where $K > 1$, takes a tuple $z = (z_1, \ldots, z_K) \in \mathbb{R}^K$ and computes each component of vector $\sigma(z) \in (0,1)^K$ with

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

3.3 Sparse Attention

3.3.1 Sparse matrix-Matrix Multiplication

[9] Sparse matrix-Matrix Multiplication computes the following [2]:

$$C = \alpha A \cdot B + \beta C$$

[2]Standard implementations of the SpMM kernel compute $C = \alpha AB + \beta C$ where A is sparse and B is dense. Kernel equivalence proof in Appendix A

- A is a dense matrix of size $m \times k$
- B is a sparse matrix of size $k \times n$
- C is a dense matrix of size $m \times n$
- α, β are scalar

For the remainder of this work, $\alpha = 1$ and $\beta = 0$.

As shown in Equations 3.1, 3.2, 3.3 and 3.4, SpMM represents the key computational workload in MHSA. Sparsity induces irregular memory accesses, which hinder throughput and make SpMM the main performance-limiting factor on GPU implementations.

3.3.2 General Sparse Formats

Given the importance of memory for GPU performance and the inefficient access pattern in sparse attention, *how* matrices are stored becomes critical:

- *Coordinate List (COO).* Keeps a coordinate and value structure *(row, column, value)* for each non-zero, packed in list. The leading dimension is controlled by the ordering of these vectors. Sorting them by row first makes the rows as the leading dimension and vice versa.
- *Compressed Sparse Row (CSR).* Similar to a row-major layout in dense matrices where rows are the leading dimension, the compressed sparse row (CSR) format focuses on storing positional information of the non-zeros. It achieves this through the use of three arrays, *row_ptr, col_idx* and *val*. The last two, store the column indices and values of the non-zeros, respectively, traversed in a row-major fashion. The *row_ptr* array stores a prefix sum of the number of non-zeros of each row. Hence, we can retrieve the number of non-zeros of row r by computing $row\_ptr[r+1] - row\_ptr[r]$ and their respective indices in *col_idx* and *val* by iterating the range $[row\_ptr[r], row\_ptr[r+1])$.
- *Compressed Sparse Column (CSC).* Equivalent to CSR but with the leading dimension being the column instead. The metadata arrays are typically called *col_ptr, row_idx* and *val*.

Quite a few GSFs (BCSR [10], ELLPACK, DIA, CSF) and custom sparse formats [14, 8, 15] have been proposed, it is noted that most of them exploit the data reusability that emerges from matrix multiplication in a block of elements.

3.3.3 Bounded by Memory

Here I provide a theoretical upper bound for compute and memory time, as bounded by the theoretical FP32 compute and memory bandwidth of the GPU used for benchmarking this work[3]. For each row of matrix A we perform one multiplication and one addition with

[3]Full specifications in Chapter 5. Experimental Evaluation.

each non-zero element of sparse matrix B. Modern hardware fuse these two operations into one, called Fused Multiply Add (FMA). However, theoretical FLOP bound calculations have both operations contribute. Therefore, total FLOPs for a Sparse matrix-Matrix Multiplication are calculated as:

$$m \times nnz \times 2$$

The matrix of the input data used in this work are of shape (512×512). At 50% sparsity, on average, we have a total of $130K$ non-zeros per matrix. Typical m dimensions range, as powers of two, from 512 up to $16K$. GPT-2 uses $1K$ for input sequence length:

$$1024 \times 131,000 \times 2 \approx 268.288 \text{ MFLOPs}$$

In regards to memory transactions, we read the input and store the output. Dense FP32 matrix A is represented with:

$$1024 \times 512 \times 4 \approx 2.09 \text{ MB}$$

Sparse matrix B in CSC format takes up:

$$(2 \times nnz + n + 1) \times 4 \approx 1.05 \text{ MB}$$

In total ≈ 3.14 MB. The resulting matrix is stored back in global memory, represented as a total of:

$$1024 \times 512 \times 4 \approx 2.09 \text{ MB}$$

Therefore, the total read and writes total up to 5.24 MB. The GPU's compute and memory bandwidth are as follows: 44.10 TFLOPs/s of FP32 and 672.3 GB/s of memory bandwidth.

The upper bound of compute:

$$\frac{268.288 \times 10^6}{44.10 \times 10^{12}} = 6.08\mu\text{s}$$

The upper bound of memory:

$$\frac{5.24 \times 10^6}{672.3 \times 10^9} = 7.79\mu\text{s}$$

That's $\sim 1.3\times$ more memory access time than compute time. This ratio gets even more disproportionate at higher sparsities. Performing the same calculation at 98% sparsity or $\approx 5K$ non-zeros results in $\sim 27.3\times$ more memory access. Hence, Sparse matrix-Matrix Multiplication is a memory bound operation.

4 Sparse Matrix Multiplication Kernels

Each section in this chapter presents a distinct Sparse Matrix–Matrix Multiplication (SpMM) kernel, providing a detailed analysis of its design, characteristics, and differences compared to its predecessor, along with an overview of its performance.

4.1 Kernel 1: Element-wise work distribution with shared memory

The main goal of this kernel is to try and bring reusable data closer to the computing cores by using shared memory, reducing the overall number of memory transactions in the process. Shared memory's scope, encompasses all threads in block, i.e. each block has its personal shared memory space shared amongst all threads in it. Adhering to the main goal, work is distributed to thread blocks such that data reusability is maximized. In Spmm, and matrix multiplication in general, multiple computations in the same data occur when calculating elements of the same row or column of C. This kernel chooses to distribute the calculation of $C(x, ::)$, consequently loading row x of A into SMEM.

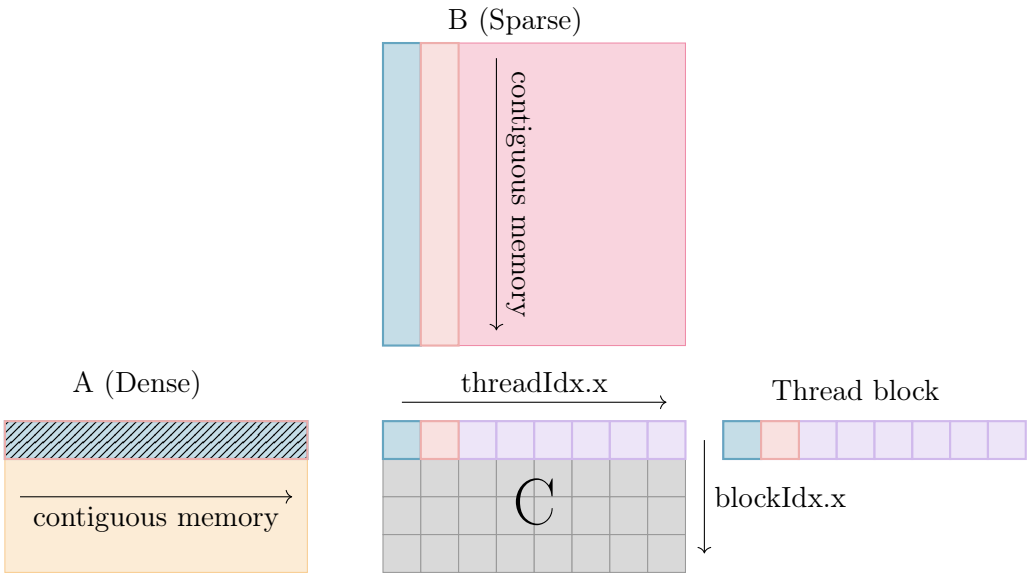

Figure 4.1. *A thread block (purple) over a row of C. Each thread calculates an element of C. The corresponding row of A is reused and consequently loaded into shared memory.*

We distribute a thread block across a row of the result matrix C. Consequently, each thread is assigned with the computation of an element from that row of the matrix in a $1:1$ scheme. Initially, each thread loads its corresponding element from A into shared memory, until the whole row is loaded (Figure 4.1). A is in row-major format hence the rows are the leading dimension, therefore, accessing elements of the same row consecutively means

accessing contiguous memory. On the contrary, accessing elements of the same column consecutively means accessing discontiguous memory.

As evident from Figure 4.2, performance is many times slower than cuSPARSE's implementation for low sparsities, but for extreme sparsities $> 95\%$ it performs $\sim 11\%$ faster.

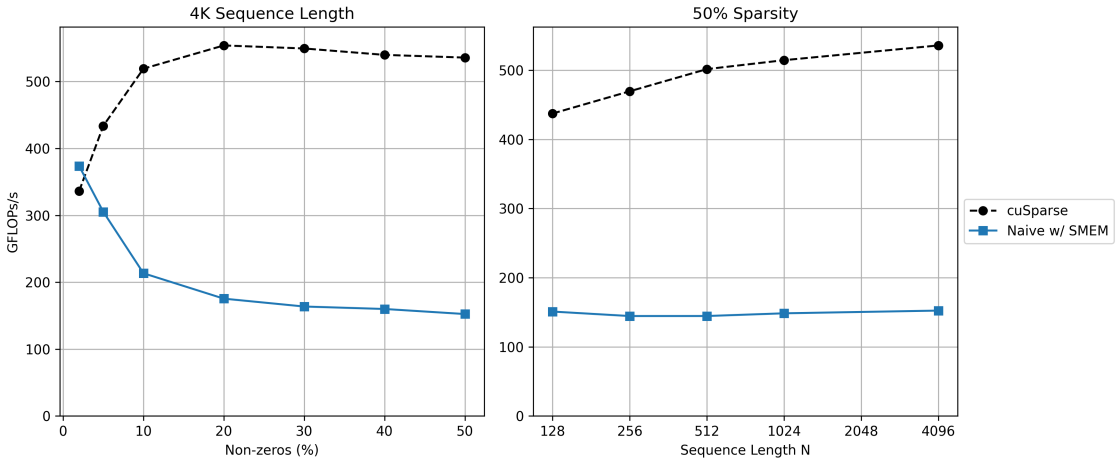

Figure 4.2. ***Element-wise w/ Shared memory*** *against cuSPARSE.* ***Left*** *Variable sparsity at 4K input sequence length.* ***Right*** *Variable sequence length @50% sparsity.* ***X-Axis:*** *Extreme sparsities for small x, dense matrices for high x.*

At both sparsities, this kernel suffers from the same memory access problem: *The hardware can not coalesce memory transactions into as few as possible.* This hypothesis is supported by NVIDIA's proprietary profiler, Nsight Compute which reports 87% excessive sectors and that warps are stalled for 244 cycles and 243 cycles waiting for the Global Memory and L1/TEX cache memory instruction queue to empty, respectively. The term *excessive sectors* refers to the proportion of global memory transactions that go beyond what is strictly necessary for computation. In other words, it quantifies how much redundant memory is being transferred, indicating wasted bandwidth.

Another critical issue lies in how this kernel addresses the uncertainty of which elements of row x are required by each sparse column. By disregarding sparsity, it transfers the entire row from GMEM to SMEM, resulting in unnecessary bandwidth consumption. This inefficiency will be examined in greater detail in Chapter 4.3. Note that this additional bandwidth waste, accumulutes with that caused by uncoalesced memory accesses, and the two sources of inefficiency are independent.

4.1.1 Coalesced memory access

As discussed in Chapter 3.1 NVIDIA GPUs are build upon Streaming Multiprocessors (SM) which utilize the SIMT architecture (Single Instruction, Multiple Threads). This enables instruction-level parallelism in a single thread and additionally, thread-level parallelism [9]. When we read (or write) to GMEM in this kernel, the compiler will generate an *LDG.E* (or *STG.E* for stores) instruction. The SIMT architecture allows all active threads (i.e. threads ready to execute the instruction), regardless of warp, to execute it. However, the hardware will only access device memory (which is where GMEM resides) via 32-, 64-,

or 128-byte contiguous memory transactions.

We can take advantage of this hardware feature such that we use less memory transactions overall, by making threads of the same warp load contiguous global memory. As an example, consider the following: We require of our threads to read 32 single precision (4 bytes) floating point numbers from memory. *Best case:* Each one of 32 floats occupies neighbouring memory spaces, hence, the hardware can now issue a single global memory read of 128-bytes satisfying the *LDG.E* instruction of every thread in that warp. *Worst Case:* Each one of the 32 floats is separated by 31 unrelated floats. The hardware, in an effort to service all the threads, will perform 32 memory transactions, each loading 124-bytes of data that goes unsued leading to excessive sectors.

4.1.2 Analyzing Memory Accesses in our Kernel

In this kernel we perform GMEM accesses when:

1. Read the input dense matrix X.

2. Read the compressed sparse matrix components *col_ptr, row_idx, val*.

3. Write back the result.

Dense matrix X is in row-major format, i.e. the leading dimension is across the rows. Each thread reads one element of X and writes it to shared memory and since it is indexed with *threadIdx.x*, we are indeed accessing contiguous memory per warp.

Let us analyze how we access the sparse matrix. Each thread is assigned with the computation of an element of C. Consequently, each thread will have to load the non-zeros of its corresponding column, and since the sparse matrix is in CSC then each thread will access contiguous memory. However, this does not take advantage of memory coalescing, *as each thread of a* **warp** *must access contiguous memory*, and not a *single* thread (Figure 4.3).

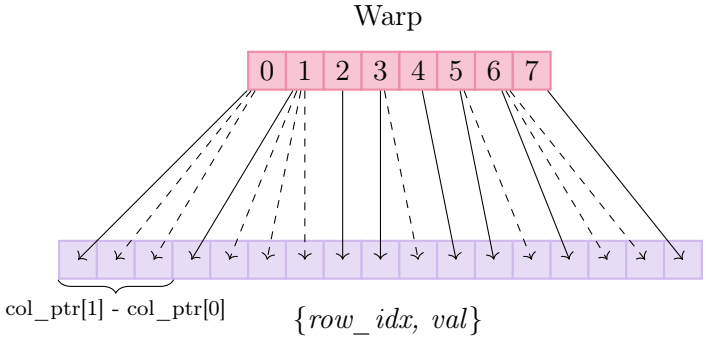

Figure 4.3. ***Uncoalesced Memory Access:*** *A single LDG.E instruction will be executed by all non-dashed threads, reading in a strided pattern. Assume eight threads per warp.*

Ideally, we want the access pattern demonstrated in Figure 4.4. This pattern ensures that a single global memory load services all the threads in a warp.

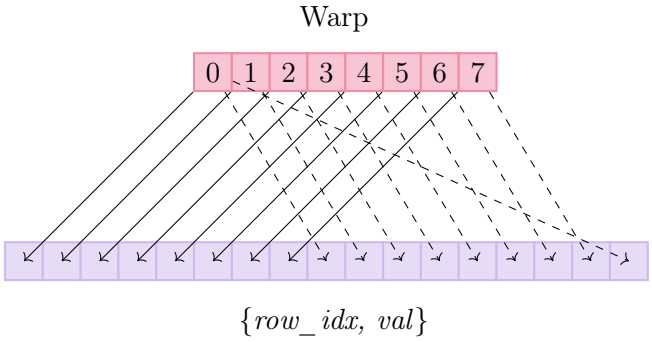

Figure 4.4. ***Coalesced Memory Access:*** *Here a single LDG.E instruction will service all the warps in the thread.*

4.2 Kernel 2: Nonzero-wise work distribution with coalesced memory access

To achieve full memory coalescing we must change our work distribution pattern. Instead of assigning each thread to the computation of an element of C, we assign a thread block to the computation of an element of C as shown in Figure 4.5.

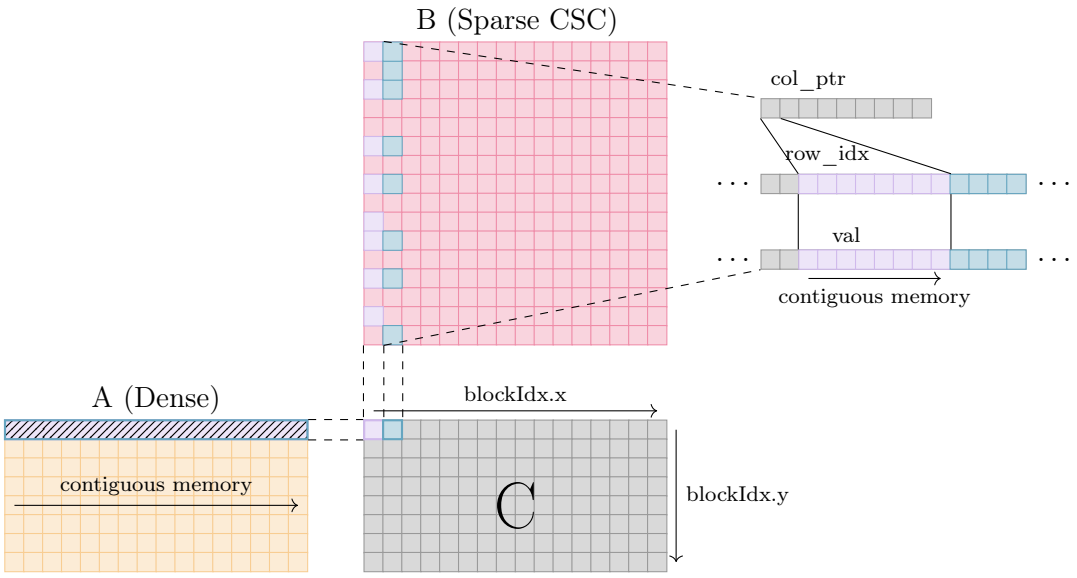

Figure 4.5. *Each block (colored) is tasked with computing a single element of the resulting matrix C. This allows its threads to access contiguous memory in the meta arrays of CSC.*

Now, the non-zero values of a column are delegated to the threads of a block. This allows for coalesced memory access, since they reside in contiguous memory in the form of the three meta arrays of the CSC sparse format. Such an approach revealed some new challenges, in turn, providing a deeper look into the nature of sparse matrices:

4.2.1 Non-constant number of non-zeros per column

The previous kernel (Section 4.1), distributed a block to a row of C, with each thread calculating one element. The number of columns stays constant for a matrix, so we,

trivially, set the number of threads equal to the number of columns.

With this current approach however, we assign non-zeros non-uniformly. Some blocks end up with more non-zeros, while some end up with less. As a result, this directly influences how many non-zeros are assigned per thread. However, the block size (no. threads per block) is defined at compile time and is constant among all blocks of the grid. This required a more sophisticated strategy as to how we assign non-zeros to threads, since a one-to-one scheme would no longer work:

Listing 4.1: Thread-strided loop over nonzeros in a column. Note the iteration step.

```
for (size_t i = col_ptr[blockIdx.x] + threadIdx.x;
     i < col_ptr[blockIdx.x + 1];
     i += blockDim.x) {
                acc += x_row_smem[row_idx[i]] * val[i];
}
```

Each thread processes up to $\left\lfloor \frac{\text{col\_ptr}[i+1]-\text{col\_ptr}[i]}{d_b} \right\rfloor$ elements of the column, where d_b is the block dimension. Any remaining elements are handled by reusing the corresponding threads, while the unused threads ($d_b - (\text{col\_ptr}[i+1] - \text{col\_ptr}[i]) \mod d_b$ in size) are temporarily stalled until the last iteration of the loop completes (Figure 4.6).

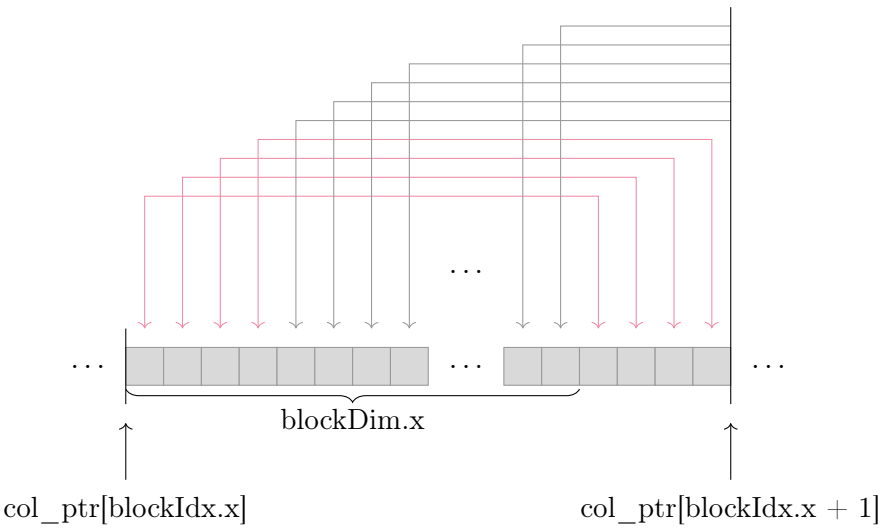

Figure 4.6. ***Thread-strided loop:*** *Red threads will get reused while the grey ones will stall.*

4.2.2 Partial results

As evident from Figure 4.6, threads might accumulate more than one pointwise multiplication. This signals that individual threads will need to synchronize and cooperate such that partial results are correctly accumulated and the final element of C is stored. While not inherent to sparsity, this challenge highlights the necessity for warp-level synchronization support by the runtime. CUDA offers warp-level primitives, which guarantee safe warp-synchronous programming. These intrinsics permit the exchange of a variable

between threads within a warp without use of shared memory. The exchange happens simultaneously for all active threads within the warp [9]. The current kernel implementation first performs a warp-wide reduce, accumulating all partial results of individual warps, then a block-wide reduce to get the final result.

Figure 4.7 shows the performance benchmarks of this kernel against cuSPARSE and against the previous kernel. We have lost any and all performance leads for higher sparsities, but gained a speedup at lower sparsities against the previous kernel.

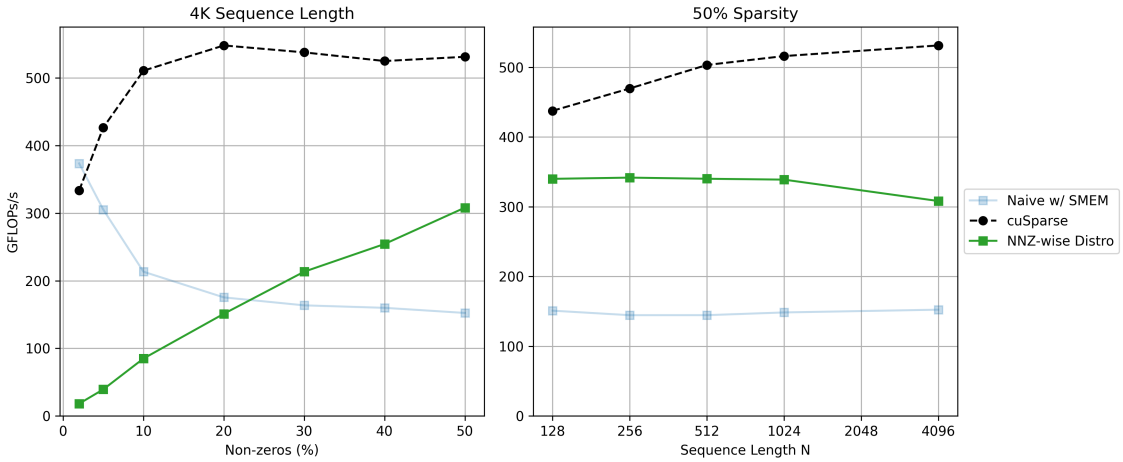

Figure 4.7. ***Nonzero-wise distro w/ coalesced memory access: Left.*** *Variable sparsity at 4K input sequence length.****Right.*** *Variable sequence length @50% sparsity.*

Results of Figure 4.7 highlight the importance of following different approaches when it comes to sparsity.

4.3 Kernel 3: Minimizing shared memory usage and restoring L1 cache functionality

As discussed in 3.1 shared memory is an abstraction of the L1 cache and as such, inefficient usage of shared memory cripples L1 cache functionality, that is, keeping reusable data on a low latency close-to-the-cores storage. This kernel's one and only change is the minimization of shared memory usage with utilization of it remaining only when absolutely necessary (reduction step).

For 50% sparsity, results are trailing cuSPARSE's implementation, and we see a minor improvement for higher sparsities. However, the L1 hit rate improvement was smaller than anticipated, reaching $\sim 55\%$ from the $\sim 47\%$ of the Kernel 2. The large leap in performance is due to the fact the we no longer waste memory bandwidth on unused bytes. With this kernel, the two main issues discussed in the first kernel's implementation are largely solved.

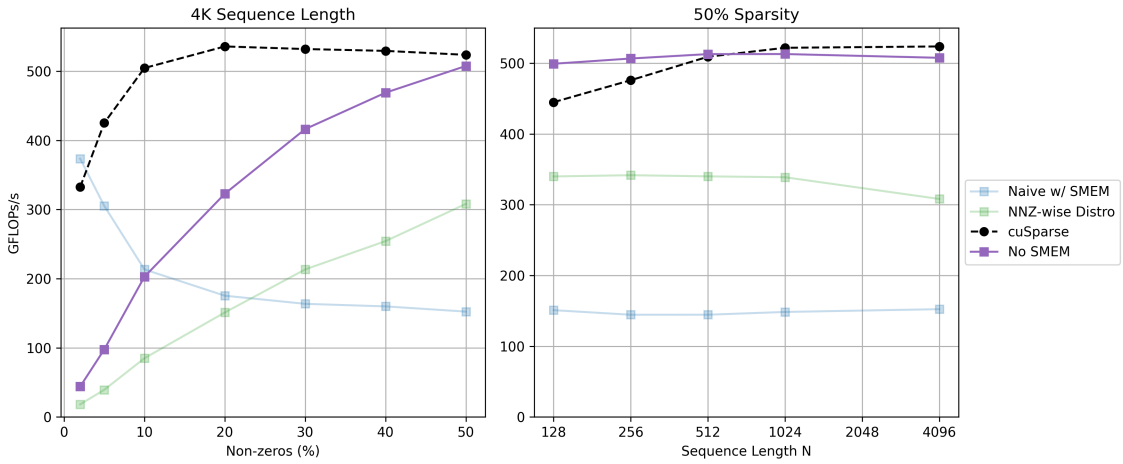

Figure 4.8. ***No shared memory: Left.*** *Variable sparsity at 4K input sequence length.* ***Right.*** *Variable sequence length @50% sparsity.*

4.4 Kernel 4: Vectorized Memory Access with Adaptive Nonzero Tiling

Vectorized memory access is a pretty common performance optimization technique and this kernel contributes just that, trying to maximize memory bandwidth by making the necessary changes to adhere to the strict alignment requirements to utilize vectorized memory access. In addition, it allows for ≥ 1 number of blocks to be assigned to the non-zeros of each column, through both thread-strided and block-strided loops. I'll analyze how these new characteristics are implemented and their impact on performance, but first, the distribution of work: the x and y dimension of the launched grid is distributed uniformly across the columns and rows respectively. The z dimension of the grid is distributed across the non-zeros of the corresponding column (Figure 4.9). For reasons discussed in Kernel 3, no shared memory is used.

4.4.1 Vector Data Types and Alignment

The aforementioned *LDG.E* instruction loads (or *STG.E* for stores) 32 bits from global memory [16]. Vectorized load instruction *LDG.E.{64, 128}* (or *STG.E.{64, 128}*), does so in 64 or 128 bits. This reduces the total number of instructions, latency and improves bandwidth utilization. There are a couple of tradeoffs, however: Firstly, we increase the number of utilized registers, possibly descreasing occupancy as a result. Less important to performance and more of a development overhead, vector data types require pointers, pointing to naturally aligned memory addresses. That is, the data must be stored in addresses multiples of their size in bytes.

Before discussing alignment, I will provide an overview of how input and output is laid out in memory, both in the CPU and the GPU. The allocation scheme I chose follows a few principles:

1. There must only be a *single* CPU and a *single* GPU memory allocation. This implies

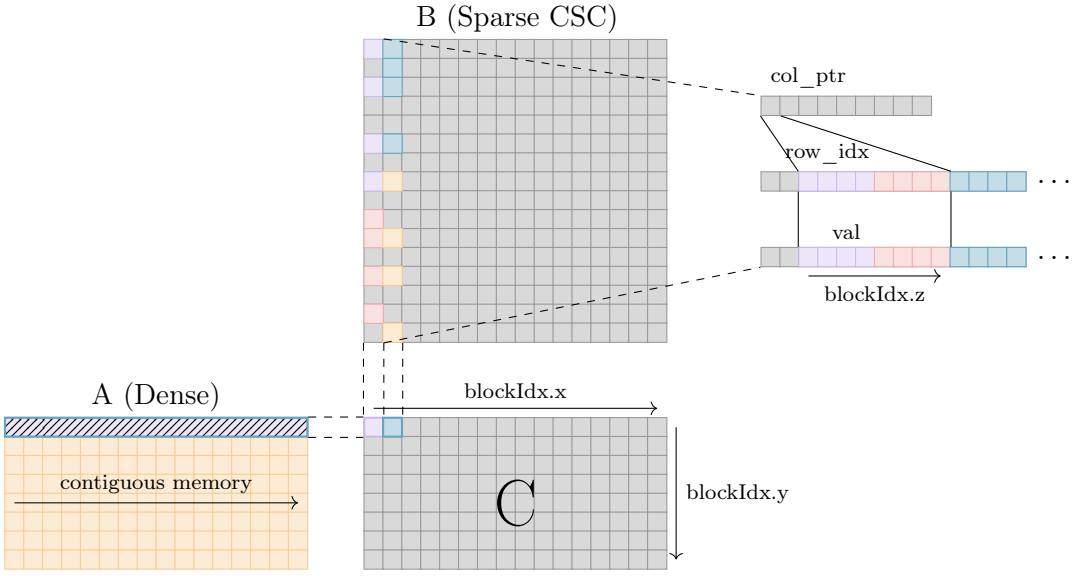

Figure 4.9. *gridDim.z thread blocks (colored) compute an element of C.*

that there is a single *deallocation* of memory for each.

2. There must only exist one CPU to GPU memory transfer for the input and one for the output.

System calls for allocation and deallocation, as well as, CPU to GPU memory transactions are very costly with regards to time. Hence it is imperative to perform as few of them as possible. As an added benefit, a single allocation provides (guaranteed by the operating system/GPU driver) contiguous memory. This will increase data locality as a function of sparsity. Allocation scheme showcased in Figure 4.10.

A_1	$\cdots$	A_n	*col_ptr*	*row_idx*	*val*	C_1	$\cdots$	C_n

Figure 4.10. *n dense matrices of size $m \times k$, where $m = 2^v : v \in \mathbb{N}$ and $k = N$, where N the input sequence length. Following that, the sparse matrix's meta arrays, col_ptr, row_idx, val. Finally, the output C_1 to C_n.*

Keeping matrix sizes of A_1 to A_n as powers of two, guarantees that the pointer *col_ptr** is aligned. However, alignment possibly breaks with the pointers of *row_idx* and *val* since the size of *col_ptr* is equal to $N+1$ and *row_idx* is equal to the number of non-zeros. This possible misalignment is caught and handled during the allocation phase with sufficient use of padding. Remaining ($nnz \mod 4$)[1] elements are handled with scalar loads and the rest are handled with vectorized loads (Figure 4.11).

4.4.2 Adaptive Block Tiling

In addition to stepping by a thread-stride (Figure 4.6) in the main loop, each block's range of work inside the meta arrays of the sparse matrix was calculated beforehand for

[1]The kernel uses the 128-bit version of the instruction.

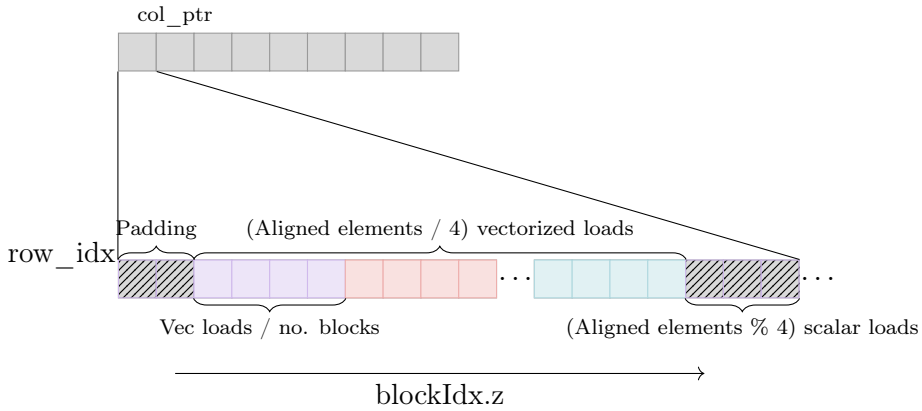

Figure 4.11. *Let nnz be the non-zeros of a column. Blocks (colored) are assigned v/gridDim vectorized loads, where v = nnz/4 and gridDim the grid dimension. Any remainder is handled with scalar loads by the first block of that column group (blockIdx.z = 0)*

each, such that it was split among them as evenly as possible. That way, the number of blocks along a column was configurable and the work was adaptively split among them.

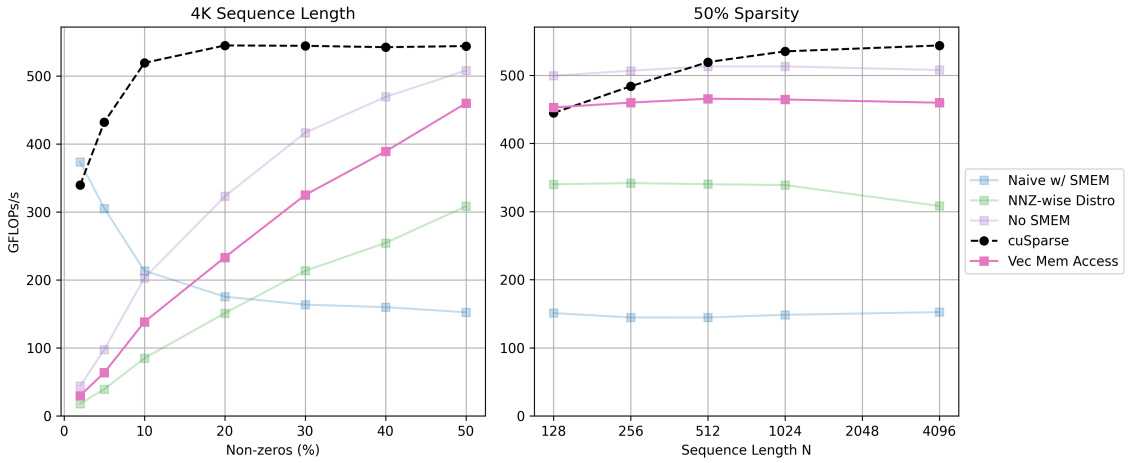

Figure 4.12. ***Vectorized w/ Adaptive block tiling: Left.*** *Variable sparsity at 4K sequence length.* ***Right.*** *Variable sequence length @50% sparsity.*

Benchmarking results (Figure 4.12) show a notable performance degradation. The bandwidth gained from vectorizing memory access is negligible at such small total bytes transferred (our sparse matrices are 512×512) and does not outweigh the overhead introduced. Also the adaptive block tiling helped uncover another intrinsic property of the operation. The best results (shown in Figure 4.12) are when we assign only one block per column, that is, not use adaptive block tiling at all. For bigger z-dimension grids, performance degrades exponentially.

4.5 Kernel 5: Column Tiling w/ Nonzero-wise work distribution

Although adaptive block tiling may have yielded suboptimal benchmark results, it nevertheless suggests that we may be assigning *too little* work per block. Hence, instead of splitting non-zeros of a single column to more blocks, this kernel does the opposite: it splits one block to multiple columns-worth of non-zeros, controlled by a parameter BN. An immediate benefit of this appraoch is a better L1 cache hit rate, as the elements fetched from the correposnding row of A, are reused for any other columns that block has been assigned to (Figure 4.13).

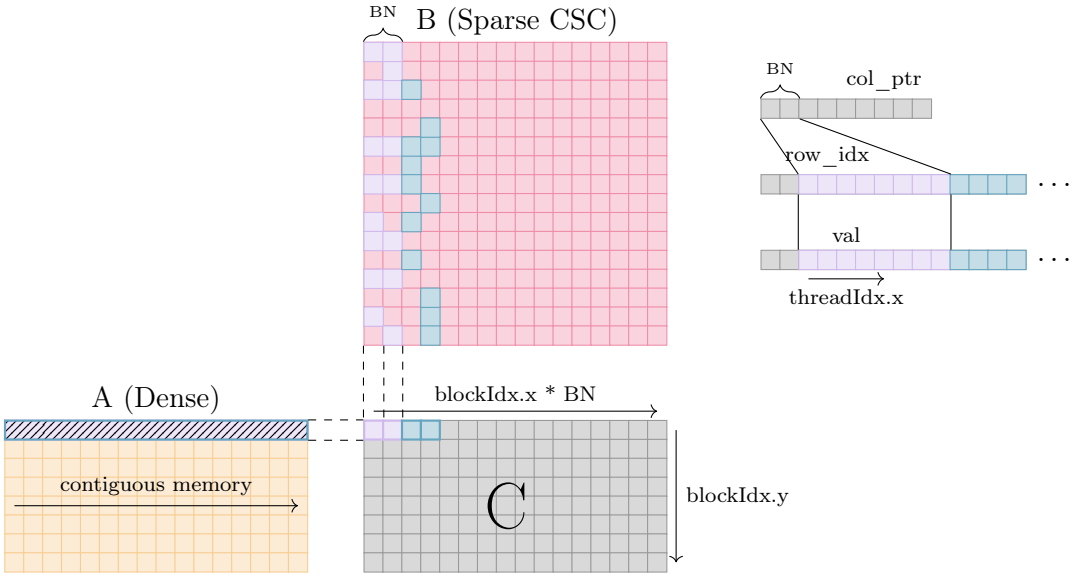

Figure 4.13. *A block (colored) is assigned BN columns*

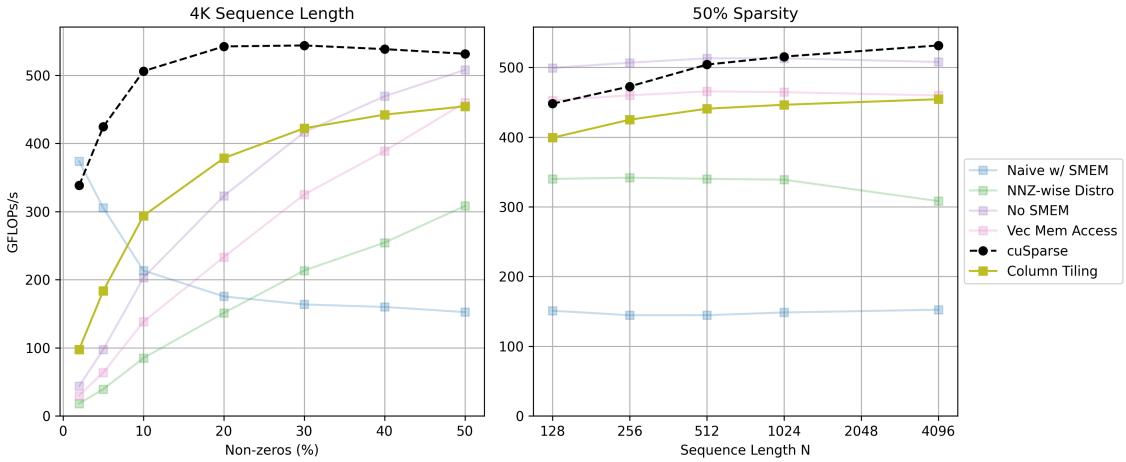

Figure 4.14. ***Column tiling: Left.*** *Variable sparsity at 4K sequence length.* ***Right.*** *Variable sequence length @50% sparsity.*

Figure 4.14 shows that we have managed to smooth out the performance for higher sparsities, indicating the low performance at higher sparsities might be a result of the

overhead introduced with over-launching blocks of threads.

4.6 Kernel 6: Block Tiling w/ Element-wise work distribution

A compilation of previous benchmarks allows us to pinpoint the following reasons for performance fluctuations:

1. Work distribution scheme (amplified by sparsity levels).
2. Wasted memory bandwidth.
3. L1 cache functionality retention (keep *only* fully reusable data in shared memory).

This kernel pays attention to all three by tiling work in a block. Block tiling is done on the element-level of the resulting C matrix and each block is assigned $BK \times BN$ elements which then subsequently distributes on a $1:1$ ratio to its threads. This increases L1/L2 cache hit to a range of $95\% - 99\%$ due to how matrix multiplication reuses elements in a block, making this the best performing arrangement out of all kernels. Note that, this kernel accesses global memory in a way that makes it hard for the hardware to coalesce the transaction to as few as possible. The first kernel suffered from the same issue and, as discussed previously, this happens due to how each thread of a warp doesn't read from neighbouring memory locations. Work distribution is shown in Figure 4.15.

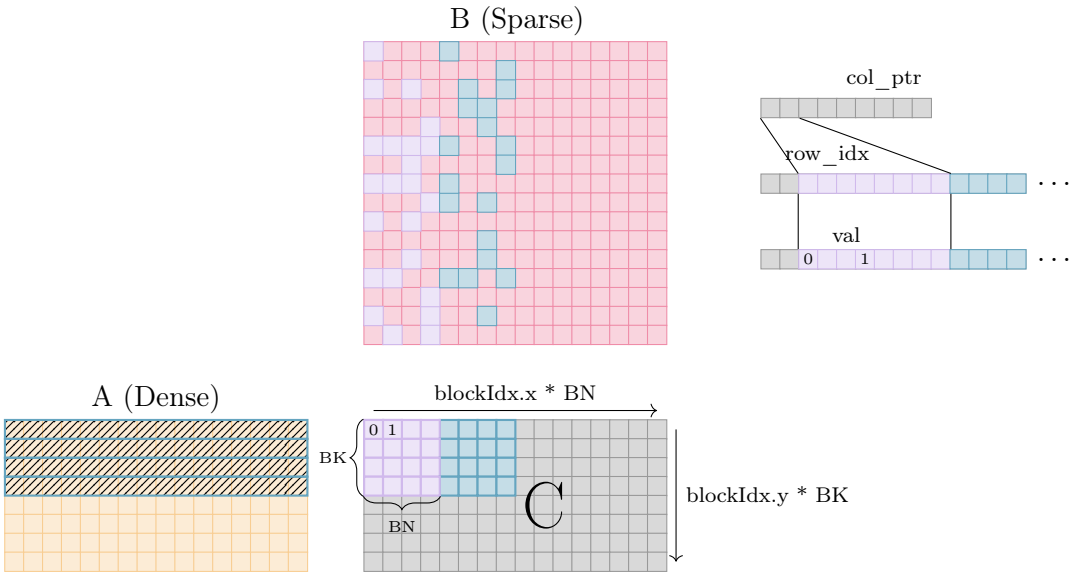

Figure 4.15. *Threads, grouped by block (colored) are assigned a $BK \times BN$ grid of elements to process, however threads of warp (numbered) read distant locations in memory (numbered).*

Figure 4.16 shows a massive leap between cuSPARSE's implementation at more dense-like matrices achieving a speedup of $\sim 57\%$ but above the 90% sparsity mark, performance still suffers comparatively. Metrics will be discussed in more detail in Chapter 5.

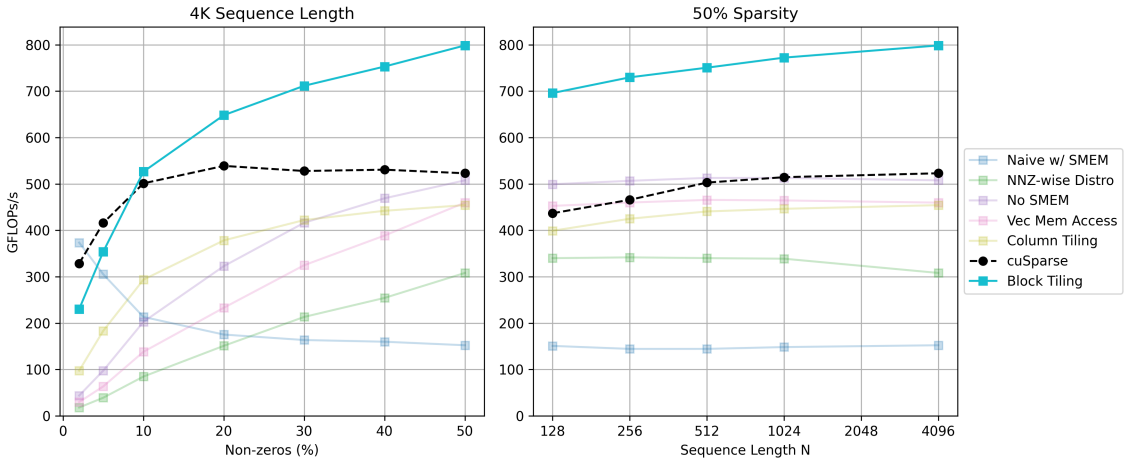

Figure 4.16. ***Block tiling: Left.*** *Variable sparsity at 4K sequence length.* ***Right.*** *Variable sequence length @50% sparsity.*

5 Experimental Evaluation

5.1 Hardware Specifications

Evaluation of the kernels was performed on the commercial NVIDIA RTX 4070Ti Super. I provide some relevant properties for this GPU model:

Metric	Value
Architecture	Ada Lovelace
Compute Capability	8.9
FP32	44.10 TFLOPs/s
FP64	689.0 GFLOPs/s
Max Threads per Block	1024
Max Threads per SM	1536
Threads per Warp	32
Max Registers per Block	65536
Max Registers per SM	65536
Total Global Memory	16714 MB
Memory Type	GDDR6X
Global Memory Bandwidth	672.3 GB/s
Max Shared Memory per Block	49KB
Max Shared Memory per SM	102400B
SM Count	66
Theoretical Active Warps per SM	48

5.2 Input Data

Input weight matrices W^Q, W^K, W^V where provided by the Deep Learning Matrix Collection [7] (DLMC). Benchmarks were performed for $50\%, 60\%, 70\%, 80\%, 90\%, 95\%$ and 98% sparsities for each of the following sparsification methods: *L0 Regularization, Magnitude Pruning, Random Pruning* and *Variational Dropout*.

First, a warmup phase was conducted and results were verified against cuSPARSE's SpMM implementation. Afterwards, kernels were executed for a thousand times, measuring total wall-clock time. Mean time and FLOPs/s were subsequently calculated. Note that, the figures of Chapter 4 showcase only the L0 Regularization pruning method since results were pretty similar between all of them. However, full benchmarking results are provided in Section 5.3.3.

5.3 Benchmarks

5.3.1 cuSPARSE

cuSPARSE is NVIDIA's closed source basic linear algebra accelerator library for sparse matrices. It targets matrices of the sparsity range 70%-99.9% and is designed to perform well on a variety of hardware and fields.

5.3.2 Profiler metrics

I will present and analyze metrics profiled by NVIDIA's Nsight compute of each kernel against cuSPARSE's implementation and the best in work (BiW), Kernel 6, as measured for the *L0 Regularization* pruning method at 4K sequence length mainly at 50% sparsities, showing off 98% sparsity metrics only in remarkable performance changes as is the case with Kernel 1 and Kernel 6.

Kernel 1: Element-wise w/ SMEM

This kernel has a *Warp Cycles Per Issued Instruction* of 533.54 in cycles, compared to cuSPARSE's 20.29 and the BiW of 43.51 at 50% sparsity. Most stalls are of type: *Stall MIO Throttle* and *Stall LG Throttle*, indicating that threads are waiting for the instruction queues of the corresponding GMEM and L1/TEX cache operations to complete. At 98% sparsity, the warp cycles are reduced to 28.72 per issued instruction. This is expected, since at higher sparsities less total memory transactions are required, hence the memory queue can handle the small number of memory instructions. The number of executed instructions are summed to 113,340,416 against cuSPARSE's 851,856,384 and the BIW's of 217,650,176. A similar ratio is maintained and at 98% sparsity. Grid size is the smallest amongst all at 4,096 while cuSPARSE dominates at 154,880 blocks while BiW is at 32,768. This kernel achieves the highest occupancy across all three at 96.11% (95.30% cuSPARSE, 79.97% BiW). Uncoalescing is prevalent, as discussed, resulting in 87% excessive sectors while cuSPARSE has the best memory access pattern at 39% excessive sectors and the BiW is at 64%. cuSPARSE's not predicated off threads per warp (avg.) is at 23.26 out of the 32 threads of the warp. Meanwhile this implementation achieves a 27.60 on average while the BiW a 29.10. This warp divergence is possibly due to some control sequence that acts as a catch-all solution in the cuSPARSE implementation, totalling 76,669.09 divergence branches (avg.) compared to this implementation's 2,156.61 and the BiW's 1,648.48 (avg.). Finally L1 cache hit rate is at 68.07% compared to cuSPARSE's 73.60% and BiW's 94.81%. L2 cache hit rate is at $\sim$ 99% across the board.

Kernel 2: NNZ-wise with coalesced mem access

Warp Cycles Per Issued Instruction have been massively reduced, down to 30.61 per cycle (cuSPARSE 20.29, BiW 43.51). The majority of stalls are of type *Long Scoreboard*, which means that threads are waiting on a memory operation dependency to complete. Number of executed instructions increased dramatically to 947,478,528 (cuSPARSE

851,856,384 & BiW 217,650,176) with more than half of them being for integer operations. Grid size is up to 2,097,152 (cuSPARSE 154,880, BiW 32,768) due to how we do a $1:1$ assignment of blocks to elements of C. Occupancy is similar to the previous kernel, measured at 94.22% (cuSPARSE 95.30%, 79.97%). Only 10% excessive sectors fetched (cuSPARSE 39%, BiW 64%). 30.03 avg. threads per warp are not predicated off, having the highest convergence so far on a warp-scope, however the average number of divergent branches is up to 7,664.48. L1 cache hit rate has dropped to 45.83%, since block-wide reusability is minimized with the current work distribution scheme. L2 cache hit rate similar to the previous kernel.

Kernel 3: No SMEM

Comparable *Warp Cycles Per Issued Instruction* to previous implementations at 27.80 cycles, with the majority still being of type *Long Scoreboard*. The number of executed instructions reduced, down to 585,949,184 due to how we don't perform any shared memory stores (*STS*) instructions. The grid size remained the same, as we didn't change work distribution in this kernel. Occupancy dropped a bit (down to 89.21%) due to the increased register demands, now that we don't utilize shared memory. Excessive sectors increased (29%), as we don't access the corresponding x row of A in a contiguous fashion, instead only request random addresses dictated by the row of the corresponding non-zeros. Hit rate increased to 52.99% due to restoring L1 cache functionality but not as much as expected. L2 cache hit rate similar.

Kernel 4: Vectorized Mem Access

Warp Cycles per Issued Instruction at the lowest point of 12.68. However, the complex control scheme of handling unaligned and remainder element loads as scalar while vectorizing the main bulk of the data caused a lot of divergent warp (only 18.63 of threads are not predicated off on average) and a lot of average divergent branches in total (15,653.79). Whilst previous kernels had a theoretical occupancy of 100%, derived from the amount of GPU resources they required, this kernel's theoretical occupancy dropped to 50% primarily due to the amount of registers used to service the vectorized loads, with an achieved occupancy of 47.41%. Once again we have an identical work distribution (at least when not using any adaptive tiling), hence the same grid size as the previous kernel. Number of executed instructions jumped to 770,789,376 due to the vectorized memory access and all the control sequences (alignment, scalar loads) that it required. This irregular access pattern could not be coalesced into as few memory transactions as possible, resulting in 61% excessive sectors.

Kernel 5: Column Tiling

Warp Cycles Per Issued Instruction increased to 22.45 cycles, losing to cuSPARSE's 20.29 cycles. Warp Divergence is minimized with only 2.7 threads on average being predicated off per warp. Removing vectorized access meant reducing the excessive number

instructions of the previous kernel, and as such we are now down to 504,029,184 instructions lower than cuSPARSE's 851,856,384 but higher than BiW's 217,650,176. The number of blocks decreased to 131,072 hitting one this kernel's marks, i.e. giving more work to blocks. More work however requires more registers, and the 27 registers per thread reduce the theoretical occupancy down to 50%. On an A100 or H100, this kernel might give better results, due to higher occupancy. Despite this, achieved occupancy comes very close to theoretical at 49.54%, possibly hinting at a performance gain on a non-commercial GPU.

Kernel 6: Block Tiling (BiW)

The biggest factor contributing to this kernel's performance is the L1/L2 cache hit rate, reaching a 94.81% hit rate for L1 and a 99.96% for L2, toppling cuSPARSE's 73.60% and 98.28% respectively. This kernel's total instructions are at 217,650,176 which is lower than cuSPARSE. This kernel wins on grid size as well, having the smallest across all at 32,768 blocks due to how work is partitioned into blocks, incurring the least amount of overhead. Theoretical occupancy is at 83.33% limited mainly by the number of registers per thread (using 48, would need less than 40 for 100% theoretical occupancy). However, having less than perfect accurary can sometimes be beneficial to performance [17]. As discussed in Section 4.6, our global memory access pattern leaves little room for coalescing and this is supported by the reported 64% excessive sectors fetched from memory.

5.3.3 Full Benchmarks

Here I present the full benchmarking results for each of the kernels previously discussed. The results are grouped by sparsity and pruning technique and both wall-clock time measure and GFLOPs/s are included.

	Time (ms)							GFLOPs/s						
Sparsity	50%	60%	70%	80%	90%	95%	98%	50%	60%	70%	80%	90%	95%	98%
cuSPARSE														
L0 Regularization	0.080	0.056	0.044	0.030	0.020	0.013	0.010	437.388	487.792	475.902	444.983	355.970	230.696	128.271
Magnitude	0.069	0.056	0.042	0.030	0.018	0.014	0.010	487.897	485.900	479.877	449.278	367.499	246.251	131.250
Random	0.068	0.056	0.043	0.030	0.019	0.014	0.011	492.696	483.872	472.744	446.272	359.733	245.862	128.220
Variational Dropout	0.073	0.061	0.048	0.033	0.023	0.015	0.011	497.040	488.511	478.689	440.901	365.107	271.415	168.597
Naive w/ Shared Memory														
L0 Regularization	0.234	0.179	0.138	0.088	0.045	0.018	0.007	150.956	151.704	152.300	158.376	175.723	199.093	209.949
Magnitude	0.221	0.176	0.135	0.086	0.041	0.013	0.005	152.248	153.574	150.313	157.073	165.243	261.224	257.362
Random	0.219	0.178	0.141	0.088	0.042	0.013	0.005	153.812	152.128	143.614	153.609	162.182	264.618	283.247
Variational Dropout	0.244	0.201	0.155	0.096	0.057	0.026	0.010	149.812	148.152	149.040	153.215	179.647	215.627	221.277
Nonzero-wise Distribution														
L0 Regularization	0.104	0.096	0.088	0.082	0.078	0.075	0.073	340.007	283.714	237.977	166.135	94.302	44.824	20.418
Magnitude	0.100	0.094	0.088	0.083	0.077	0.075	0.073	338.614	289.172	229.800	163.246	88.232	45.300	18.521
Random	0.099	0.095	0.089	0.084	0.077	0.074	0.073	340.167	284.126	227.896	160.645	87.871	45.656	18.537
Variational Dropout	0.102	0.097	0.090	0.082	0.080	0.074	0.072	358.651	306.126	255.843	176.739	111.199	61.315	29.841
No Shared Memory														
L0 Regularization	0.070	0.058	0.051	0.042	0.037	0.036	0.035	499.351	463.895	405.344	317.880	194.928	93.551	42.319
Magnitude	0.066	0.057	0.050	0.042	0.036	0.036	0.035	510.873	476.763	408.414	320.885	187.620	94.683	38.246
Random	0.066	0.058	0.050	0.042	0.036	0.036	0.036	507.781	466.954	400.907	317.835	187.069	94.310	37.717
Variational Dropout	0.070	0.061	0.053	0.043	0.039	0.036	0.035	518.121	477.824	430.674	330.985	227.078	128.474	62.058
Vectorized Memory Access														
L0 Regularization	0.078	0.072	0.065	0.060	0.055	0.053	0.051	452.694	377.732	317.653	226.549	133.589	61.284	28.657
Magnitude	0.078	0.072	0.069	0.059	0.057	0.056	0.054	433.604	375.805	293.779	230.048	118.221	59.789	25.167
Random	0.078	0.072	0.070	0.058	0.057	0.057	0.055	432.105	375.116	287.607	232.095	118.955	59.591	24.696
Variational Dropout	0.080	0.074	0.068	0.059	0.058	0.055	0.053	453.718	399.609	337.218	240.388	154.133	82.692	39.684
Column Tiling														
L0 Regularization	0.088	0.070	0.057	0.042	0.030	0.021	0.018	399.140	387.021	364.309	319.412	233.404	145.700	77.366
Magnitude	0.083	0.069	0.054	0.041	0.028	0.020	0.018	408.388	391.009	372.126	324.574	244.742	168.427	72.896
Random	0.083	0.070	0.054	0.041	0.027	0.019	0.019	407.104	383.754	374.722	329.990	247.865	174.655	72.618
Variational Dropout	0.091	0.076	0.061	0.043	0.032	0.023	0.019	402.042	387.128	375.177	330.789	256.525	177.501	102.542
Block Tiling														
L0 Regularization	0.051	0.042	0.035	0.025	0.017	0.012	0.007	695.938	648.799	599.343	532.564	409.578	267.310	185.974
Magnitude	0.047	0.041	0.033	0.024	0.014	0.010	0.007	713.797	648.694	604.892	571.167	489.130	337.979	188.774
Random	0.046	0.040	0.033	0.023	0.013	0.009	0.007	738.504	675.878	613.045	585.954	509.918	373.323	200.303
Variational Dropout	0.052	0.045	0.037	0.026	0.020	0.013	0.008	706.963	650.494	616.249	538.750	421.296	312.370	227.392

Table 5.1. *Time in milliseconds and GFLOPs/s for all kernels at input sequence $N = 128$. Underlined values are best for that sparsity.*

	Time (ms)							GFLOPs/s						
Sparsity	50%	60%	70%	80%	90%	95%	98%	50%	60%	70%	80%	90%	95%	98%
cuSPARSE														
L0 Regularization	0.150	0.106	0.082	0.055	0.033	0.020	0.014	469.567	514.120	513.046	494.401	426.553	306.223	187.648
Magnitude	0.132	0.105	0.078	0.054	0.030	0.021	0.014	509.493	513.637	517.654	503.244	454.636	327.495	190.976
Random	0.131	0.106	0.080	0.054	0.030	0.021	0.014	515.514	509.895	506.012	498.939	445.793	321.899	189.957
Variational Dropout	0.140	0.116	0.091	0.059	0.040	0.024	0.015	520.342	513.658	510.214	488.263	434.100	343.882	239.989
Naive w/ Shared Memory														
L0 Regularization	0.491	0.369	0.278	0.174	0.088	0.033	0.012	144.485	148.092	151.354	160.458	185.370	225.572	263.206
Magnitude	0.468	0.362	0.270	0.171	0.080	0.024	0.008	143.976	149.052	150.126	157.698	167.840	286.623	332.358
Random	0.465	0.370	0.282	0.177	0.082	0.022	0.007	145.100	146.291	143.614	152.922	164.003	301.585	363.979
Variational Dropout	0.515	0.417	0.315	0.190	0.113	0.050	0.018	142.291	144.245	147.677	156.199	183.463	247.734	269.829
Nonzero-wise Distribution														
L0 Regularization	0.206	0.190	0.174	0.162	0.154	0.147	0.147	341.775	285.536	239.845	168.276	95.942	45.421	20.384
Magnitude	0.199	0.186	0.175	0.164	0.153	0.147	0.144	340.502	291.400	231.396	164.703	88.231	45.951	18.795
Random	0.198	0.189	0.176	0.166	0.152	0.146	0.144	342.440	286.460	230.169	162.521	88.882	46.264	18.722
Variational Dropout	0.202	0.192	0.180	0.161	0.158	0.147	0.143	361.975	308.070	256.092	179.269	112.502	61.985	30.255
No Shared Memory														
L0 Regularization	0.139	0.115	0.101	0.083	0.073	0.070	0.069	506.736	470.558	412.369	322.489	199.175	95.474	43.211
Magnitude	0.131	0.112	0.097	0.082	0.070	0.070	0.069	513.943	483.611	418.687	328.474	191.328	96.579	38.992
Random	0.132	0.114	0.099	0.084	0.071	0.070	0.070	512.690	472.961	408.420	322.877	189.598	96.288	38.571
Variational Dropout	0.139	0.121	0.104	0.084	0.076	0.070	0.069	523.783	483.890	440.024	337.942	231.109	131.393	62.811
Vectorized Memory Access														
L0 Regularization	0.153	0.140	0.129	0.118	0.108	0.104	0.100	459.933	385.837	322.156	229.641	135.971	62.794	29.371
Magnitude	0.153	0.141	0.136	0.114	0.112	0.111	0.105	439.538	382.600	296.871	236.113	120.736	61.045	25.771
Random	0.154	0.142	0.138	0.114	0.111	0.111	0.106	437.828	381.169	293.077	237.467	121.326	60.663	25.314
Variational Dropout	0.158	0.145	0.134	0.117	0.113	0.107	0.104	461.355	405.322	342.622	244.548	157.494	84.618	40.544
Column Tiling														
L0 Regularization	0.166	0.131	0.106	0.077	0.053	0.039	0.033	424.975	416.365	395.438	351.036	261.981	160.774	84.682
Magnitude	0.155	0.128	0.101	0.076	0.050	0.036	0.034	435.595	422.041	401.946	354.552	271.230	186.311	80.424
Random	0.155	0.130	0.098	0.074	0.049	0.035	0.034	433.760	416.185	411.395	363.911	274.891	190.460	80.107
Variational Dropout	0.170	0.142	0.113	0.079	0.059	0.043	0.034	429.341	418.111	406.366	361.916	283.031	195.056	114.234
Block Tiling														
L0 Regularization	0.097	0.078	0.065	0.046	0.031	0.020	0.013	729.747	693.286	642.608	578.873	457.410	307.091	206.883
Magnitude	0.090	0.078	0.062	0.044	0.025	0.018	0.013	749.471	685.970	646.616	611.088	533.444	370.328	207.864
Random	0.087	0.076	0.063	0.043	0.024	0.017	0.013	772.919	708.651	643.716	622.117	550.910	387.151	212.493
Variational Dropout	0.099	0.085	0.070	0.047	0.035	0.023	0.014	738.482	691.624	652.708	600.483	479.510	360.037	261.810

Table 5.2. *Time in milliseconds and GFLOPs/s for all kernels at input sequence $N = 256$. Underlined values are best for that sparsity.*

	Time (ms)							GFLOPs/s						
Sparsity	50%	60%	70%	80%	90%	95%	98%	50%	60%	70%	80%	90%	95%	98%
cuSPARSE														
L0 Regularization	0.281	0.206	0.158	0.103	0.060	0.033	0.020	501.613	528.910	532.434	529.399	475.518	366.591	251.046
Magnitude	0.258	0.205	0.151	0.101	0.053	0.034	0.021	522.431	526.235	535.449	533.313	509.649	394.862	262.412
Random	0.258	0.207	0.154	0.102	0.054	0.035	0.021	523.016	521.634	524.187	527.204	500.881	387.915	257.326
Variational Dropout	0.273	0.225	0.175	0.113	0.074	0.042	0.024	534.172	528.511	528.679	515.344	479.403	397.955	302.500
Naive w/ Shared Memory														
L0 Regularization	0.982	0.737	0.556	0.342	0.172	0.061	0.021	144.525	148.610	152.374	163.355	197.142	269.328	315.106
Magnitude	0.936	0.725	0.526	0.336	0.157	0.041	0.014	144.176	148.931	154.027	160.685	171.582	328.162	392.400
Random	0.935	0.746	0.547	0.347	0.163	0.041	0.012	144.412	144.850	148.104	155.900	166.041	327.459	435.480
Variational Dropout	1.030	0.830	0.623	0.371	0.220	0.094	0.033	142.372	145.381	149.389	159.841	203.317	291.279	308.864
Nonzero-wise Distribution														
L0 Regularization	0.415	0.382	0.348	0.325	0.308	0.298	0.293	340.137	284.994	240.229	168.180	96.186	44.823	20.281
Magnitude	0.397	0.372	0.350	0.328	0.302	0.295	0.287	341.114	290.830	232.050	165.277	89.719	45.928	18.847
Random	0.395	0.377	0.353	0.333	0.304	0.292	0.289	342.536	286.749	230.336	162.592	89.034	46.295	18.726
Variational Dropout	0.412	0.385	0.355	0.324	0.318	0.294	0.286	355.210	307.611	258.635	177.891	111.792	62.114	30.268
No Shared Memory														
L0 Regularization	0.274	0.228	0.199	0.165	0.145	0.138	0.136	512.989	474.162	417.159	326.339	201.461	96.763	43.980
Magnitude	0.259	0.223	0.193	0.163	0.140	0.138	0.137	520.712	484.446	419.879	330.681	193.289	97.728	39.465
Random	0.262	0.227	0.196	0.165	0.141	0.139	0.138	515.468	475.337	412.783	327.562	190.827	96.726	39.004
Variational Dropout	0.277	0.241	0.207	0.165	0.150	0.139	0.137	526.825	486.512	441.679	341.309	234.875	132.738	63.698
Vectorized Memory Access														
L0 Regularization	0.302	0.279	0.256	0.234	0.214	0.209	0.197	465.569	388.297	325.063	230.738	137.834	63.290	29.616
Magnitude	0.304	0.280	0.271	0.226	0.221	0.218	0.207	442.696	384.783	298.645	238.223	122.033	61.749	26.106
Random	0.307	0.281	0.272	0.225	0.220	0.220	0.211	438.868	384.089	296.832	239.701	122.975	61.315	25.606
Variational Dropout	0.314	0.288	0.264	0.230	0.223	0.212	0.205	464.545	407.972	346.007	247.072	159.209	85.503	41.055
Column Tiling														
L0 Regularization	0.320	0.253	0.203	0.147	0.100	0.072	0.060	440.732	429.595	410.611	366.856	279.191	173.584	92.166
Magnitude	0.301	0.247	0.194	0.145	0.093	0.066	0.062	448.194	435.764	416.998	372.766	291.231	203.558	87.220
Random	0.302	0.247	0.189	0.141	0.091	0.065	0.062	446.575	436.798	427.671	382.059	296.901	205.742	87.106
Variational Dropout	0.330	0.274	0.219	0.150	0.112	0.080	0.063	442.565	432.628	421.284	379.490	299.622	208.954	123.374
Block Tiling														
L0 Regularization	0.188	0.150	0.123	0.088	0.058	0.038	0.024	750.560	719.990	675.905	613.844	484.242	326.079	219.374
Magnitude	0.174	0.151	0.120	0.084	0.052	0.034	0.025	775.869	712.819	673.639	639.822	519.444	396.736	215.497
Random	0.169	0.146	0.120	0.083	0.051	0.033	0.025	800.111	736.284	673.980	649.838	532.139	413.921	213.598
Variational Dropout	0.190	0.163	0.135	0.090	0.065	0.043	0.027	766.954	719.148	682.526	629.929	513.210	379.816	278.549

Table 5.3. *Time in milliseconds and GFLOPs/s for all kernels at input sequence $N = 512$. Underlined values are best for that sparsity.*

	Time (ms)							GFLOPs/s						
Sparsity	50%	60%	70%	80%	90%	95%	98%	50%	60%	70%	80%	90%	95%	98%
cuSPARSE														
L0 Regularization	0.548	0.407	0.309	0.201	0.115	0.061	0.035	514.532	536.341	542.689	545.528	500.953	401.067	292.193
Magnitude	0.511	0.404	0.299	0.197	0.102	0.061	0.036	528.040	534.425	540.833	548.740	530.237	441.137	302.669
Random	0.512	0.410	0.302	0.200	0.104	0.062	0.037	527.020	527.354	535.824	539.447	519.539	437.612	293.378
Variational Dropout	0.541	0.445	0.345	0.220	0.142	0.078	0.043	539.097	535.401	536.503	530.085	503.911	434.263	344.555
Naive w/ Shared Memory														
L0 Regularization	1.914	1.427	1.074	0.659	0.331	0.118	0.039	148.432	153.709	157.826	169.623	205.775	284.957	342.028
Magnitude	1.814	1.421	1.013	0.648	0.305	0.076	0.026	148.768	151.957	160.077	166.876	176.940	353.820	433.397
Random	1.818	1.446	1.043	0.672	0.317	0.076	0.022	148.683	149.583	155.433	160.842	170.626	355.904	482.968
Variational Dropout	2.013	1.593	1.202	0.717	0.422	0.180	0.063	145.418	151.327	154.916	165.948	214.322	311.864	336.078
Nonzero-wise Distribution														
L0 Regularization	0.833	0.773	0.706	0.658	0.629	0.607	0.594	338.882	281.938	237.500	166.914	93.794	44.085	20.183
Magnitude	0.803	0.759	0.709	0.666	0.611	0.597	0.580	337.472	285.810	229.720	163.012	88.778	45.462	18.702
Random	0.804	0.766	0.713	0.678	0.615	0.592	0.585	337.520	283.041	228.270	159.910	88.169	45.843	18.560
Variational Dropout	0.835	0.780	0.710	0.655	0.643	0.595	0.579	350.438	304.369	259.265	175.781	111.283	61.456	29.997
No Shared Memory														
L0 Regularization	0.548	0.460	0.401	0.328	0.288	0.275	0.272	513.069	471.076	413.461	327.154	202.182	96.972	44.099
Magnitude	0.519	0.445	0.385	0.324	0.277	0.274	0.272	520.043	484.614	420.447	333.378	194.969	98.284	39.700
Random	0.524	0.454	0.391	0.329	0.282	0.278	0.275	515.213	475.661	413.860	328.646	191.515	96.931	39.160
Variational Dropout	0.556	0.483	0.411	0.330	0.298	0.276	0.269	524.491	486.467	444.398	342.757	235.240	133.398	64.559
Vectorized Memory Access														
L0 Regularization	0.606	0.555	0.510	0.464	0.424	0.413	0.391	464.480	389.595	326.021	233.028	139.044	63.519	29.782
Magnitude	0.606	0.560	0.537	0.450	0.441	0.434	0.411	444.591	385.082	301.069	240.029	122.387	62.230	26.260
Random	0.612	0.562	0.539	0.448	0.436	0.435	0.417	440.497	384.010	299.955	240.783	123.725	61.959	25.854
Variational Dropout	0.630	0.573	0.524	0.459	0.445	0.421	0.408	462.760	409.644	349.048	247.991	159.942	85.922	41.321
Column Tiling														
L0 Regularization	0.631	0.499	0.399	0.286	0.194	0.139	0.116	446.400	435.916	418.502	375.724	287.528	178.965	95.426
Magnitude	0.596	0.488	0.382	0.283	0.180	0.128	0.119	452.613	442.141	423.334	381.050	300.023	210.318	90.308
Random	0.592	0.486	0.374	0.278	0.176	0.127	0.119	455.843	444.463	432.915	387.402	305.475	212.368	90.226
Variational Dropout	0.653	0.540	0.429	0.294	0.217	0.154	0.120	447.578	438.356	429.468	388.561	308.380	216.711	128.520
Block Tiling														
L0 Regularization	0.365	0.295	0.240	0.170	0.110	0.071	0.046	772.238	734.628	693.579	635.641	510.544	342.537	226.873
Magnitude	0.340	0.296	0.236	0.164	0.100	0.066	0.049	793.089	728.841	686.186	658.484	540.194	406.955	219.915
Random	0.332	0.287	0.235	0.163	0.098	0.065	0.050	812.089	751.956	688.786	662.114	549.903	416.768	215.348
Variational Dropout	0.374	0.320	0.264	0.176	0.126	0.084	0.051	780.236	734.492	696.674	648.412	533.525	394.203	291.288

Table 5.4. *Time in milliseconds and GFLOPs/s for all kernels at input sequence $N = 1024$. Underlined values are best for that sparsity.*

	Time (ms)							GFLOPs/s						
Sparsity	50%	60%	70%	80%	90%	95%	98%	50%	60%	70%	80%	90%	95%	98%
cuSPARSE														
L0 Regularization	2.107	1.618	1.223	0.792	0.446	0.227	0.125	535.844	539.899	549.469	554.020	519.269	433.785	336.324
Magnitude	2.033	1.607	1.183	0.775	0.395	0.228	0.129	530.737	537.418	547.270	556.707	546.011	472.575	334.137
Random	2.036	1.617	1.198	0.785	0.403	0.228	0.133	529.728	534.320	540.725	550.303	535.855	472.461	325.316
Variational Dropout	2.164	1.778	1.368	0.863	0.551	0.300	0.157	539.488	536.433	542.043	541.804	523.072	456.979	386.475
Naive w/ Shared Memory														
L0 Regularization	7.448	5.478	4.141	2.550	1.285	0.450	0.148	152.367	159.928	163.615	175.382	213.349	305.258	373.600
Magnitude	7.075	5.442	3.903	2.519	1.197	0.285	0.093	152.440	158.628	166.014	171.415	180.631	378.461	478.711
Random	7.103	5.592	4.022	2.594	1.226	0.288	0.081	152.098	154.543	161.082	166.704	176.217	375.666	533.676
Variational Dropout	7.828	6.164	4.665	2.773	1.631	0.698	0.238	149.397	156.582	160.509	170.627	224.746	335.326	363.195
Nonzero-wise Distribution														
L0 Regularization	3.709	3.477	3.194	2.986	2.881	2.824	2.754	308.190	254.373	213.417	151.132	84.925	39.274	18.181
Magnitude	3.632	3.399	3.154	2.982	2.738	2.750	2.644	303.612	261.550	211.353	148.968	81.325	40.570	16.841
Random	3.601	3.401	3.182	3.047	2.787	2.722	2.685	307.788	260.720	208.955	145.553	79.887	40.852	16.641
Variational Dropout	3.684	3.484	3.055	3.004	2.879	2.740	2.700	322.529	278.002	245.951	155.126	102.532	55.403	26.241
No Shared Memory														
L0 Regularization	2.219	1.848	1.593	1.330	1.151	1.095	1.092	507.748	469.068	416.405	322.870	202.841	97.680	43.950
Magnitude	2.107	1.811	1.557	1.309	1.106	1.097	1.082	512.705	476.929	416.149	330.296	195.391	98.337	39.905
Random	2.132	1.854	1.587	1.333	1.124	1.112	1.097	506.931	466.331	408.697	324.220	192.011	97.031	39.338
Variational Dropout	2.243	1.966	1.662	1.338	1.197	1.099	1.075	520.777	478.354	439.738	338.174	234.350	133.628	64.624
Vectorized Memory Access														
L0 Regularization	2.449	2.226	2.049	1.853	1.706	1.653	1.563	459.718	388.838	324.874	233.172	138.280	63.671	29.952
Magnitude	2.425	2.259	2.137	1.813	1.765	1.743	1.649	444.772	382.170	303.182	237.997	122.396	61.950	26.228
Random	2.471	2.249	2.161	1.789	1.757	1.741	1.690	436.370	383.939	299.201	241.279	122.922	61.981	25.541
Variational Dropout	2.520	2.295	2.111	1.841	1.779	1.716	1.632	462.752	409.335	345.994	247.550	160.004	84.559	41.316
Column Tiling														
L0 Regularization	2.479	1.970	1.584	1.137	0.762	0.541	0.454	454.320	442.138	422.168	378.375	293.772	183.602	97.673
Magnitude	2.386	1.934	1.534	1.117	0.703	0.500	0.466	452.417	446.289	421.829	385.976	307.040	215.597	92.616
Random	2.328	1.917	1.506	1.106	0.698	0.495	0.469	463.552	450.359	429.644	390.042	309.004	218.076	91.971
Variational Dropout	2.617	2.142	1.709	1.162	0.857	0.603	0.470	447.187	442.728	432.155	393.679	312.883	220.968	131.492
Block Tiling														
L0 Regularization	1.413	1.151	0.939	0.665	0.427	0.276	0.180	798.502	753.037	711.565	648.286	526.479	354.190	230.247
Magnitude	1.337	1.160	0.923	0.642	0.389	0.261	0.192	806.643	742.928	700.463	672.355	554.688	413.504	225.439
Random	1.295	1.125	0.925	0.637	0.383	0.259	0.199	833.234	766.914	699.364	676.981	562.280	416.331	216.419
Variational Dropout	1.467	1.266	1.039	0.694	0.493	0.325	0.202	796.614	743.397	706.857	658.160	545.772	405.291	297.240

Table 5.5. *Time in milliseconds and GFLOPs/s for all kernels at input sequence $N = 4096$. Underlined values are best for that sparsity.*

6 Future Work

This chapter outlines potential directions for further improving the performance and efficiency of Sparse Matrix-Matrix Multiplication (SpMM) kernels in Transformer-based architectures.

1. **Solve Work Imbalance with a Partition Step:** Current kernels suffer from uneven work distribution across threads, especially for irregular sparsity patterns. Introducing a partitioning step, as cuSPARSE seems to employ, to dynamically assign rows or blocks to threads could help achieve better load balancing and improve overall throughput.

2. **Switch from General Sparse Format to a Custom One:** General sparse formats are versatile but often suboptimal for attention matrices due to their disregard for data reusability. Designing a custom format that exploits the specific sparsity patterns of Transformer workloads could reduce memory overhead and accelerate computation.

3. **Row Swizzle Load Balancing:** Row swizzling can reorder rows to improve memory coalescing and reduce execution imbalance.

4. **Quantization:** Reducing the numerical precision of weights and activations can cut memory usage and increase computational throughput. Exploring quantized SpMM implementations while maintaining accuracy is a promising direction for future research.

5. **Fused Kernels:** [8] Fused kernels are the current state-of-the-art accelerator of attention computation, used in the highly optimized NVIDIA CUDA Deep Neural Network library (cuDNN) [18]. Instead of focusing the atomic optimization of individual Attention operations, they apply a macroscopic analysis across the entire pipeline, an approach that ultimately delivers the best results.

7 Conclusion

In this work, we investigated the performance bottlenecks inherent in GPU-based Sparse Matrix-Matrix Multiplication (SpMM) kernels, with a focus on applications in Transformer models and attention mechanisms. Through the implementation and study of multiple kernel variants, we were able to identify key factors affecting both computational throughput and memory efficiency, providing insights into why standard implementations sometimes fail to deliver expected speedups.

Subsequent analysis demonstrated that careful design of kernel execution strategies, including memory coalescing, work distribution that favors data reusability paired with efficient L1 cache use, can significantly improve performance. By benchmarking against NVIDIA's cuSPARSE library, we were able to quantify speedups and slowdowns under varying sparsities, highlighting the importance of adapting kernel behavior to input characteristics rather than relying on general solutions.

Additionally, the work highlighted the limitations of sparse attention mechanisms when confronted with irregular sparsity patterns and non-uniform workloads. These findings underscore the need for continued optimization, particularly in areas such as load balancing, custom sparse formats, and fusion of kernel operations, to fully exploit the computational potential of modern GPU architectures.

Overall, this thesis contributes a deeper understanding of the interplay between sparsity, memory access patterns, and GPU execution efficiency. The proposed analyses and experimental results provide a foundation for future research in high-performance sparse computation, paving the way for more scalable and efficient implementations of Transformer models and other large-scale neural networks.

A Proofs

Let $A \in \mathbb{R}^{m \times k}$ be a sparse matrix, and $B \in \mathbb{R}^{k \times n}$ be a dense matrix. For any two matrices X, Y:

$$(XY)^T = Y^T X^T.$$

therefore

$$(AB)^T = B^T A^T \implies AB = (B^T A^T)^T$$

where AB the original product. Let's consider the elementwise multiplications and additions. Let $C = AB \in \mathbb{R}^{m \times n}$, with elements

$$C_{ij} = \sum_{l=1}^{k} A_{il} Blj$$

Now consider $D = B^T A^T \in \mathbb{R}^{n \times m}$, with elements

$$D_{ji} = \sum_{l=1}^{k} B_{jl}^T A_{li}^T = \sum_{l=1}^{k} B_{lj} A_{il} = C_{ij}$$

Therefore, any kernel implementing *sparse* $\times$ *dense* can be mathematically replaced by a kernel implementing *dense*$\times$ *sparse* on the transposed operands without changing the result or total number of operations.

Transposing dense matrices incurs *zero* cost, as it only involves a reinterpretation of the memory layout rather than any arithmetic operations. In contrast, transposing sparse matrices carries a nontrivial computational overhead, as it requires rearranging the internal storage structures; however, this is considered a one-time preprocessing step that can be amortized across multiple subsequent multiplications.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser and Illia Polosukhin. *Attention is all you need. Advances in neural information processing systems*, 30, 2017.

[2] Rewon Child, Scott Gray, Alec Radford and Ilya Sutskever. *Generating long sequences with sparse transformers. arXiv preprint arXiv:1904.10509*, 2019.

[3] Aurko Roy, Mohammad Saffar, Ashish Vaswani and David Grangier. *Efficient content-based sparse attention with routing transformers. Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[4] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder and Donald Metzler. *Long range arena: A benchmark for efficient transformers. arXiv preprint arXiv:2011.04006*, 2020.

[5] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang and others. *Big bird: Transformers for longer sequences. Advances in neural information processing systems*, 33:17283–17297, 2020.

[6] Nikita Kitaev, Łukasz Kaiser and Anselm Levskaya. *Reformer: The efficient transformer. arXiv preprint arXiv:2001.04451*, 2020.

[7] Trevor Gale, Matei Zaharia, Cliff Young and Erich Elsen. *Sparse gpu kernels for deep learning. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra and Christopher Ré. *Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in neural information processing systems*, 35:16344–16359, 2022.

[9] NVIDIA Corporation, Santa Clara, CA. *CUDA C++ Programming Guide*, 13.0th edition, 2025.

[10] Ali Pinar and Michael T Heath. *Improving performance of sparse matrix-vector multiplication. Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 30–es, 1999.

[11] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat and others. *Gpt-4 technical report. arXiv preprint arXiv:2303.08774*, 2023.

[12] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser and Björn Ommer. *High-resolution image synthesis with latent diffusion models. Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[13] Wikipedia contributors. *Softmax function — Wikipedia, The Free Encyclopedia*, 2025. [Online; accessed 20-October-2025].

[14] Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou and Charith Mendis. *SPLAT: A framework for optimised GPU code-generation for SParse reguLar ATtention. Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1632–1660, 2025.

[15] Shigang Li, Kazuki Osawa and Torsten Hoefler. *Efficient quantized sparse matrix operations on tensor cores. SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[16] Rajeshwari Devaramani Justin Luitjens. *CUDA Pro Tip: Increase Performance with Vectorized Memory Access*, 2013.

[17] NVIDIA Corporation, Santa Clara, CA. *CUDA C++ Best Practices Guide*, 13.0th edition, 2025.

[18] NVIDIA Corporation, Santa Clara, CA. *NVIDIA CUDA Deep Neural Network library*, 2025.

List of Abbreviations

LLM	Large Language Model
MHSA	Multi-head Sparse Attention
SpMM	SParse matrix–dense Matrix Multiplication
SDDMM	Sampled Dense-Dense Matrix Multiplication
NLP	Natural Language Processing
FMA	Fused Multiply Add
COO	Coordinate List
CSR	Compressed Sparse Row
CSC	Compressed Sparse Column
GMEM	Global Memory
SMEM	Shared Memory
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor
GPU	Graphics Processing Unit
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Arrays
HBM	High Bandwidth Memory
GEMM	General Matrix Multiplication
GSF	General Sparse Formats
BiW	Best in Work