Politecnico di Torino

Master's Degree in Electronic Engineering





HW/SW Co-Design of a Fault-Tolerant Accelerator for Sparse Transformer Self-Attention Through Packed Token Pruning

Supervisors

Prof. Luciano Lavagno

Prof. Jing-Jia Liou

Candidate

Giacomo Gramaglia

Academic Year 2025/2026

1 Abstract

The transformer architecture has become dominant in fields such as natural language processing and computer vision, as its self-attention mechanism enables greater parallelism than convolutional and recurrent architectures. This allows significant speedups when run on parallel hardware, such as GPUs or tensor processors, and has driven a fast-paced growth in LLM sizes and context lengths. However, the high computational complexity of transformer self-attention, which grows quadratically with sequence length, combined with its high memory bandwidth requirements, poses a serious challenge to the architecture's scalability. Memory usage is particularly critical, as memory bandwidth has become the primary bottlneck in most AI workloads.

An additional challenge often overlooked is posed by hardware reliability. For instance, researchers at Google and Meta have reported occurrences of silent data corruption in HPC clusters. Detecting and correcting these faults at the hardware level could potentially save significant compute in long-running training jobs, and protect inference output from being corrupted.

To address the aforementioned issues, this work presents the HW/SW co-design of a fault-tolerant accelerator for transformer sparse self-attention through packed token pruning. This implementation reduces memory traffic by pruning (i.e. "detecting and removing") unimportant tokens, at the cost of small accuracy degradations. Moreover, the accelerator incorporates a lightweight hardware-based ABFT checksum scheme, which enhances its reliability during inference by protecting matrix multiplications.

Contents

1 Abstract **1**

2 Introduction **3**
2.1 The AI memory wall . . . 3
2.2 Fault-tolerance in large-scale AI systems . . . 3
2.3 Proposed solution . . . 4
2.4 Contributions . . . 5

3 Background **6**
3.1 Transformer self-attention . . . 6
3.1.1 Full self-attention . . . 6
3.1.2 Sparse self-attention . . . 8
3.1.3 Sparsity via token pruning . . . 9
3.1.4 Selection function . . . 13
3.2 Tiling operations, mapping to hardware . . . 14
3.2.1 Block matrix notation . . . 14
3.2.2 Tiled matrix multiplication (TMUL) . . . 15
3.2.3 Block-wise softmax . . . 16
3.3 ABFT for matrix multiplication . . . 17
3.3.1 Checksum matrix definitions . . . 17
3.3.2 Error detection in matrix multiplication . . . 18

4 Algorithm implementation **19**
4.1 Fused operations and FlashAttention . . . 19
4.2 Baseline algorithm . . . 19
4.3 Sparse algorithm . . . 20

5 Hardware implementation **24**
5.1 Systolic array . . . 24
5.1.1 Baseline systolic array . . . 25
5.1.2 Fault-tolerant systolic array . . . 26
5.2 Arithmetic and logic units . . . 29
5.3 Top-p candidate selection unit . . . 29
5.4 Hadware synthesis and optimization . . . 29
5.4.1 Sizing and datatypes . . . 29
5.4.2 Latency . . . 30
5.4.3 Synthesis reports . . . 30

6 Benchmarking **32**
6.1 Memory traffic estimation . . . 32
6.2 Accuracy evaluation . . . 35
6.3 Fault injection campaign . . . 35

7 Conclusions **36**

References **37**

8 Acknowledgements **40**

2 Introduction

In recent years, large language models (LLMs) have transitioned from research prototypes to technologies deployed at scale for the broader public, powering applications such as chat-bots, agentic code generators, and conversational assistants. This rapid adoption has been driven by breakthroughs in model architecture, such as the attention mechanism [1], as well as by continued advancements in underlying hardware. Historically, the growth in computing performance has followed a steady trend described by Moore's Law [2]. This progressive advancement has translated into higher processor performance, enabling the integration of increasingly more powerful logic. Neural network architectures have aggressively exploited this opportunity: recent studies report LLM parameter counts scaling at a rate of 410× every two years in the 2018-2022 period [3], growing from hundreds of millions [4] to hundreds of billions [5]. However, memory bandwidth and latency have not scaled at a similar rate. While compute throughput has benefitted from transistor scaling and the development of advanced hardware architectures, DRAM has improved at a much slower rate, and the energy cost of moving data has become dominant over the cost of computation itself.

2.1 The AI memory wall

Thus arises the memory wall problem, thoroughly analyzed in [3]. The paper shows how over the last 20 years hardware peak compute power has increased by $60,000\times$, against $100\times$ for DRAM bandwidth, and $30\times$ for interconnect bandwidth. On top of this, memory latency has been even harder to improve, and it is challenging to increase capacity without compromising latency itself. All these factors contributed in making memory the bottleneck for training and serving AI workloads, even in cases where the entire model fits on-chip.

2.2 Fault-tolerance in large-scale AI systems

In 2021, both Google [6] and Meta [7] independently reported the widespread occurrence of silent data corruption (SDC) across their production server fleets. While traditional fail-stop hardware faults are detected and reported by the system, SDCs cause a subset of processor cores to silently compute incorrect results without crashing or flagging an error. This class of faults is particularly insidious, as it can remain undetected for long periods of time while silently propagating through a computation before manifesting as corrupted output. In large-scale AI training and inference systems, this poses a serious risk: an SDC affecting a single matrix multiplication can silently corrupt model weights or activations without any visible failure, wasting large amounts of compute in long-running training jobs or degrading the correctness of inference results without detection. These findings motivate the need for fault tolerance to be addressed directly at the hardware level, rather than relying solely on software-level mitigations such as checkpointing or result validation, which are unable to detect or correct errors originating within the compute datapath itself.

2.3 Proposed solution

Multiple strategies have been explored in state-of-the-art (SOTA) research to mitigate the memory wall, spanning algorithmic, architectural, and memory-technology approaches, as summarized in Table 1.

Strategy	**Techniques**
Algorithmic	Sparsity, model compression, efficient mechanisms, KV-cache optimization
Architectural	Data reuse, locality-aware dataflows, application-specific architectures
Memory-technology	Compute-in-memory, near-memory computing

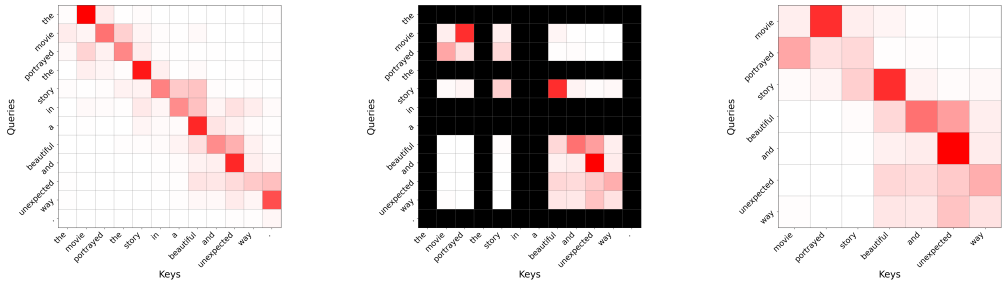
Table 1: Strategies and methods

Efficient mechanisms, such as operation fusion as proposed in FlashAttention [8], restructure the attention computation to avoid materializing the full attention matrix in memory, significantly reducing memory I/O and improving data reuse. Sparsity is an intriguing algorithmic approach, as it can reduce memory and compute requirements without requiring model re-training. However, it is challenging to implement efficiently, since traditional hardware (GPGPUs, vector processors) tends to perform poorly on sparse data, at times underperforming relative to dense computation. Additionally, sparsity must not compromise the accuracy of the results. HW/SW co-design has therefore emerged as an effective solution, allowing an efficient sparse attention algorithm and a special-purpose hardware architecture to support it to be designed in tandem.

The proposed solution is an HW/SW co-designed accelerator that sparsifies the attention computation through packed token pruning. By packing the surviving tokens after pruning, the resulting data remains dense, allowing the underlying functional units to operate at full utilization without requiring specialized sparse-data support. The accelerator's dataflow follows an adaptation of FlashAttention, and matrix multiplications are performed by a fault-tolerant systolic array equipped with a lightweight hardware-based ABFT checksum unit, providing online error detection for the computation.

Figure 1 illustrates the principle behind token pruning, first introduced in "SpAtten" [9]. Approximate attention scores are first estimated at low memory cost (a), and are used to determine important tokens to retain versus unimportant ones to prune (b). Once the surviving tokens are identified, full-precision attention is computed only for this reduced set (c) [1].

[1]Token pruning has an intuitive interpretation: tokens corresponding to words with strong semantic content (e.g., nouns, verbs, adjectives) tend to receive higher importance scores, while "filler" words (e.g., articles, conjunctions, prepositions) are more likely to be pruned.



(a) (b) (c)

Figure 1: (a) low-precision self-attention over all tokens (b) candidate selection and token pruning over low-precision attention scores (c) full-precision attention scores after token pruning

2.4 Contributions

This work offers the following contributions:

1. An adaptation of the FlashAttention algorithm [8] for optimal mapping to systolic array under output stationary (OS) dataflow;
2. A proof of equivalence between masked and packed token pruning;
3. The application of top-p (nucleus) candidate selection for token pruned sparse attention;
4. The hardware design of a fault-tolerant accelerator which combines all these techniques.

3 Background

3.1 Transformer self-attention

Self-attention, first introduced by the paper "Attention is All You Need"[1], is the core mechanism transformers use to compute correlations, or attention scores, between every pair of tokens in a sequence. Unlike RNNs and CNNs, which process tokens sequentially or with limited context windows, self-attention computes relationships between every token pair in parallel, regardless of their distance in the sequence [2]. The remainder of this subsection shows the calculations involved in a self-attention layer as defined by the referenced paper.

3.1.1 Full self-attention

Computing a self-attention layer involves these dimensions (table 2):

Dimension	**Symbol**	**Description**
Sequence length	L	Number of tokens in the sequence.
Model dimension	d_{model}	Size of the model embedding space.
Number of heads	h	Number of attention heads in multi-headed attention.
Head dimension	$d_{head} = d_{model}/h$	Hidden dimension of a single attention head.
FFN dimension	d_{FFN}	Hidden dimension of the feed-forward network.

Table 2: Transformer self-attention: dimensions

And the following inputs (table 3):

Operand	**Symbol**	**Description**
Input embedding matrix	$\mathbf{X} \in \mathbb{R}^{L \times d_{model}}$	Input of the attention layer, obtained by tokenizing and applying positional encoding to the input sequence.
Weight projection matrices	$\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{model} \times d_{model}}$	Learned matrices used to project the input embeddings and obtain $\mathbf{Q}$,$\mathbf{K}$,$\mathbf{V}$.
FFN projection matrices	$\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{FFN}}$ $\mathbf{W}_2 \in \mathbb{R}^{d_{FFN} \times d_{model}}$	Learned matrices used for FFN linear layers.
Bias vectors	$\overline{b_1} \in \mathbb{R}^{d_{FFN}}$ $\overline{b_2} \in \mathbb{R}^{d_{model}}$	Additive vectors used to apply positive/negative bias in the FFN.

Table 3: Transformer self-attention: operators

[2] Self-attention also has a concrete interpretation: through training, tokens that share strong semantic context (e.g. adjective + noun) tend to develop higher attention scores than semantically unrelated ones. Strongly correlated tokens thus "attend to each other".

Scaled Dot-Product Attention

Given Query, Key and Value matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{L \times d_{head}}$, scaled dot-product attention (figure 2a) is defined as:

$$\mathbf{O} = \text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_{head}}}\right) \times \mathbf{V} \in \mathbb{R}^{L \times d_{head}}$$

With softmax being a normalization function applied row-wise to the attention score matrix defined as:

$$\text{softmax}(\overline{x}) = \frac{\exp(\overline{x})}{\text{sum}(\exp(\overline{x}))} = \frac{[e^{x_1}, \ldots, e^{x_L}]}{\sum_{j=1}^{L} e^{x_j}} \in \mathbb{R}^L$$

Multi-Head Attention

Instead of computing attention once over $\{\mathbf{Q},\mathbf{K},\mathbf{V}\} \in \mathbb{R}^{L \times d_{model}}$ matrices, self-attention can be reformulated into multi-head attention (figure 2b) to break down the computation into h parallel heads, all of which operate on $\{\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h\} \in \mathbb{R}^{L \times d_{head}}$ matrices. These are obtained as follows:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q \qquad \mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K \qquad \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V \qquad i = 1, \ldots, h$$

Where $\mathbf{W}_i \in \mathbb{R}^{d_{model} \times d_{head}}$ is the i^{th} submatrix of the original $\mathbf{W} \in \mathbb{R}^{d_{model} \times d_{model}}$. Then, the attention outputs of all independent heads are merged back into one:

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\mathbf{O}_1, \ldots, \mathbf{O}_h)\mathbf{W}^O$$

Feed-Forward Network

Finally, the attention outputs are fed into a FFN (figure 2c), a Neural Network that comprises 2 linear layers with a non-linear layer (activation) in between:

$$\text{FFN} = \text{activation}(\mathbf{O}\mathbf{W}_1 + \overline{b}_1)\mathbf{W}_2 + \overline{b}_2 \in \mathbb{R}^{\text{L} \times d_{model}}$$

Where the activation function is usually a ReLU, GELU, or SiLU.

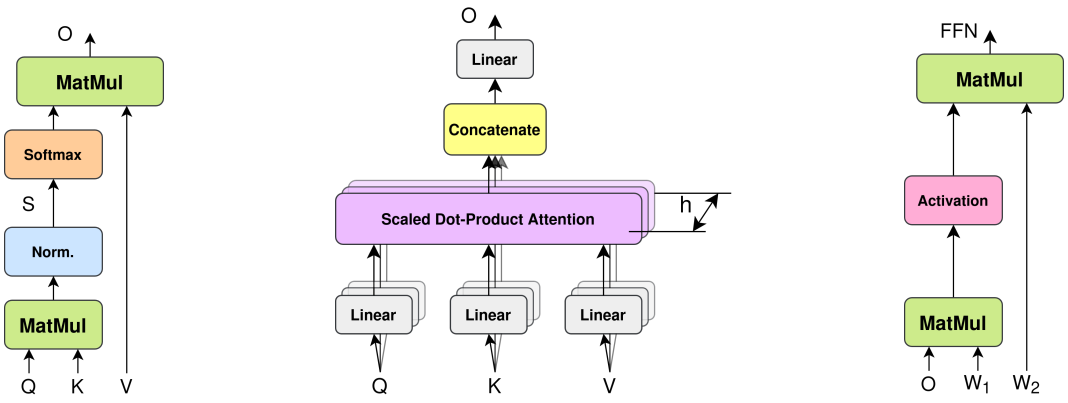


Figure 2: (a) Scaled dot-product attention (b) Multi-head attention (c) FFN

Computational Complexity

From the previous formulae, the multiplication $\mathbf{Q} \times \mathbf{K}^T$, which is the step concerning attention score calculation, has computational complexity $\text{O}(dL^2)$.

3.1.2 Sparse self-attention

To overcome the quadratic complexity of self-attention and the memory wall problem, many different sparse self-attention methods have been devised [10]. These methods take roots in the fact that LLMs are often over-parametrized, and many Query-Key (QK) pairs can be pruned away safely without compromising accuracy. Sparse methods can be classified into:

- **Static:** candidate selection is pre-determined and independent of input sequence;
- **Dynamic:** candidate selection is done during inference, and is dependent on the input sequence;
- **Hybrid:** static and dynamic methods are combined.

While static methods are the easiest to implement, their limited flexibility makes them more prone to missing important relationships, especially between far-away tokens. This results in accuracy losses, which can be significant. For this reason, dynamic sparse schemes are generally the preferred choice in recent literature. These typically restructure attention computation as follows:

1. Estimate approximate attention scores at low memory/compute cost;
2. Perform candidate selection to isolate important QK-pairs;
3. Recalculate full-precision self-attention for important QK-pairs only.

Some popular dynamic sparse attention accelerator co-designs are (table 4):

Accelerator	**Pattern**	**Sparsity level**	**Algorithm**
SpAtten [9]	Coarse-grained, structured	Low	Cascade token and head pruning
ELSA [11]	Fine-grained, unstructured	Medium	Approximation through Sign Random Projection (SRP)
A^3 [12]	Unstructured	Medium	Approximation
FACT [13]	Coarse-grained, structured	Medium	Eager correlation prediction
Sanger [14]	Fine-grained, structured	High	Attention pruning, matrix restructuring

Table 4: Accelerators for dynamic sparse attention

These solutions differ in their prediction schemes (low-cost approximation method, candidate selection function), hardware implementation, and optimization goals. Since comparing all of them goes well beyond the scope of this work, the reader is invited to refer to "A Survey on Sparsity Exploration in Transformer-Based Accelerators"[10] for a more in-depth analysis of different methods. The remaining part of this subsection will focus on sparsity via token pruning.

3.1.3 Sparsity via token pruning

The prediction scheme adopted introduces sparsity via token pruning, first introduced by "SpAtten" [9]. This strategy consists in first calculating approximate attention scores at low precision (i.e. low memory cost), performing candidate selection to isolate important tokens (unlike other fine-grained methods, which isolate individual QK-pairs), and re-calculate attention at full precision only considering these. Token pruning introduces then a coarse-grained, structured sparsity. This strategy has the advantages of being easy to implement, and being well suitable for packing. In turn, the overall sparsity that can be obtained from it is usually lower than other unstructured, fine-grained sparse methods. The operations performed by a token pruned sparse attention scheme are described as follows.

Masked token pruning

1. **QKV generation:**
   The full-precision QKV matrices are calculated as previously defined:

   $$\mathbf{Q} = \mathbf{X}\mathbf{W}^Q \qquad \mathbf{K} = \mathbf{X}\mathbf{W}^K \qquad \mathbf{V} = \mathbf{X}\mathbf{W}^V \qquad \{\mathbf{Q}, \mathbf{K}, \mathbf{V}\} \in \mathbb{R}^{L \times d_{model}}$$

   Then, Q and K are quantized to obtain their low-precision counterparts. In this work, full-precision QKV matrices are in 32-bit fixed point format, while low-precision Q and K are 8-bit fixed point:

   $$\widetilde{\mathbf{Q}} = \text{quantize}\,(\mathbf{Q}) \qquad \widetilde{\mathbf{K}} = \text{quantize}\,(\mathbf{K}) \qquad \{\widetilde{\mathbf{Q}}, \widetilde{\mathbf{K}}\} \in \mathbb{R}^{L \times d_{model}}$$

2. **Low-precision prediction scheme:**
   The first step of the prediction scheme is calculating approximate attention scores using the quantized QK matrices:

   $$\widetilde{\mathbf{S}} = \frac{\widetilde{\mathbf{Q}} \times \widetilde{\mathbf{K}}^T}{\sqrt{d_{model}}} \in \mathbb{R}^{L \times L}$$

   Then, the approximate attention score matrix is used to find the list of important tokens (and consequently, which tokens to be pruned). The result is a column importance vector [3]:

   $$\overline{c} = \text{softmax}\left(\frac{\text{colsum}(\widetilde{\mathbf{S}})}{L}\right) = \text{softmax}\left(\frac{1}{L}\sum_{i=1}^{\text{L}} \widetilde{\mathbf{S}}(i,:)\right) \in \mathbb{R}^L$$

   At this point, the selection function is used to obtain the list of indices of remained tokens:

   $$\overline{I} = \text{selection}(\overline{c}) \in \mathbb{N}^B, \quad 1 \leq B \leq L$$

   With B being the token budget.

[3] The reason why cumulative column importance scores are used to determine the set of globally important tokens is not arbitrary: a small column sum means that the key connected to that token does not contribute to other tokens by providing semantic context. Therefore, the token itself is not globally influential, and pruning it should not affect the attention output.

3. **Full-precision sparse attention:**
   Finally, attention scores are calculated at full-precision:

$$\mathbf{S}_{\text{full}} = \frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_{model}}}$$

   And masking is applied to retain selected tokens only:

$$\mathbf{S}_{\text{sparse}}(i,j) = \begin{cases} \mathbf{S}_{\text{full}}(i,j), & i,j \in \overline{I} \\ -\infty, & \text{elsewhere} \end{cases}$$

   And softmax is applied:

$$\mathbf{O}_{\text{sparse}} = \text{softmax}(\mathbf{S}_{\text{sparse}}) \times V \qquad \leftarrow \text{Note: } 0/0 \text{ exception resolved as } 0$$

   Since some rows of the sparse attention matrix will be filled with $-\infty$, applying softmax will yield $0/0$ exceptions for every element of the row. In such case, the exceptions are resolved with 0.

4. **Sparse feed-forward network:**
   As a last step, the sparse attention scores are fed to the FFN:

$$\text{FFN}_{\text{sparse}} = \text{activation}(\mathbf{O}_{\text{sparse}}\mathbf{W}_1 + \overline{b}_1)\mathbf{W}_2 + \overline{b}_2$$

A scheme of operations is shown in figure 3. Some remarks:

- For simplicity, these formulae were given in single-head attention formulation. As for regular self-attention (i.e. without sparsity), these can be reformulated in multi-head form;
- The previous formulation shows sparsity via masked token pruning. This formulation assumes the entire full attention matrix is computed. Since avoiding this expensive calculation is exactly the goal of this work, the next section will show how equivalent results can be obtained by packing this operation;
- An in-depth discussion on the selection function will follow.

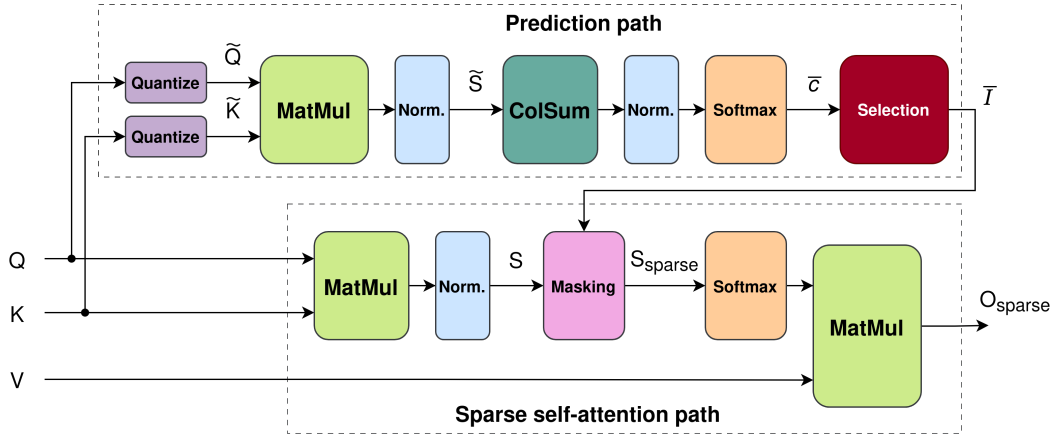


Figure 3: Operations - prediction and sparse self-attention schemes

Packed token pruning

The core idea behind packed token-pruned sparse attention is to:

1. Fetch full-precision QKV vectors only for tokens returned by candidate selection;
2. Pack the vectors in dense matrices;
3. Perform the full-precision attention and FFN computation in a dense manner on the packed matrices;

This subsection provides an algebraic proof to validate this method is interchangeable to masked token pruning, and no information is lost in the process.

Algebraic proof:

From the previous subsection, the sparse attention score matrix is defined through masking as follows (figure 4a):

$$\mathbf{S}_{\text{full}} = \frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d}} \in \mathbb{R}^{L \times L}$$

$$\mathbf{S}_{\text{sparse}}(i,j) = \begin{cases} \mathbf{S}_{\text{full}}(i,j), & i,j \in \overline{I} \\ -\infty, & \text{elsewhere} \end{cases}$$

A selection matrix $\mathbf{P} \in \mathbb{R}^{B \times L}$ is defined to perform row and column gathering:

$$\mathbf{P}(i,j) = \begin{cases} 1, & j = I_i \\ 0, & \text{elsewhere} \end{cases}$$

Where $\overline{I}$ is the vector containing the indices of remained tokens. When $\mathbf{P}$ is applied to the sparse attention score matrix, only non-pruned rows and columns are kept. This results in the packed attention score matrix $\mathbf{S}_p$:

$$\mathbf{S}_p = \mathbf{P}\mathbf{S}_{\text{sparse}}\mathbf{P}^T = \mathbf{P}\mathbf{S}_{\text{full}}\mathbf{P}^T \in \mathbb{R}^{B \times B}$$

The same selection matrix $\mathbf{P}$ is used to perform row gathering on $\{\mathbf{Q}, \mathbf{K}, \mathbf{V}\} \in \mathbb{R}^{L \times d}$ matrices, to obtain the packed matrices $\{\mathbf{Q}_p, \mathbf{K}_p, \mathbf{V}_p\} \in \mathbb{R}^{B \times d}$:

$$\mathbf{Q}_p = \mathbf{P}\mathbf{Q} \qquad \mathbf{K}_p = \mathbf{P}\mathbf{K} \qquad \mathbf{V}_p = \mathbf{P}\mathbf{V}$$

Calculating self-attention with the packed matrices (as shown in figure 4b) yields:

$$\frac{\mathbf{Q}_p \times \mathbf{K}_p^T}{\sqrt{d}} = \frac{(\mathbf{P}\mathbf{Q}) \times (\mathbf{P}\mathbf{K})^T}{\sqrt{d}} = \mathbf{P}\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\mathbf{P}^T = \mathbf{P}\mathbf{S}_{\text{full}}\mathbf{P}^T = \mathbf{S}_p$$

This result proves that the packed attention score matrix can be found by performing row gathering on full Q and K matrices, therefore only selecting necessary query and key vectors. Now we prove how softmax can be applied to the packed matrix. First, softmax is defined as:

$$\text{softmax}(\mathbf{S})_{ij} = \frac{\mathbf{S}(i,j)}{\sum_k e^{\mathbf{S}(i,k)}}$$

Since $e^{-\infty} = 0$, the denominator of the i^{th} row, with $i \in \overline{I}$ is:

$$\sum_{k \in \overline{I}} e^{\mathbf{S}_{full}(i,k)}$$

While for $i \notin \overline{I}$, the whole row will be zeroed out after softmax (0/0 exception resolved with 0). Therefore:

$$\mathbf{A}_{\text{sparse}}(i,j) = \text{softmax}(\mathbf{S}_{\text{sparse}})_{ij} = \begin{cases} \frac{e^{\mathbf{S}_{full}(i,j)}}{\sum_{k\in\overline{I}} e^{\mathbf{S}_{full}(i,k)}}, & i,j \in \overline{I} \\ 0, & \text{elsewhere} \end{cases}$$

Applying softmax to the packed attention score matrix yields the packed attention probability matrix:

$$\mathbf{A}_p(i,j) = \text{softmax}(\mathbf{S}_p)_{ij} = \frac{e^{\mathbf{S}_{full}(I_i,I_j)}}{\sum_{k\in\overline{I}} e^{\mathbf{S}_{full}(I_i,k)}}$$

Finally, to unpack this matrix, the selection matrix and its transpose are applied in reverse order:

$$\mathbf{P}^T\mathbf{A}_p\mathbf{P} = \mathbf{A}_{\text{sparse}} = \left(\begin{cases} \frac{e^{\mathbf{S}_{full}(i,j)}}{\sum_{k\in\overline{I}} e^{\mathbf{S}_{full}(i,k)}}, & i,j \in \overline{I} \\ 0, & \text{elsewhere} \end{cases} \right)$$

Having established how $\mathbf{A}_p$ can be unpacked to obtain $\mathbf{A}_{\text{sparse}}$, now we prove how $\mathbf{A}_p$ can be used to pack the subsequent linear projection. Since proof would be the same for attention output projection and FFN, only the former is shown. Starting from identity:

$$\mathbf{A}_{\text{sparse}} = \mathbf{P}^T\mathbf{A}_p\mathbf{P}$$

Then performing the projection by V:

$$\mathbf{O}_{\text{sparse}} = \mathbf{A}_{\text{sparse}}\mathbf{V} = (\mathbf{P}^T\mathbf{A}_p\mathbf{P})\mathbf{V} = \mathbf{P}^T\mathbf{A}_p(\mathbf{P}\mathbf{V}) = \mathbf{P}^T(\mathbf{A}_p\mathbf{V}_p) = \mathbf{P}^T\mathbf{O}_p$$

Which proves the sparse attention output matrix obtained thorugh masked token pruning equals the one computed densely through packing and unpacking.

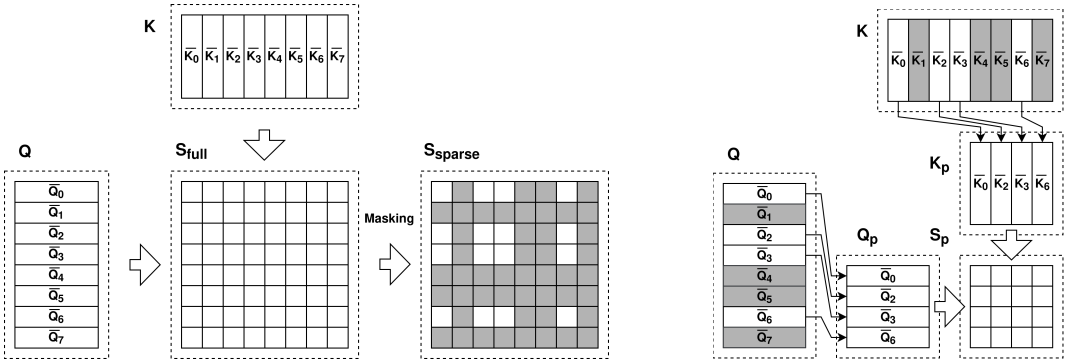


Figure 4: (a) Sparse attention scores through masking (b) Packed attention scores calculation

3.1.4 Selection function

Given a vector of attention scores, the selection function is the operator used by the prediction scheme to determine which tokens must be pruned and which must remain. This subsection shows the findings from "Twilight: Adaptive Attention Sparsity with Hierarchical Top-p Pruning" [15]. While the paper showcases an adaptive candidate selection method in the context of KV cache optimization, the "Twilight" framework can be extended to any sparse attention method and/or co-design modularly, impoving their efficiency and accuracy at low cost.

top-k selection

In many sparse attention methods, including "SpAtten" [9], candidate selection is preformed through top-k sampling. Given a vector of attention scores $\overline{s} \in \mathbb{R}^L$, a fixed budget B, the top-k operator is defined as:

$$\overline{I} = \text{top-}k(\overline{s}, B) = \arg\left(\max_{\overline{I}}\left(\sum_{i\in\overline{I}} \overline{s}[i]\right)\right) \quad s.t. \quad |\overline{I}| = B$$

In the previous definition, the returned $\overline{I}$ is a B-dimensional list of indices which point to the B largest elements of $\overline{s}$. The interpretation is simple: given the fixed budget, top-k optimizes candidate selection by choosing the B largest elements of the set.

Focused vs. diffuse attention

The paper shows how top-k selection is fundamentally flawed because of two reasons. Quoting from [15]:

(a) "The best budget choices vary dynamically at runtime";
(b) "Existing algorithms need different degrees of over-selection to compensate the estimation inaccuracy".

To demonstrate this, the research introduces the concepts of:

- **Focused attention:** (figure 5a) occurs when the attention distribution focuses on very few tokens, rendering the other scores negligible;
- **Diffuse attention:** (figure 5b) occurs when attention is more evenly distributed among all tokens, making them all equally important for the final score.

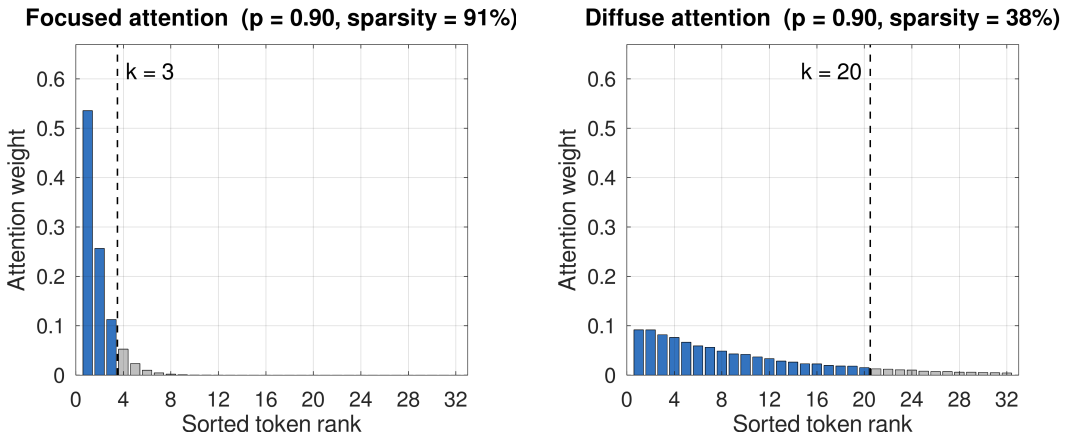
From figure 5, it is easy to see how fixed-budget candidate selection may result in under-pruning in case of focused attention (wasting resources), and over-puning in case of diffuse attention (leading to accuracy loss).

top-p selection

To come around this issue, the paper [15] proposes adaptive-budget candidate selection thorugh top-p sampling (a.k.a. nucleus sampling). The top-p operator is defined as:

$$\overline{I} = \text{top-}p(\overline{s}, p) = \arg\left(\min_{\overline{I}}\left(\overline{I}\right)\right) \quad s.t. \quad \sum_{i\in\overline{I}} \overline{s}[i] \geq p$$

Form the previous definition, the returned $\overline{I}$ is the smallest list of indices such that the sum of indexed elements goes above the theshold set by the p budget. This last one is thus a cumulative probability threshold that must always be



(a) Focused attention (b) Diffuse attention

Figure 5: Focused and diffuse attention examples

met regardless of the attention weight distribution, ultimately leading to more aggressive pruning in case of focused attention, and more consevative pruning in case of diffuse attention (see figure 5). As a final remark: while top-k can be trivially applied to unnormalized attention scores, top-p has to be applied to a probability distribution and thus always requires softmax normalization.

3.2 Tiling operations, mapping to hardware

The previous subsection showed the operations involved in full and sparse self-attention. Since d_{model} and d_{head} can vary between models, and L changes for every batch, hardware accelerators must be able to implement these operations for arbitrarily large matrices. The only way to ensure this requirement is met is by tiling operations: whenever input operands are too large for one-to-one mapping to hardware, the operation must be broken down into multiple passes, each operating on "tiles" (i.e. submatrices) of the input operands. The following subsections follow the notation described in [16].

3.2.1 Block matrix notation

Given a matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$, and $\mathbf{B} \in \mathbb{R}^{p \times m}$, and given the maximum tile size $\mathbb{R}^{t_r \times t_c}$, which corresponds to the maximum input operand size that can be directly mapped to hardware, the block matrix representation of A,B is:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,T_S} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{T_A,1} & \cdots & \mathbf{A}_{T_A,T_S} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \cdots & \mathbf{B}_{1,T_B} \\ \vdots & \ddots & \vdots \\ \mathbf{B}_{T_S,1} & \cdots & \mathbf{B}_{T_S,T_B} \end{bmatrix}$$

Where the original matrices are now partitioned into sub-matrices (tiles). The number of tiles each matrix is divided into is calculated as follows:

$$T_A = \lceil n/t_r \rceil \qquad T_S = \lceil p/t_c \rceil \qquad T_B = \lceil m/t_r \rceil$$

These are:

- T_A: number of tiles along row-dimension of $\mathbf{A}$;
- T_B: number of tiles along column-dimension of $\mathbf{B}$;
- T_S: (shared) tiles along column/row dimension of $\mathbf{A}/\mathbf{B}$, respectively;

3.2.2 Tiled matrix multiplication (TMUL)

The first operation to be tiled is matrix multiplication. This is the core of self-attention: from QKV generation, to the computation of attentions scores, to the FFN. The output matrix C can be represented in block notation:

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{bmatrix} \mathbf{C}_{1,1} & \cdots & \mathbf{C}_{1,T_B} \\ \vdots & \ddots & \vdots \\ \mathbf{C}_{T_A,1} & \cdots & \mathbf{C}_{T_A,T_B} \end{bmatrix}$$

Where each tile of $\mathbf{C}$ can be found as follows (see figure 6):

$$\mathbf{C}_{\alpha,\beta} = \sum_{\gamma=1}^{T_S} \mathbf{A}_{\alpha,\gamma} \times \mathbf{B}_{\gamma,\beta} \qquad \alpha = 1 : T_A, \quad \beta = 1 : T_B$$

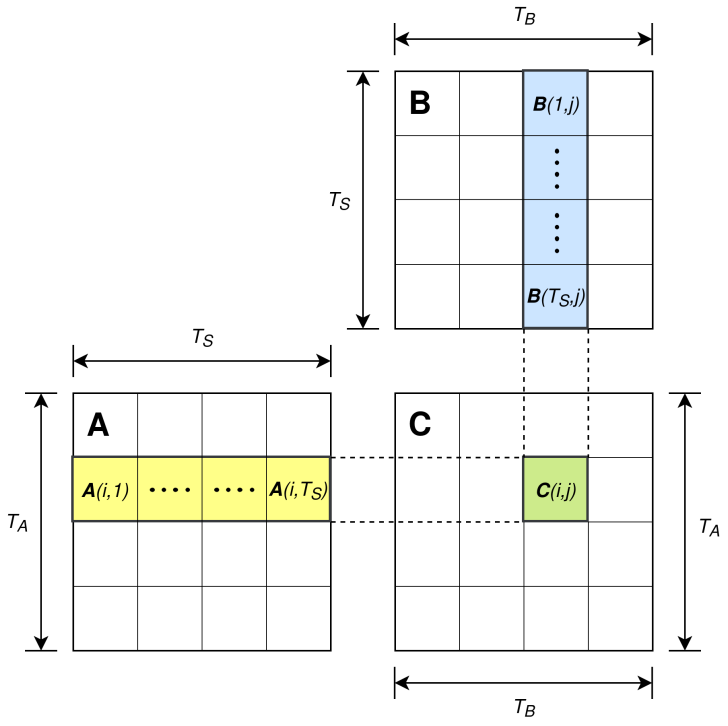


Figure 6: computation of a tile of matrix C through TMUL

TMUL algorithms allow obtaining the result of a large GEneral Matrix Multiply (GEMM) by breaking down the computation into multiple (smaller) matrix multiplications between tiles of $\mathbf{A}$ and $\mathbf{B}$. The main advantages are:

1. The memory access patterns of TMUL algorithms usually achieve higher cache hit rates than large untiled implementations;
2. Tiling allows mapping any matrix multiplication to a systolic array (more on this topic in the next sections), regardless of the input sizes.

3.2.3 Block-wise softmax

Softmax normalization requires a full row of the attention score matrix to compute its denominator, which conflicts with a tiled computation strategy. The paper "FlashAttention"[8] addresses this by reformulating softmax so that it can be computed incrementally over blocks, as shown below. Having a vector $\overline{x} \in \mathbb{R}^B$, softmax normalization is reformulated in its "safe" form:

$$m(\overline{x}) := \max_i \overline{x}_i \qquad f(\overline{x}) := [e^{x_1 - m(\overline{x})}, \ldots, e^{x_B - m(\overline{x})}] \qquad l(\overline{x}) := \sum_i f(\overline{x})_i$$

$$\text{softmax}(\overline{x}) := \frac{f(\overline{x})}{l(\overline{x})}$$

Which prevents overflow through the subtraction of $m(\overline{x})$.
We can apply this definition to the concatenation of two subvectors $\overline{x} = [\overline{x}^{(1)}\overline{x}^{(2)}] \in \mathbb{R}^{2B}$:

$$m(\overline{x}) = m([\overline{x}^{(1)}\overline{x}^{(2)}]) = \max(m(\overline{x}^{(1)}), m(\overline{x}^{(2)}))$$

$$f(\overline{x}) = \left[e^{m(\overline{x}^{(1)}) - m(\overline{x})} f(\overline{x}^{(1)}) \quad e^{m(\overline{x}^{(2)}) - m(\overline{x})} f(\overline{x}^{(2)})\right]$$

$$l(\overline{x}) = l([\overline{x}^{(1)}\overline{x}^{(2)}]) = e^{m(\overline{x}^{(1)}) - m(\overline{x})} l(\overline{x}^{(1)}) + e^{m(\overline{x}^{(2)}) - m(\overline{x})} l(\overline{x}^{(2)})$$

Since softmax is fully determined by $m(\overline{x})$, $l(\overline{x})$, and $f(\overline{x})$, and these quantities can be computed incrementally over subvectors, softmax admits a block-wise (tiled) computation. This forms the mathematical basis for the "FlashAttention" algorithm which is discussed in the next section.

3.3 ABFT for matrix multiplication

Algorithm-Based Fault Tolerance (ABFT) was first introduced in the paper [17] as a technique to protect matrix computations at the word level. It is very effective in providing a framework for error detection and/or correction, implementable both in hardware and software at low overhead, especially compared to other techniques such as Hamming codes, dual/triple modular redundancy (DMR/TMR). In principle, it provides fault tolerance through three phases:

1. Encoding data at the word level on the input matrices;
2. Performing the matrix computation;
3. Comparing the result with encoded data to detect/correct potential faults.

The mathematical basis for ABFT checksum schemes, as presented in [17], is introduced below.

3.3.1 Checksum matrix definitions

Column Checksum Matrix

Given matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, the *column checksum matrix* $\mathbf{A}_c \in \mathbb{R}^{(n+1) \times m}$ consists of the matrix A in the first n rows, and a *column summation vector* in the $(n+1)^{st}$ row. The elements of the summation vector are obtained as follows:

$$a_{n+1,j} = \sum_{i=1}^{n} a_{i,j} \qquad j = 1, \ldots, m$$

Using compact notation, we first define the ones vector:

$$\overline{e}^T = [1, \ldots, 1]$$

Then the column checksum matrix can be written as:

$$\mathbf{A}_c = \left[\frac{\mathbf{A}}{\overline{e}^T \mathbf{A}} \right]$$

Row Checksum Matrix

Given matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$, the *row checksum matrix* $\mathbf{A}_r \in \mathbb{R}^{n \times (m+1)}$ consists of the matrix A in the first m columns, and a *row summation vector* in the $(m+1)^{st}$ column. The elements of the summation vector are obtained as follows:

$$a_{i,m+1} = \sum_{j=1}^{m} a_{i,j} \qquad i = 1, \ldots, n$$

Using compact notation, the row checksum matrix can be written as:

$$\mathbf{A}_r = \left[\begin{array}{c|c} \mathbf{A} & \mathbf{A}\overline{e} \end{array} \right]$$

Full Checksum Matrix

The multiplication between a row checksum matrix and column checksum matrix generates a *full checksum matrix* (see figure 7):

$$\mathbf{A}_c \times \mathbf{B}_r = \left[\frac{\mathbf{A}}{\overline{e}^T \mathbf{A}} \right] \times \left[\begin{array}{c|c} \mathbf{B} & \mathbf{B}\overline{e} \end{array} \right] = \left[\begin{array}{c|c} \mathbf{AB} & \mathbf{AB}\overline{e} \\ \hline \overline{e}^T \mathbf{AB} & \overline{e}^T \mathbf{AB}\overline{e} \end{array} \right] = \mathbf{C}_f$$

Definition: each row or column in the full checksum matrix is called *checksum encoded vector*, or *CSEV*.
Some important corollaries and theorems follow:

- **Corollary 1.** The result of the addition of two CSEV's is a CSEV;
- **Corollary 2.** The product of a scalar and a CSEV is a CSEV;
- **Theorem 1.** The product of a scalar and a full checksum matrix is a full checksum matrix;
- **Theorem 2.** The transpose of a full checksum matrix is a full checksum matrix.

Based on these, matrix addition, multiplication, scalar product, LU decomposition, and transpose preserve the checksum property.

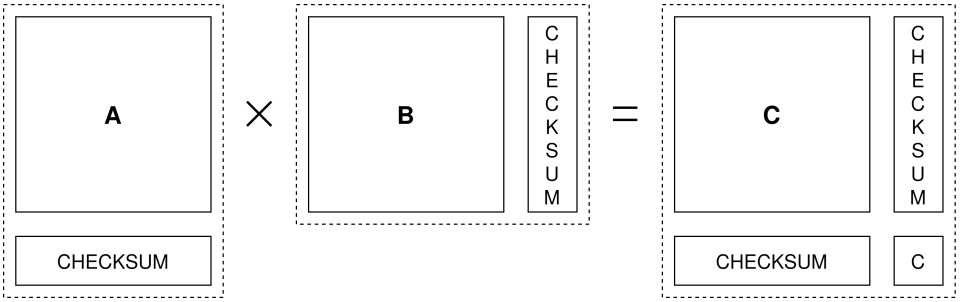


Figure 7: computation of a full checksum matrix

3.3.2 Error detection in matrix multiplication

Given matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$ and a row checksum matrix $\mathbf{B}_r \in \mathbb{R}^{p \times (m+1)}$, then their product yields row checksum matrix $\mathbf{C}_r \in \mathbb{R}^{n \times (m+1)}$:

$$\mathbf{A} \times \mathbf{B}_r = \mathbf{A} \times \left[\begin{array}{c|c} \mathbf{B} & \mathbf{B}\overline{e} \end{array} \right] = \left[\begin{array}{c|c} \mathbf{AB} & \mathbf{AB}\overline{e} \end{array} \right] = \left[\begin{array}{c|c} \mathbf{C} & \mathbf{C}\overline{e} \end{array} \right] = \mathbf{C}_r$$

Similarly, given a vector of rescale factors $\overline{f} \in \mathbb{R}^n$, the row-wise rescale operation can be performed on $\mathbf{C}_r$ as follows:

$$\mathrm{diag}(\overline{f}) \times \mathbf{C}_r = \mathbf{D} \times \left[\begin{array}{c|c} \mathbf{C} & \mathbf{C}\overline{e} \end{array} \right] = \left[\begin{array}{c|c} \mathbf{DC} & \mathbf{DC}\overline{e} \end{array} \right]$$

Hence, these two operations (matrix multiplication and row-wise rescale) both retain the checksum property.
If only error detection is to be implemented through ABFT, either row or column matrix encoding is sufficient (no full checksum matrix needed). In this case, the procedure consists of:

1. Encoding the row summation vector to generate a row checksum matrix;
2. Performing the matrix operation;
3. Computing the sum of all information elements in each row, and comparing with corresponding row checksum.

Any inconsistency from step 3 indicates a fault must have occurred in that row. While this simplified scheme does not allow for error location and correction, once an error is detected the whole computation can be repeated.

4 Algorithm implementation

This section covers in detail the adopted algorithms. It provides a logical connection between the implementation-independent background, and the hardware implementation. Two schemes were implemented in order to analyze the effect of token pruning in terms of memory traffic and accuracy, these are:

1. **Baseline**: FlashAttention [8] style full self-attention;
2. **Sparse**: FlashAttention [8] style self-attention with packed token pruning [9] and top-p [15] candidate selection;

Some preliminary remarks on the algorithms shown in this section:

- The baseline algorithm operates on data in 32-bit Fixed-Point/Floating-Point formats;
- The sparse algorithm operates on data in 8-bit Fixed-Point format (in the prediction scheme), and 32-bit Fixed-Point/Floating-Point formats (in the full-precision self-attention scheme);
- Since the following algorithms will be run on custom parallel hardware, the pseudocode are shown at tile/vector-level granularity, i.e. showing vector and tile operations instead of element-wise operations.
- The algorithms shown in this section adopt this color code:

Blue	Memory operations
Red	Matrix multiplication, row-wise tile rescale
Green	Vector operations
Yellow	Top-p (nucleus) candidate selection

4.1 Fused operations and FlashAttention

FlashAttention [8] introduces a novel algorithm that allows to tile the softmax operation, which is particularly critical in TMUL-based systolic array schemes because of its denominator depending on entire rows of the attention score matrix. This ultimately allows to fuse attention operations together, reducing the re-writes and re-reads to/from memory. The algorithm, adapted for optimal mapping to the designed hardware, is described in the following subsubsection.

4.2 Baseline algorithm

The following pseudocodes have been used to calculate an attention layer through systolic array-based TMUL and fused softmax.

Attention scores

Given query and key matrices $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{L \times d}$, the attention score matrix $\mathbf{S} \in \mathbb{R}^{L \times L}$ is found applying algorithm 1.

Fused softmax, attention output

After having obtained $\mathbf{S}$, the attention output is calculated by fusing together the softmax calculation and the projection by matrix $\mathbf{V} \in \mathbb{R}^{L \times d}$, as proposed in [8]. Algorithm 2 shows how this is performed.

Algorithm 1 Baseline: tiled attention score computation

```
1  TA = TB = ceil(n/t)
2  TS = ceil(d/t)
3  S = zeros(n,n)
4  for a = 1 : TA
5          for b = 1 : TB
6                  for s = 1 : TS
7                          load tile Q(a,s) to on-chip buffer
8                          load tile K(s,b) to on-chip buffer
9                          S(a,b) += Q(a,s) × K^T(s,b) / sqrt(d)
10                 end
11                 write S(a,b) to memory
12         end
13 end
```

This second algorithm follows the same dataflow and access patterns as the previous one, with the addition of component-wise vector operations to obtain scaling factors, resulting in a very memory efficient self-attention algorithm.

4.3 Sparse algorithm

Low-precision attention scores

The attention scores are calculated at low-precision following algorithm 3. This is equivalent to algorithm 1, with the only difference being it operates on low-precision 8-bit data.

Top-p candidate Selection

Algorithm 4 shows how candidate selection is performed. In principle, the low precision attention scores are read to accumulate column importances (line 5:7), to then apply softmax and top-p selection.
An important note: in order to reduce the complexity of the algorithm, the top-p operator is applied tile-wise, instead of globally. While the final distribution will slightly diverge from the ideal global top-p selection, this simplification makes for a more efficient algorithm.
After each inner loop, the local list of remained indices is appended to the global list and kept on-chip.

Token-pruned packed attention scores

Then, the list of remained indices is used to fetch packed $\mathbf{Q}_P$,$\mathbf{K}_P$ matrices, and $\mathbf{S}_P$ is calculated as in algorithm 5.

Token-pruned fused softmax and attention output

Finally, packed attention scores are used along with $\mathbf{V}_P$ to calculate packed attention outputs, as in algorithm 6.

Algorithm 2 Baseline: fused softmax and attention output

```
TA = TS = ceil(n/t)
TB = ceil(d/t)
O = zeros(n,d)
for a = 1 : TA
    for b = 1 : TB
        m_old = -inf(t)
        l_old = zeros(t)
        for s = 1 : TS
            load tile S(a,s) to on-chip buffer
            load tile V(s,b) to on-chip buffer
            # extract local {m,l}, exponentiate
            m_loc = rowmax(S(a,s))
            E(a,s) = exp(S(a,s) - m_loc)
            l_loc = rowsum(E(a,s))
            # scaling factors
            m_new = max(m_old, m_loc)
            alpha = exp(m_old-m_new)
            beta = exp(m_loc-m_new)
            l_new = l_old × alpha + l_loc × beta
            scale_old = alpha × l_old / l_new
            scale_new = beta / l_new
            # matrix operations
            O(a,b) = O(a,b) × scale_old
            local_prod = E(a,s) × V(s,b) × scale_new
            O(a,b) = O(a,b) + local_prod
            # update old stats
            m_old = m_new
            l_old = l_new
        end
    write O(a,b) to memory
    end
end
```

Algorithm 3 Sparse: tiled low-precision attention score computation

```
TA = TB = ceil(n/t)
TS = ceil(d/t)
S̃ = zeros(n,n)
for a = 1 : TA
        for b = 1 : TB
                for s = 1 : TS
                        load tile Q̃(a,s) to on-chip buffer
                        load tile K̃(s,b) to on-chip buffer
                        S̃(a,b) += Q̃(a,s) × K̃^T(s,b) / sqrt(d)
                end
                write S̃(a,b) to memory
        end
end
```

Algorithm 4 Sparse: top-p candidate selection over approximate S

```
TA = TB = ceil(n/t)
Ī = {}
for x = 1 : TA
        c̄ = zeros(t)
        for y = 1 : TB
                load tile S̃(y,x) to on-chip buffer
                c̄ += colsum(S̃(y,x)) / n
        end
        s̄ = softmax(c̄)
        Ī = concat(Ī, top-p(s̄, p))
end
```

Algorithm 5 Sparse: tiled attention score computation

```
TA = TB = ceil(B/t)
TS = ceil(d/t)
S_P = zeros(B,B)
for a = 1 : TA
        for b = 1 : TB
                for s = 1 : TS
                        load tile Q_P(a,s) to on-chip buffer
                        load tile K_P^T(s,b) to on-chip buffer
                        S_P(a,b) += Q_P(a,s) × K_P^T(s,b) / sqrt(d)
                end
                write S_P(a,b) to memory
        end
end
```

Algorithm 6 Sparse: token pruned fused softmax and attention output

```
TA = TS = ceil(B/t)
TB = ceil(d/t)
O_P = zeros(B,d)
for a = 1 : TA
    for b = 1 : TB
        m_old = -inf(t)
        l_old = zeros(t)
        for s = 1 : TS
            load tile S_P(a,s) to on-chip buffer
            load tile V_P(s,b) to on-chip buffer
            # extract local {m,l}, exponentiate
            m_loc = rowmax(S_P(a,s))
            E_P(a,s) = exp(S_P(a,s) - m_loc)
            l_loc = rowsum(E_P(a,s))
            # scaling factors
            m_new = max(m_old, m_loc)
            alpha = exp(m_old-m_new)
            beta = exp(m_loc-m_new)
            l_new = l_old × alpha + l_loc × beta
            scale_old = alpha × l_old / l_new
            scale_new = beta / l_new
            # matrix operations
            O_P(a,b) = O_P(a,b) × scale_old
            local_prod = E_P(a,s) × V_P(s,b) × scale_new
            O_P(a,b) = O_P(a,b) + local_prod
            # update old stats
            m_old = m_new
            l_old = l_new
        end
    write O_P(a,b) to memory
    end
end
```

5 Hardware implementation

This section focuses on the hardware side of the co-design. The hadware model of the accelerator was written in SystemC, and its verification, synthesis and optimization have been performed using the Cadence Stratus HLS EDA tool. Three top-level functional units have been designed in order to implement all necessary operations required by the custom self-attention scheme. These are:

Module	Operation
Systolic Array	Matrix multiplication, row-wise tile rescale
Vector (SIMD) ALU	Vector operations (full instruction set follows)
Top-p unit	Top-p (nucleus) candidate selection

Table 5: Designed hardware functional units

The color code follows the one from the previous section: each highlighted line in the algorithms means the operation is mapped to the functional unit from table 5 with corresponding color.

The next subsections will focus on the RTL/architectural implementation of each of these modules, with the last one discussing datatypes, latency, and synthesis reports.

5.1 Systolic array

The systolic array architecture was first introduced by [18]. This hardware architecture allows to implement computationally intensive operations such as convolution or tiled matrix multiplication through the interconnection of a 1D/2D array of Processing Elements (PEs), which pass new input operands to adjacent ones every clock cycle[4] and perform multiply-accumulate (MAC) operations. The absence of crossbars and/or global buses delivering operands to each individual PE in favor of systolic interconnect makes for a very cost-effective design with good scalability, and it is the reason why this architecture is very popular in Tensor Processing Unit (TPU) designs. Two systolic array design versions have been implemented in this work, see table 6:

Version	ABFT checksum scheme
Baseline	No
Fault-tolerant	Yes

Table 6: Systolic array design versions

Both versions follow the Output Stationary (OS) dataflow: while the operands A,B are shifted into the array, the partial result C is stationary inside the PE. The description of SA operation and the corresponding hardware implementations follow.

[4]This dataflow resembles the way blood flows through the cardiovascular system, hence the name "systolic".

5.1.1 Baseline systolic array

The computation of a matrix multiplication or rescale is obtained by alternating phases, which are described in table 7.

SA phase	Figure	Description
MAC	8	$\overline{A}$ and $\overline{B}$ row/column vectors are fed into the systolic array with an input skew, and every CC the PEs register them, perform MAC, and pass the previous A,B to the neighboring PEs.
Rescale	9a	Row-wise rescale factors are fed to the systolic array, and each PE rescales its accumulated partial result by them.
Shift out	9b	After the final result is ready at each PE's C_{out} port, the PEs propagate results from their previous row to the next, until the entire output tile is buffered.

Table 7: Systolic array phases

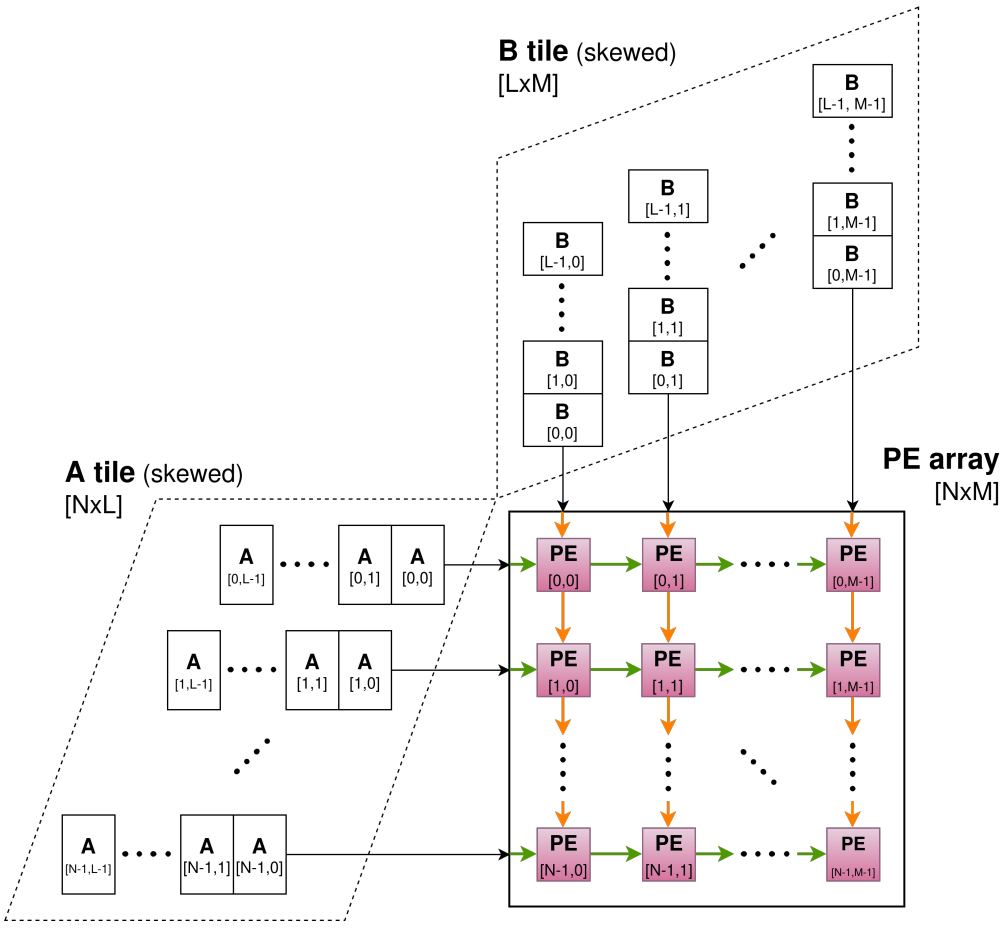
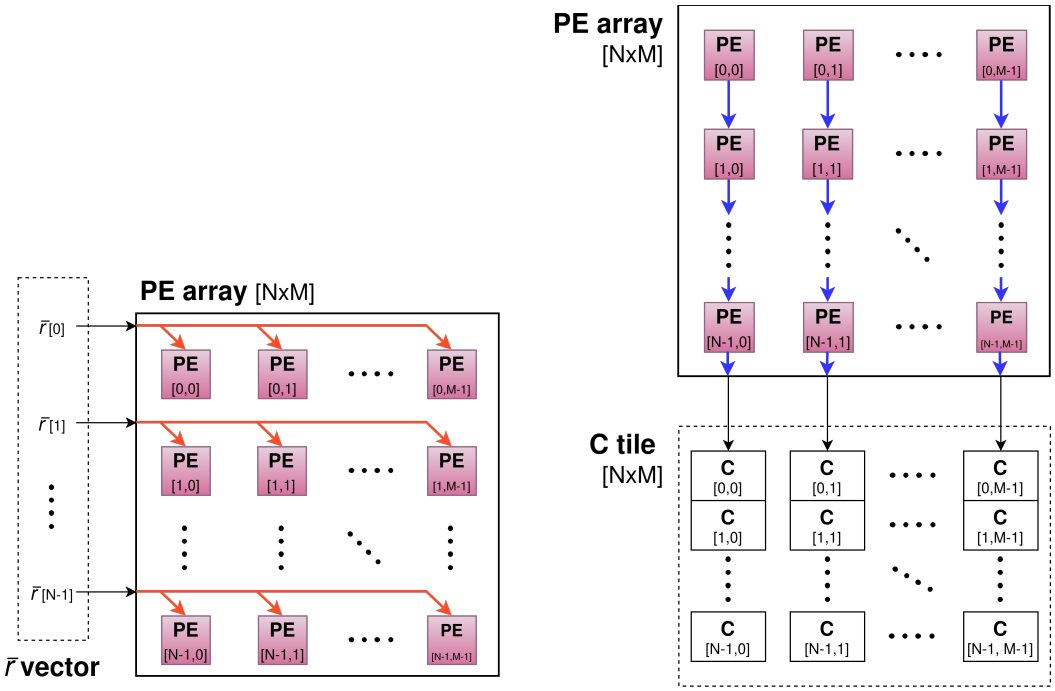


Figure 8: Systolic array MAC phase

At the core of the systolic array is the interconnection of processing elements (PEs). The PE is the module in charge of performing the elementary MAC/rescale operations and operand propagation. The complete schematic



(a) Systolic array rescale phase (b) Systolic array shift out phase

Figure 9

of the designed PE is shown in figure 10.
The presence of MUXes in the PE makes for an efficient reconfigurable architecture, which can reuse its internal functional units (multiplier, adder) to perform different operations, as shown in table 8.

Configuration	Figure	Description
MAC	11a	Multiply and accumulate, propagate A and B
Rescale	11b	Rescale accumulated result, propagate A and B
Shift out	11c	Propagate A,B and C

Table 8: PE configurations

To generate the necessary input skew in-hardware, a skew pipe element has been designed, as seen in figure 12a. This is a lower-diagonal 2D array of registers, which receives an entire input vector $\overline{A}$ or $\overline{B}$, and forces a latency (in CC) equal to each vector element's position.
A complete hardware scheme of the baseline array is shown in figure 14a.

5.1.2 Fault-tolerant systolic array

The fault-tolerant systolic array is equipped with a HW-based ABFT [17] checksum scheme, similar to the one proposed in [19]. The implemented checksum scheme is easily integrated on top of the baseline design modularly. In principle, ABFT style checksum is implemented through the addition of three modules, as seen in table 9, and thorugh an additional column of PEs where checksums will be retained and processed. A complete scheme of the fault-tolerant systolic array is shown in figure 14b.

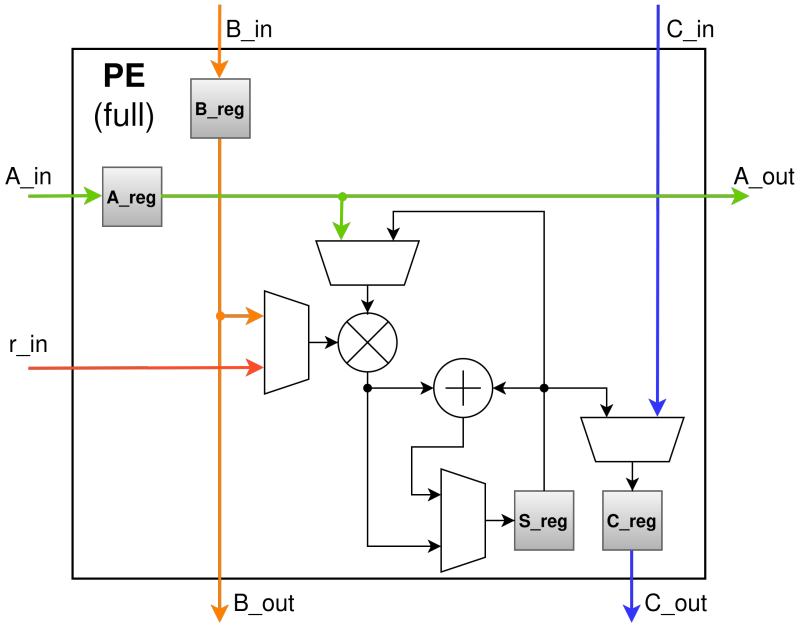
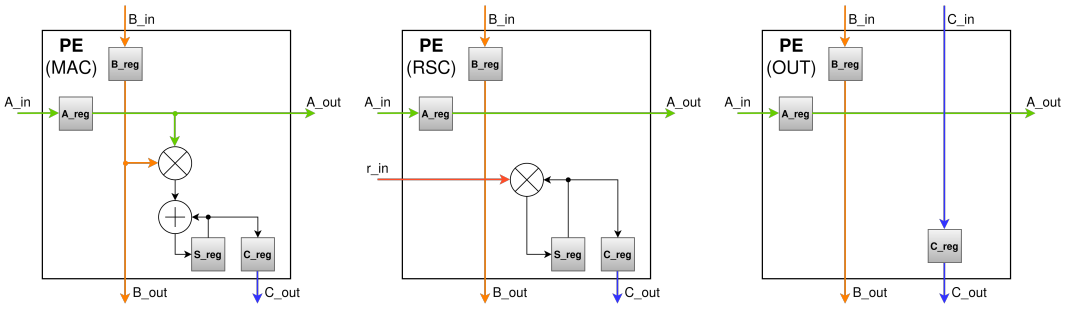
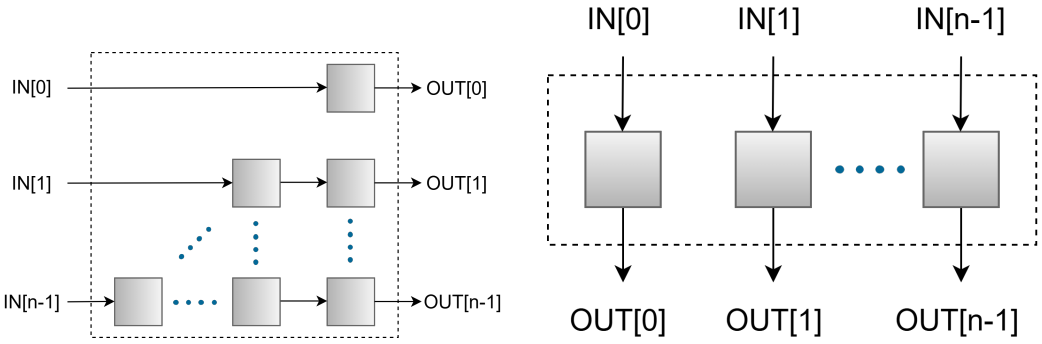


Figure 10: Processing Element (PE), full schematic



(a) MAC configuration (b) rescale configuration (c) shift out configuration

Figure 11: PE equivalent configurations

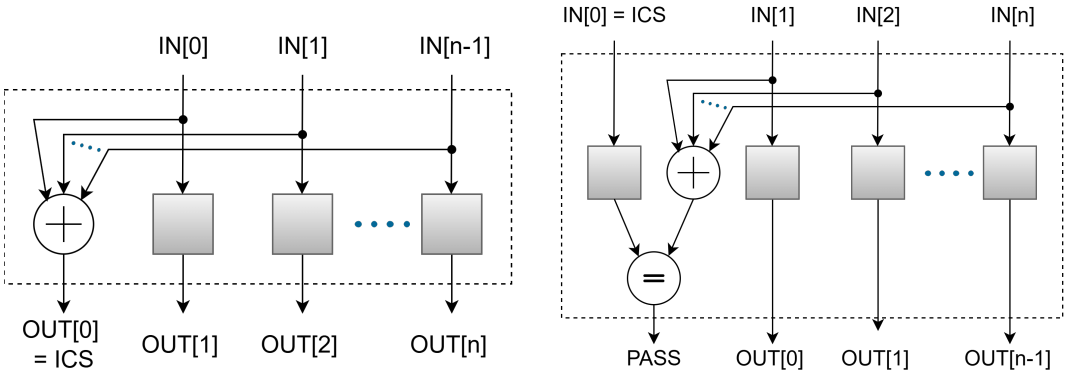


(a) Skew pipe (b) Synchronization register

Figure 12

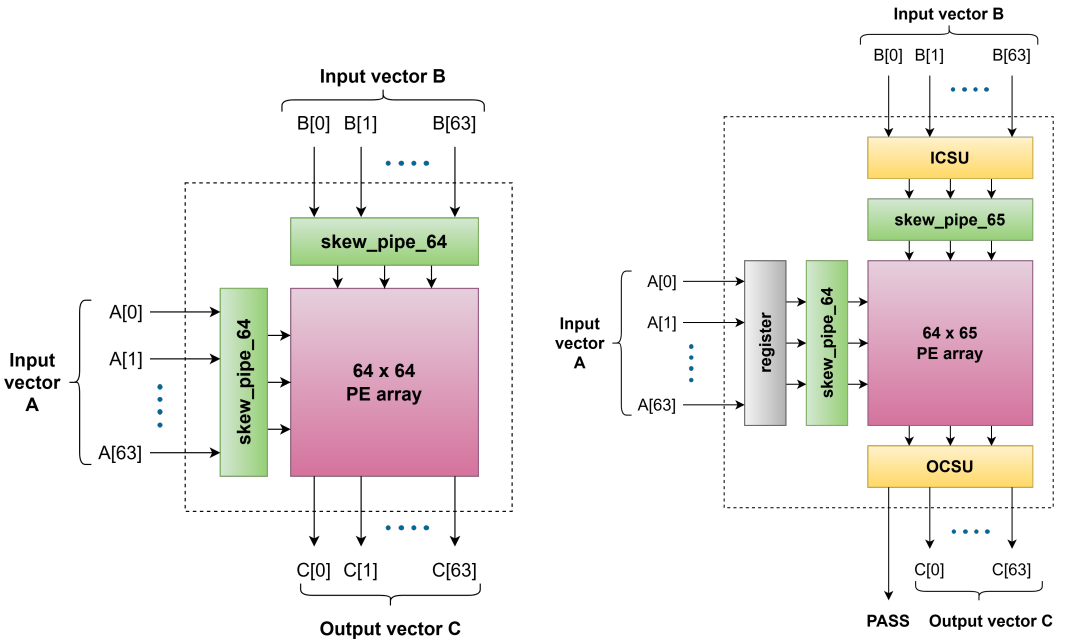
HW unit	Figure	Description
Input checksum unit (ICSU)	13a	Accumulates the row checksum of matrix B during input shift (pre-matrix computation).
Output checksum unit (OCSU)	13b	Accumulates row checksum of matrix C during output shift (post-matrix computation), compares IC and OC to determine if a fault has occurred.
Synchronization register	12b	Register array used to delay $\overline{A}$ vector by 1CC to match $\overline{B}$ vector latency.

Table 9: Additional units for ABFT checksum



(a) Input CheckSum Unit (ICSU) (b) Output CheckSum Unit (OCSU)

Figure 13



(a) Baseline Systolic Array (b) Fault-Tolerant Systolic Array

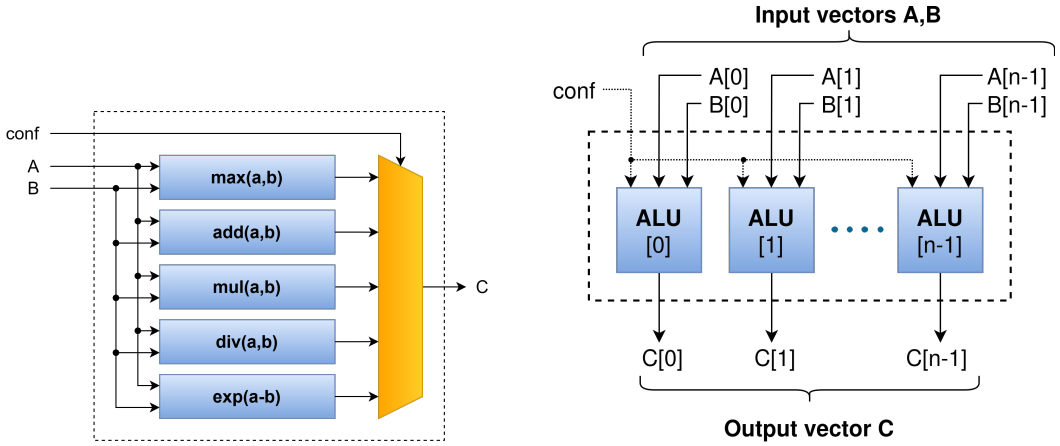
Figure 14

5.2 Arithmetic and logic units

The adoption of the FlashAttention [8] (algorithms 2 and 6) implies a wide range of operations need to be performed. These are listed below:

- Maximum
- Addition
- Multiplication
- Division
- Exponential

A fully-pipelined arithmetic and logic unit (ALU) was thus designed to perform these operations. The schematic is shown in figure 15a.
Additionally, since these operations are performed on vectors, a vectorized ALU (VALU) module has been designed by instantiating multiple ALUs in single instruction, multiple data (SIMD) fashion, as seen in figure 15b.



(a) Arithmetic and Logic Unit (ALU) (b) SIMD Vectorized ALU (VALU)

Figure 15

5.3 Top-p candidate selection unit

A special hardware module has been designed to support top-p candidate selection. In principle, this performs a hardware-based descending sort, to then find cumulative probabilities, and select the remained tokens based on the p budget. Since top-p is only performed sporadically, this module has been kept unpipelined to allow for a more aggressive optimization.

5.4 Hadware synthesis and optimization

5.4.1 Sizing and datatypes

For the three modules, a common hadware dimension of 64 was chosen. This dimension means the systolic array operates on 64×64 tiles, and the VALU and top-p unit operate on 1×64 vectors. The reasoning behind this choice is that it matches the head dimension for the benchmarked LLMs (next section), and made for easier mapping and higher utilizations.

The datatypes used by each functional unit are listed in table 10.

Functional unit	**Operand datatype**	**Reason**
Systolic array	32-bit fixed-point	Floating-point not needed, fixed-point more cost-effective.
VALU	32-bit floating-point	Operates on wide range of magnitudes.
Top-p unit	8-bit fixed-point	Operates only on 8-bit probability distributions as part of the prediction scheme.

Table 10: Functional units datatypes

Final remarks on datatypes:

- The systolic array is also used to operate on 8-bit fixed-point data in the prediction scheme through zero padding the 24 LSBs;
- While accumulating checksums might incur in overflow, if this is treated as wrap-around (instead of saturation), then the ABFT checksum property is preserved. The input and output checksum will overflow by the same amount and coincide in absence of faults.

5.4.2 Latency

In table 11, the latencies relative to all functional units are listed. Note:

- While all SA submodules (PE, skew pipe...) are pipelined, the SA itself cannot be pipelined by design: for a new tile to be calculated, the previous one has to be shifted out completely;
- The fault tolerant systolic array has an extra latency of 2CCs, introduced by the ICSU and OCSU;
- The systolic array latency is proportional to the inner dimension of the matrix multiplication: $\mathbf{A} \in \mathbb{R}^{64\times s}$, $\mathbf{B} \in \mathbb{R}^{s\times 64} \rightarrow \mathbf{C} = \mathbf{A} \times \mathbf{B} \in \mathbb{R}^{64\times 64}$;
- The systolic array tile rescale phase takes 1CC;
- As previously mentioned, the top-p unit was kept unpipelined to achieve a lower area during optimization.

Functional unit	**Latency** $[CC]$	**Pipelined**
SA (baseline) SA (fault-tolerant)	$192 + s$ $194 + s$	No (by design)
VALU	30	Yes
Top-p unit	53	No (by choice)

Table 11: Functional units latencies

5.4.3 Synthesis reports

The designed hardware was synthesized successfully at the target clock frequency of $f_{clk} = 1GHz$ using the GPDK045 design library. The optimized area reports are collected in tables 12, 13, and 14 for the systolic array (in its baseline and fault-tolerant versions), VALU, and top-p unit, respectively.

Component	**Baseline SA**			**Fault-tolerant SA**		
	Units	**Area** $[\mu m^2]$	**% Tot.**	**Units**	**Area** $[\mu m^2]$	**% Tot.**
PE	$64 \times 64 = 4096$	$16,502 \times 4096 = 67,592,192$	98.78	$64 \times 65 = 4160$	$16,502 \times 4160 = 68,648,320$	98.58
skew pipe ($n = 65$)	-	-	-	1	389,825	0.56
skew pipe ($n = 64$)	2	$2 \times 378,225 = 756,450$	1.10	1	378,225	0.54
top-level logic	1	84,440	0.12	1	84,498	0.12
ICSU	-	-	-	1	53,987	0.08
OCSU	-	-	-	1	53,905	0.08
synch. register	-	-	-	1	25,215	0.04
Total		68,433,082	100		69,633,975	100

Table 12: Systolic array synthesis area report

From table 12, it can be seen that the area overhead of the ABFT checksum scheme is a very low 1.75%. Also, the PE array is most expensive part of the SA, representing almost the entirety of its area.

Component	**VALU**		
	Units	**Area** $[\mu m^2]$	**% Tot.**
ALU	64	$99,556 \times 64 = 6,371,584$	100

Table 13: VALU synthesis area report

Module	**Area** $[\mu m^2]$
Top-p	164,717

Table 14: Top-p unit synthesis area report

Module	**Baseline**		**Fault-tolerant**	
	Area $[mm^2]$	**% Tot.**	**Area** $[mm^2]$	**% Tot.**
Systolic array	68.433	91.18	69.634	91.42
VALU	6.372	8.60	6.372	8.36
Top-p unit	0.165	0.22	0.165	0.22
Total	74.969	100	76.170	100

Table 15: Synthesis area report, comparison between functional units

From table 15: the systolic array is the most expensive unit, making up to 91% of the total area, followed by the VALU, and with the top-p unit being almost negligible.

6 Benchmarking

This section expands on the benchmarking strategies used to evaluate the effectiveness of the design, exploring the impact packed token pruning has on memory traffic, accuracy, and showing fault-injection campaign results.

6.1 Memory traffic estimation

To show the impact token pruning has on memory, the baseline and sparse algorithms' memory traffics have been estimated empirically for varying levels of sparsity (p-budget).
The two algorithms are comprehensive of the following steps:

Baseline:

1. TMUL-based QKV generation;
2. TMUL-based self-attention 1;
3. FlashAttention style fused softmax and attention output 2;
4. TUML-based FFN.

Sparse:

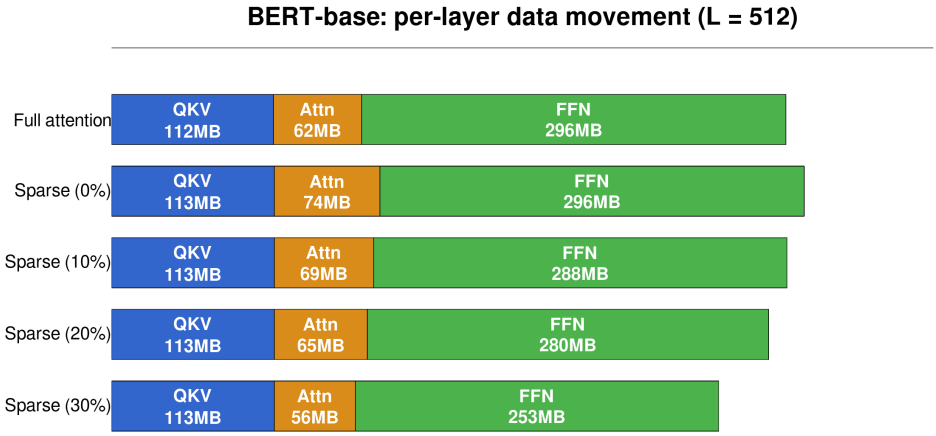
1. TMUL-based QKV generation (Q and K also written back in low-precision format);
2. TMUL-based low-precision self-attention 3;
3. top-p candidate selection 4;
4. TMUL-based full-precision token-pruned attention 5;
5. FlashAttention style token-pruned fused softmax and attention output 6;
6. TUML-based token-pruned FFN.

The memory traffic in Bytes has been estimated for the models in table 16.

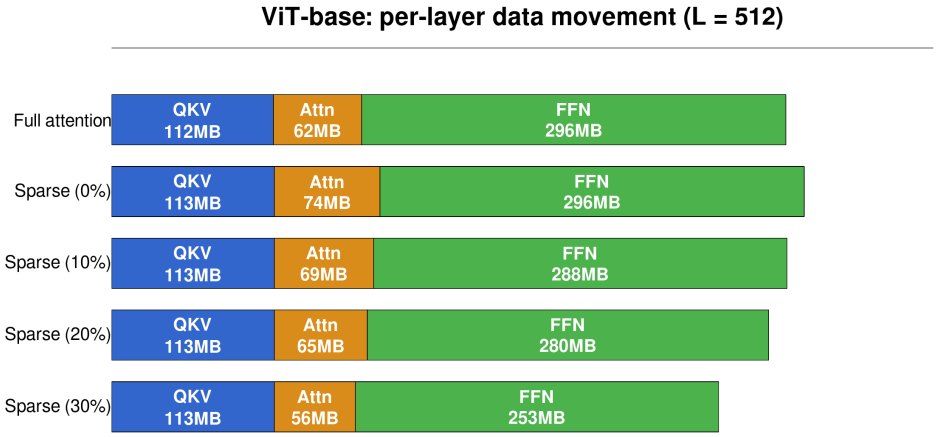
Model	**N° parameters**	$\mathbf{d_{model}}$	$\mathbf{d_{head}}$	**h**	$\mathbf{d_{FFN}}$
BERT-base [4]	110M	768	64	12	3072
ViT-base [20]	110M	768	64	12	3072
LLaMa 3.2-1B [21]	1B	2048	64	32	8192

Table 16: Analyzed LLMs and their parameters

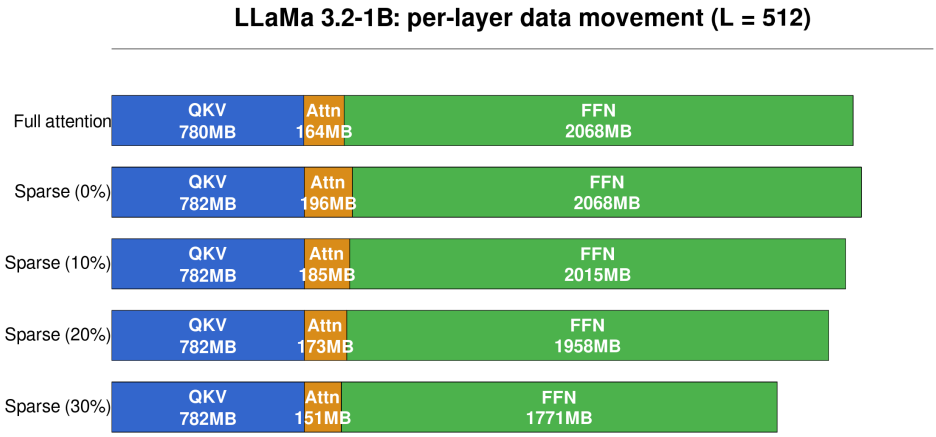
Figure 16 shows the total data movement (Bytes read and written) when running the baseline and sparse algorithms for a single attention layer (all heads) for the three models, with a fixed sequence length of $L = 512$ and varying levels of sparsity. The plots show how for 0% sparsity ($p = 1$), the sparse algorithm has a memory overhead compared to the baseline one. This is expected, as steps 4:6 of the sparse algorithm will coincide with steps 2:4 of the baseline one, with the addition of steps 2,3 and with step 1 having to write back the Query and Key matrices also in low-precision (8-bit fixed point) format.
As sparsity increases (i.e. choosing a lower p-budget) memory traffic drops for the sparse algorithm, with the main contribution coming from the token-pruned sparse FFN.



(a) BERT-base



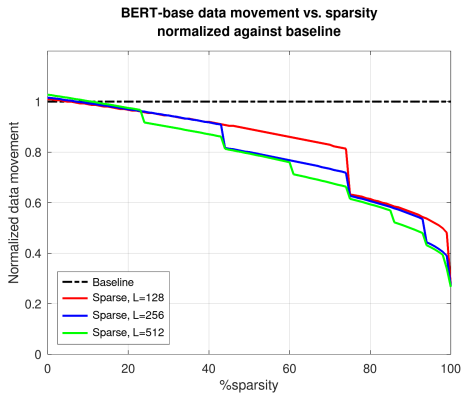
(b) ViT-base



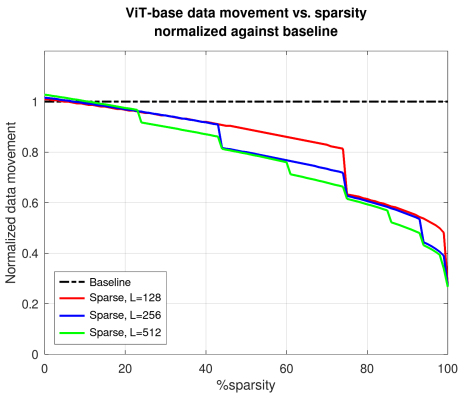
(c) LLaMa 3.2-1B

Figure 16: Per-layer data movements, $L = 512$

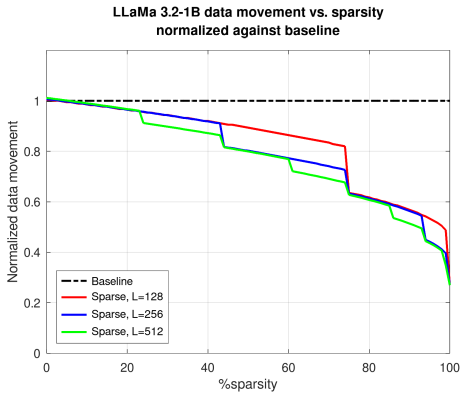
Figure 17 is obtained by repeating the same analysis parametrically for $L = \{128, 256, 512\}$ over a continuous range of sparsity values. Each curves is normalized against the baseline memory traffic (which is independent of sparsity) of corresponding sequence length.
The plots show a consistent trend: at 0% sparsity, the sparse algorithm has a small memory traffic overhead compared to the baseline algorithm. Around 10% sparsity, the memory traffics of the two algorithms are equivalent. Above that threshold, normalized memory traffic consistently drops for the sparse scheme.
Notice the step-like "jumps" in the sparse attention curves: the steep drops occur when sparsity is such that the remaining tokens are partitioned over one less tile dimension. When this happens, the TMUL algoritm's loop limits T_A, T_B or T_S (based on algorithm 5 or 6) shrink, and the number of tile re-fetches is reduced compared to the baseline scheme.



(a) BERT-base



(b) ViT-base



(c) LLaMa 3.2-1B

Figure 17: Normalized data movements vs. sparsity

6.2 Accuracy evaluation

The accuracy degradations of LLMs under adoption of the custom token-pruned self-attention algorithm were evaluated in Python.
In order to do this, pre-trained models and datasets (on which the models were already fine-tuned) were downloaded from HuggingFace Hub [22], an online repository which provides models and datasets compatible with the HuggingFace [22] Transformers library. Then, custom PyTorch [23] models were written to modify the self-attention scheme and include token pruning [5]. Finally, supervised accuracy evaluations were run for the baseline and token-pruned algorithms at varying levels of sparsity.

The tested LLMs and corresponding datasets are listed in table 17.

Model	**N° parameters**	**Dataset**	**Task**
BERT-base [4]	110M	imdb [24]	sentiment analysis
ViT-base [20]	110M	cifar-10 [25]	image classification
LLaMa 3.2-1B [21]	1B	SST-2 [26]	sequence classification

Table 17: LLMs and datasets on which they have been trained

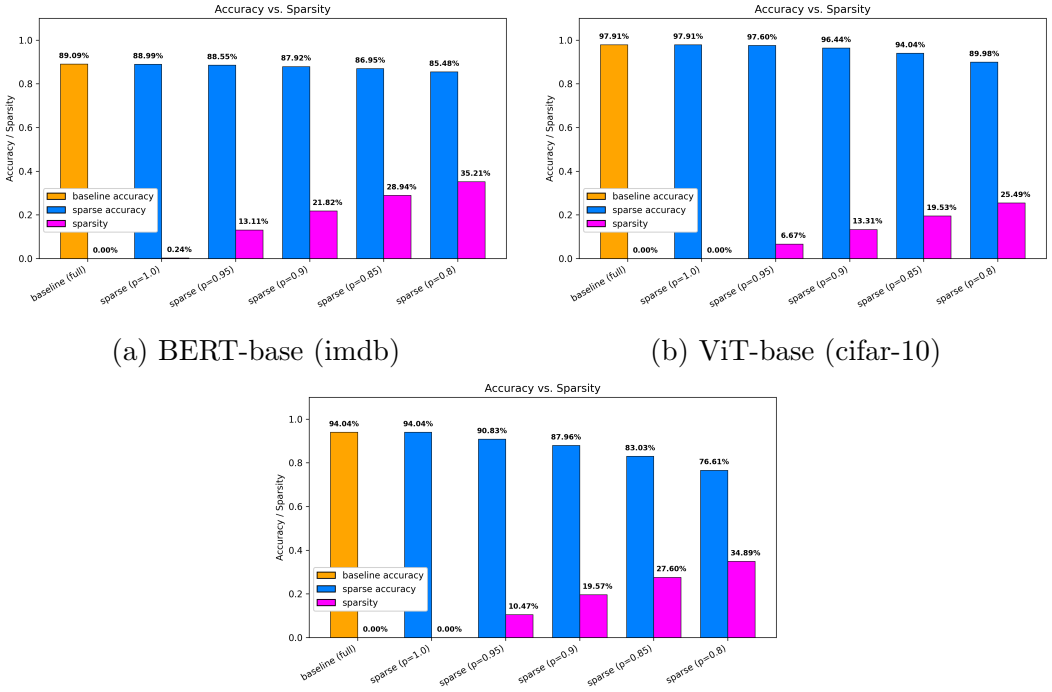
Figure 18 shows the findings from this analysis. Some remarks:

- As the figures show, there is no direct proportionality between the p-budget and sparsity, which depends on how focused or diffuse attention is on a case-to-case basis;
- While BERT and LLaMa models show comparable sparsity, the former shows a much smaller accuracy drop. This may be due to the task being sentiment analysis (binary output over a single neuron) being more robust to pruning than sequence classification (output over multiple neurons);
- Testing the Vision Transformer (ViT) over an image classification task showed more moderate accuracy degradations than LLaMa, but achieved the smallest sparsity, indicating attention to be more diffuse.

6.3 Fault injection campaign

A fault injection campaign was run to evaluate the effectiveness of the fault-tolerant systolic array's ABFT checksum scheme. The fault-injection framework adopted follows the one described in [27], and consisted in injecting a stuck-at-fault by flipping a random bit in the accumulation register of a single random PE at a time. The results are shown in table 18. Since each injected fault was detected by the scheme, the implementation was deemed successful.

[5]Note: since the evaluations were performed on PyTorch using CUDA acceleration, in absence of specialized CUDA kernels sparsity was introduced via masked token pruning. This does not in any way affect accuracy, as the background section showed how packed and masked token pruning lead to interchangeable results.



(a) BERT-base (imdb)

(b) ViT-base (cifar-10)

(c) LLaMa 3.2-1B (SST-2)

Figure 18: Accuracy evaluations for different LLMs and benchmarks

Faults injected	10,000
Faults detected	10,000
Outcome	Pass

Table 18: Fault injection campaign report

7 Conclusions

This work showed how sparse attention via token pruning can be effective at reducing memory traffic at the cost of contained accuracy degradations. The implemented hardware provides a cost-effective compute support for all operations within the attention layer, and can be equipped with a fault-tolerant ABFT checksum scheme to protect matrix computations at very low area overhead ($< 2\%$). Moreover, the sparse self-attention scheme is supported through the addition of the top-p candidate selection unit, which occupies 0.22% of the total area and represents a negligible overhead in cases where full self-attention is preferred over sparsity.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[2] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[3] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall, May 2024.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gerald Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[6] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, pages 9–16, New York, NY, USA, 2021. ACM.

[7] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, pages 16344–16359, 2022.

[9] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. *arXiv preprint arXiv:2012.09852*, 2020.

[10] Kazi Ahmed Asif Fuad and Lizhong Chen. A survey on sparsity exploration in transformer-based accelerators. *Electronics*, 12(10):2299, 2023.

[11] Tae Jun Ham, Yejin Lee, Seong Hoon Seo, Soosung Kim, Hyunji Choi, Sung Jun Jung, and Jae W. Lee. ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 692–705. IEEE, 2021.

[12] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. A^3: Accelerating attention mechanisms in neural networks with approximation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 328–341. IEEE, 2020.

[13] Yubin Qin, Yang Wang, Dazheng Deng, Zhiren Zhao, Xiaolong Yang, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. FACT: FFN-attention co-optimized transformer architecture with eager correlation prediction. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, pages 22:1–22:14. ACM, 2023.

[14] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. ACM, 2021.

[15] Chaofan Lin, Jiaming Tang, Shuo Yang, Hanshuo Wang, Tian Tang, Boyu Tian, Ion Stoica, Song Han, and Mingyu Gao. Twilight: Adaptive attention sparsity with hierarchical top-p pruning. *arXiv preprint arXiv:2502.02770*, 2025.

[16] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 4 edition, 2013.

[17] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, June 1984.

[18] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[19] Sandeep Bal, Chandra Sekhar Mummidi, Victor da Cruz Ferreira, Sudarshan Srinivasan, and Sandip Kundu. A novel fault-tolerant architecture for tiled matrix multiplication. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 1–6. IEEE, 2023.

[20] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*, 2021.

[21] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. In *Advances in Neural Information Processing Systems*, 2024.

[22] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, 2020. Association for Computational Linguistics.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035, 2019.

[24] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, 2011. Association for Computational Linguistics.

[25] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[26] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA, 2013. Association for Computational Linguistics.

[27] Udit Kumar Agarwal, Abraham Chan, Ali Asgari Khoshouyeh, and Karthik Pattabiraman. Towards reliability assessment of systolic arrays against stuck-at faults. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Supplemental Volume (DSN-S)*, pages 230–236. IEEE, 2023.

8 Acknowledgements

I would like to sincerely thank my supervisor in Taiwan, Professor Liou, for welcoming me into his research group and for guiding me throughout the process of working on this thesis.

I would also like to thank Licia, Enrico, Tommaso, and Elena for their invaluable personal and emotional support while I was away.

Finally, I wish to express my deepest gratitude to my fellow teammates at JJLab, as well as to all the kind people I have met in Taiwan, who made me feel at home despite being so far away from it.