Paper Seminar Presentation #1

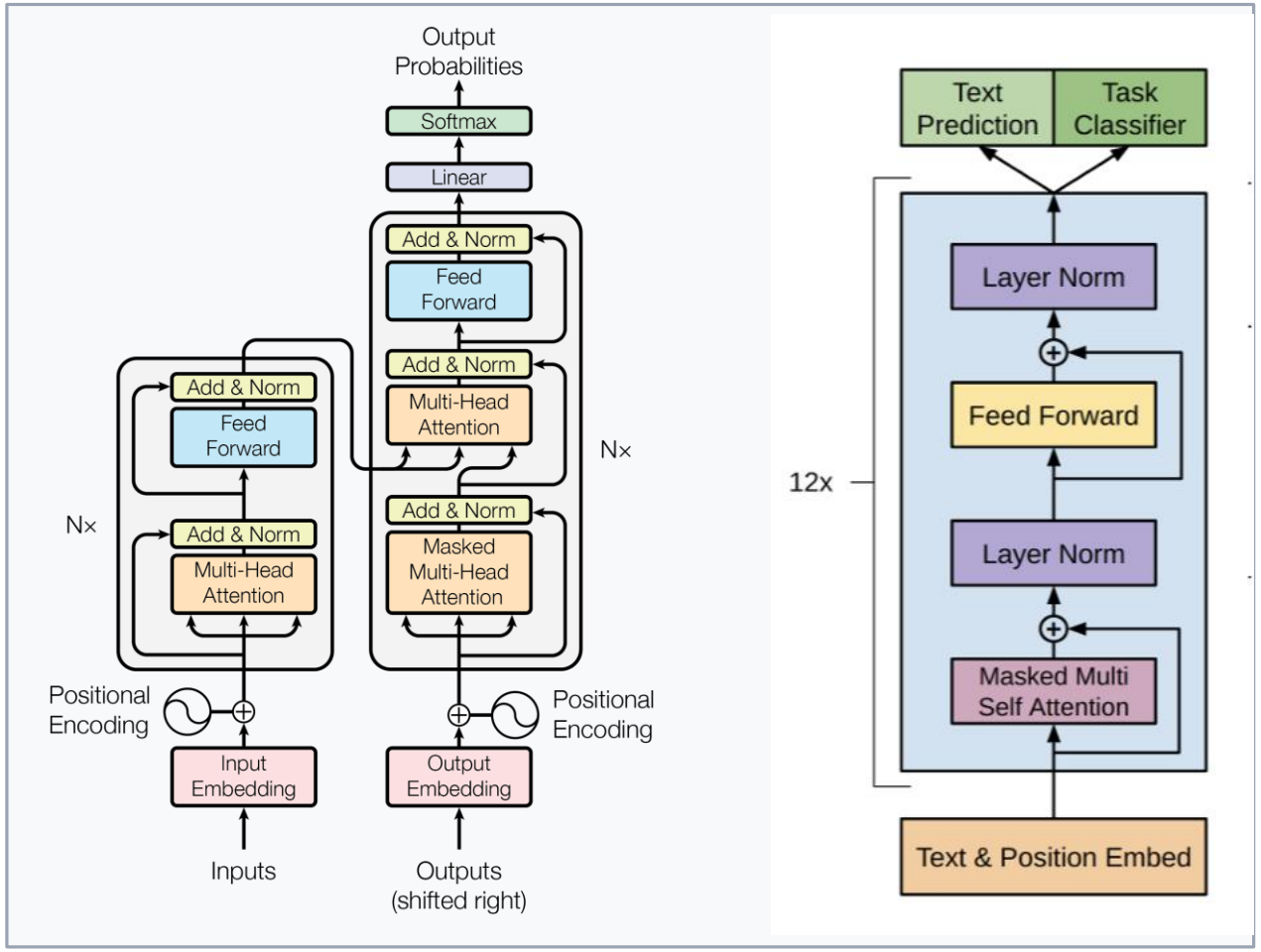
SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning, HPCA 2021

2022142255 SeokChan Jeong

Table of Content

Motivation and Background

Attention is all you need(NIPS 2017)
Improving Language Understanding by Generative Pre-Training(GPT-1, 2018)



- **Transformer**

  CNN -> Attention

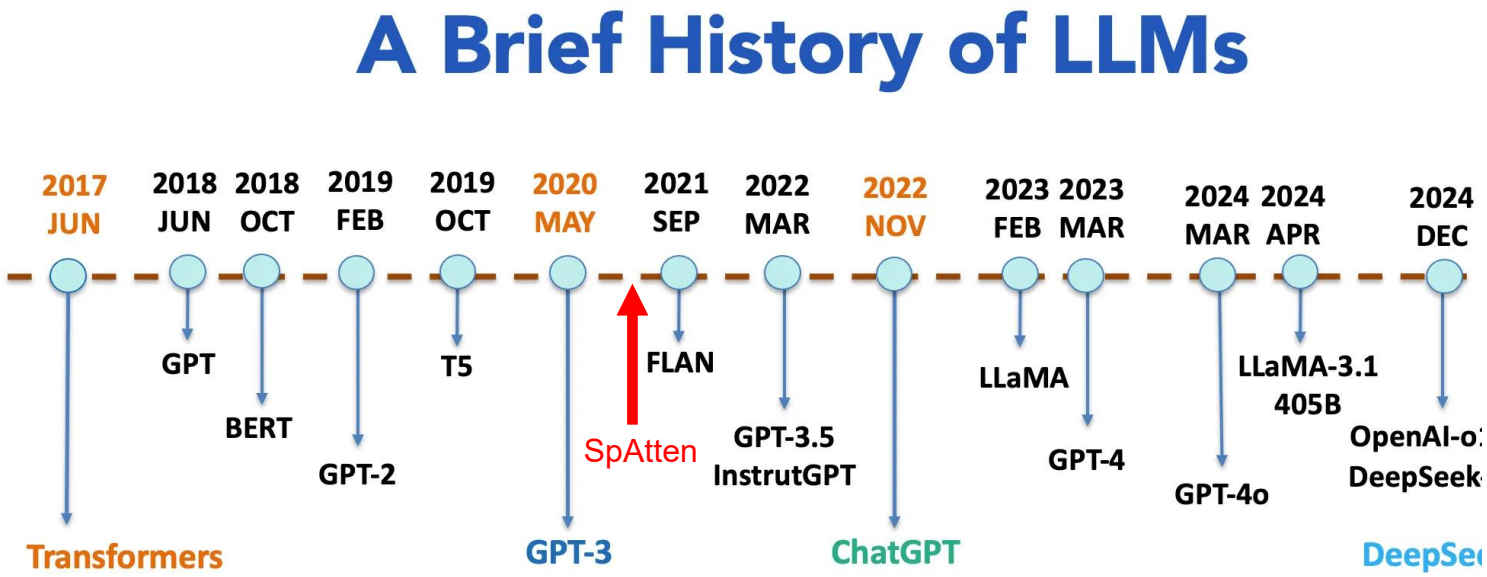
- **What became different with CNN&RNN?**

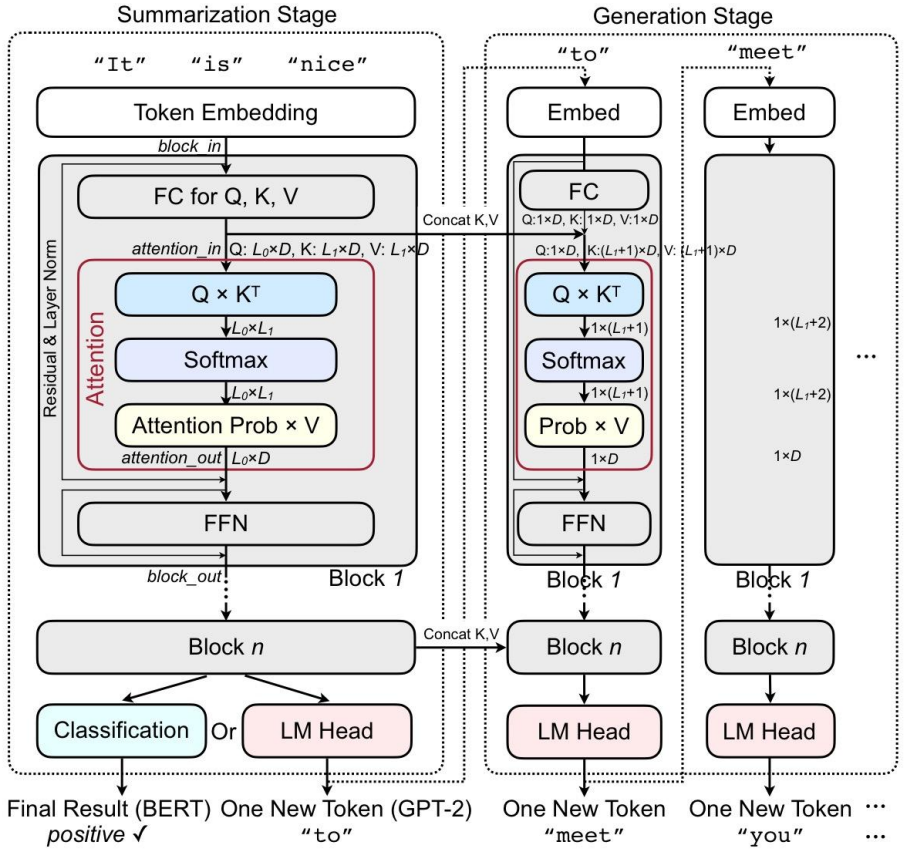
  Transformers calculate massive parallelism by GPU more effectively than CNN & RNN

Therefore

Transformer model become dominant
+ Require new acceleration method

BERT & GPT-2





- **GPT-1 vs BERT-Base**

  Encoder-only, 117M parameter, Bidirectional -> Understand language more, Computation-bound

- **GPT-1 vs GPT-2**

  Decoder-only, 1.5B parameter, Scaling up, Memory-bound → SpAtten

Background Knowledge

Token & Head (BERT-base)

As	a	visual	treat	,	the	film	is	almost	perfect	.

- **Token**

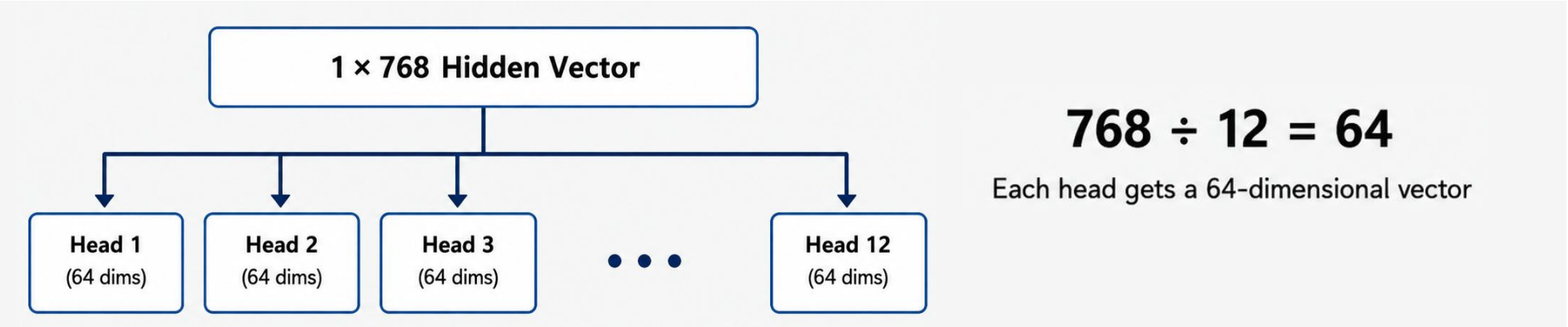
  Fundamental unit of text that an AI model processes and understands

film → [0.12, -0.38, 0.57, ..., 0.21] → 1x768 Vector

Verb Adjective Noun → **Features**

- **Head**

  One parallel attention subspace that looks at token relationships form its own perspective



You don't have to understand every word

A few key words are often enough to identify positive or negative sentiment.

Example 1. Restaurant Review (Chinese)

Full Tokens: 这家餐厅的服务态度真的太糟糕了，再也不来了！
(이 식당 서비스는 정말 최악이고, 다시는 안 온다!)

Pruned Tokens: 餐厅 服务 糟糕 不再来
(식당, 서비스, 최악, 다시 안 옴)

Negative sentiment is still obvious from the key words alone.

Example 2. Game Review (Japanese)

Full Tokens: このゲームは本当に神ゲーで、最高でした。大満足です！
(이 게임은 정말 갓겜이고, 최고였으며, 대만족입니다!)

Pruned Tokens: ゲーム 神ゲー 最高 大満足
(게임, 갓겜, 최고, 대만족)

Positive sentiment is still obvious from the key words alone.

3 Optimization of SpAtten

- **Cascade Token Pruning**
- **Cascade Head Pruning**
- **Progressive Quantization**

↕

Direct Quantization

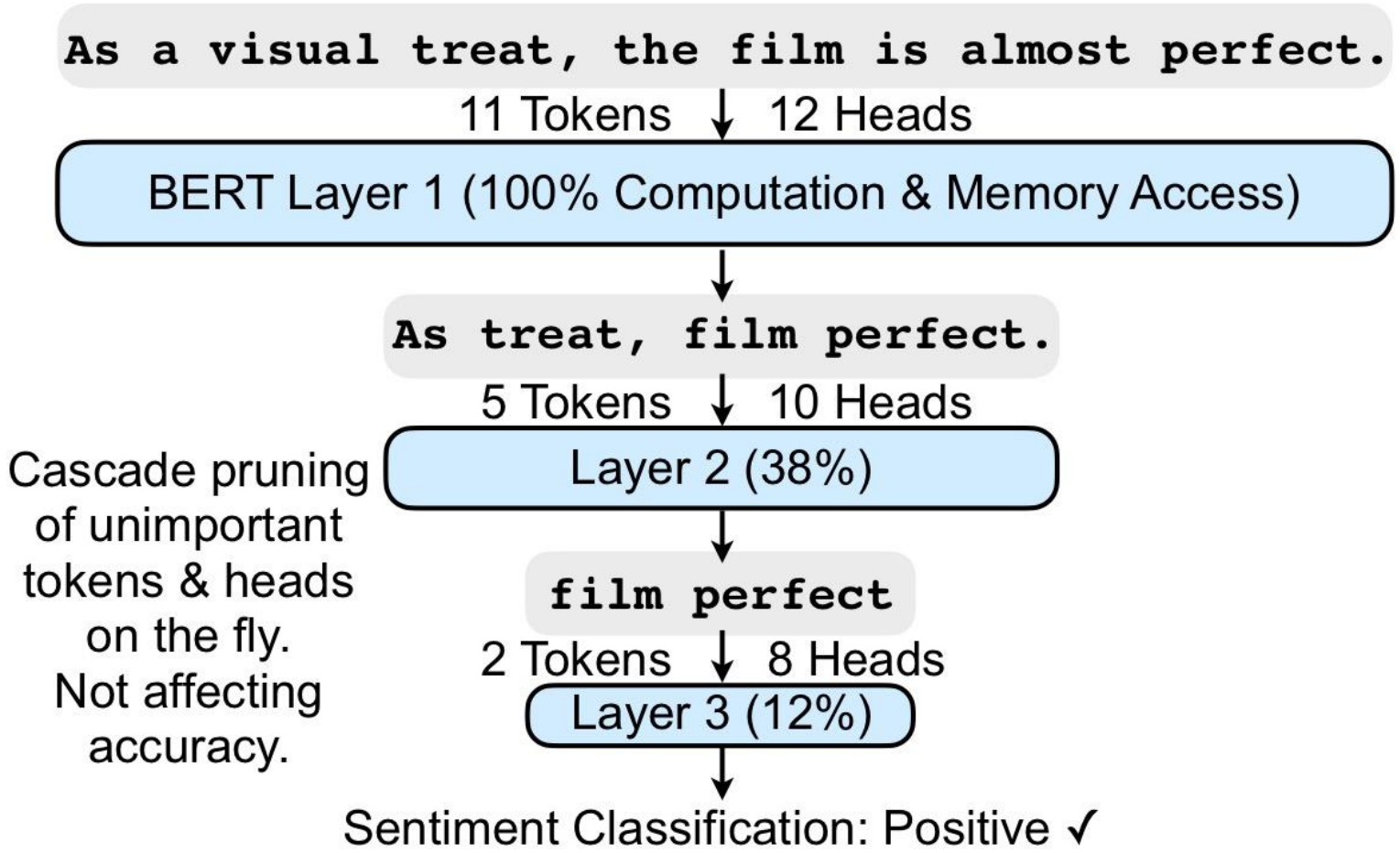
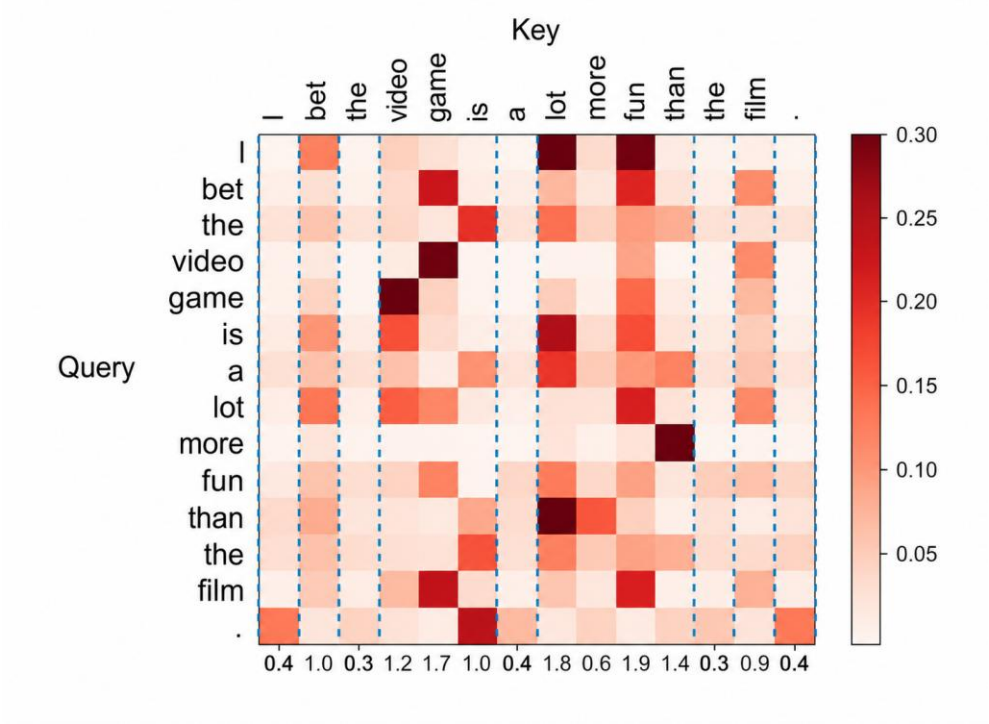


Fig. 1. *Cascade token and head pruning* removes redundant tokens and heads globally across layers. Evaluated with BERT-Base on SST-2 dataset.

Attention Mechanism

Attention score = $\frac{Q_i K_i^T}{\sqrt{D}}$



Cascade Token & Head Pruning

$$\text{Attention}(Q,K,V) = \underline{Softmax(\texttt{Attention score})}\,V = Softmax\left(\frac{QK^T}{\sqrt{D}}\right)V$$

Progressive Attention

Cascade Token Pruning vs Head Pruning

Example sentence: As a visual treat, the film is almost perfect.

Assume 11 tokens and 12 attention heads.

1. Original Token–Head Matrix

Token	Head 1	Head 2	Head 3	⋯	Head 12
As	●	●	●		●
a	●	●	●		●
visual	●	●	●		●
treat	●	●	●		●
,	●	●	●		●
the	●	●	●		●
film	●	●	●		●
is	●	●	●		●
almost	●	●	●		●
perfect	●	●	●		●
.	●	●	●		●

2. Token Pruning

Token	Head 1	Head 2	Head 3	⋯	Head 12
As	●	●	●		●
a	●	●	●		●
visual	●	●	●		●
treat	●	●	●		●
,	●	●	●		●
the	×	×	×	⋯	×
film	●	●	●		●
is	●	●	●		●
almost	●	●	●		●
perfect	●	●	●		●
.	●	●	●		●

Remove one token -> remove one entire row across all heads.

3. Head Pruning

Token	Head 1	Head 2	Head 3	⋯	Head 12
As	●	●	×		●
a	●	●	×		●
visual	●	●	×		●
treat	●	●	×		●
,	●	●	×		●
the	●	●	×		●
film	●	●	×		●
is	●	●	×		●
almost	●	●	×		●
perfect	●	●	×		●
.	●	●	×		●

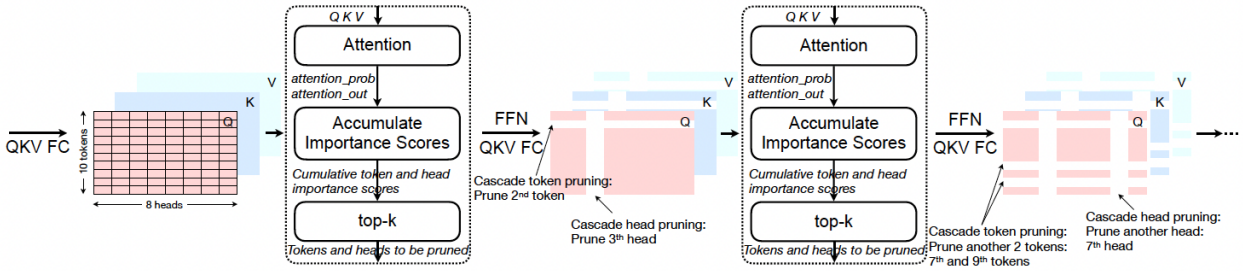
Remove one head -> remove one entire column across all tokens.

Key idea: Token pruning removes rows; head pruning removes columns.

Cascade Pruning vs Value Pruning

1. Cascade Pruning

Global & persistent pruning across later layers



What it removes

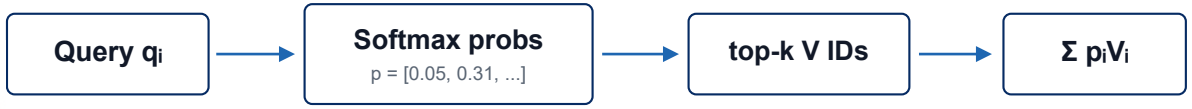
- tokens before later QKV/FFN
- whole heads in later layers

Effect

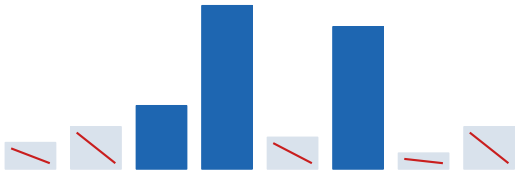
Persistent: once removed, later computation is skipped.

2. Local Value Pruning

Temporary pruning inside one query–head weighted sum



Current attention probability row



Only high-probability V vectors are fetched

What it removes

- low-probability Value vectors
- only for this query/head

Effect

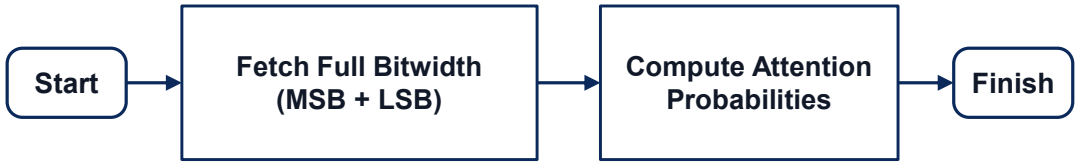
Local: tokens and heads still exist for other computations.

Aspect	Cascade Pruning	Local Value Pruning
Score source	Cumulative token/head importance scores	Current attention probabilities
Timing	Before later QKV/Attention/FFN computation	After Softmax, before probability × V
Scope	Global across later layers/heads	Local to one query and one head
Main saving	Computation + memory traffic	Value fetch + weighted-sum work

Key idea: Cascade pruning removes future work; local value pruning skips unnecessary V reads only for the current weighted sum.

Progressive Quantization vs Direct Quantization

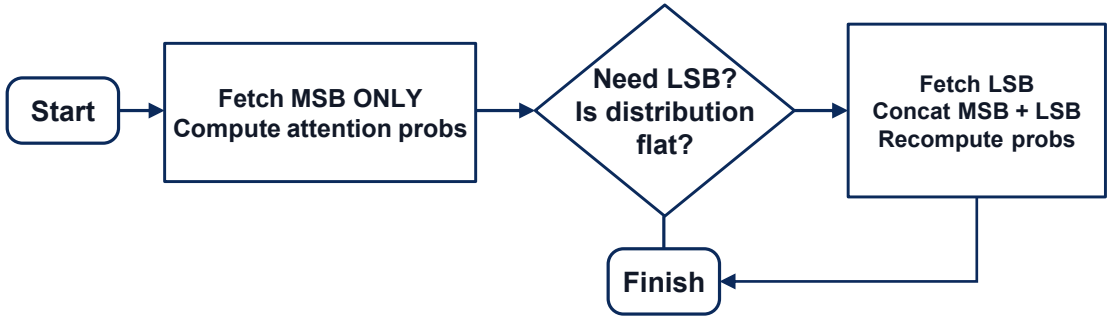
1. Direct Quantization



Same bitwidth for every input

- Fixed precision for all attention inputs
- No decision or recomputation step
- Simple control, but higher memory traffic

2. Progressive Quantization



Adaptive bitwidth by attention uncertainty

- Starts with MSB-only low precision
- Fetches LSB only when distribution is flat
- Lower average memory access and bandwidth

Aspect	Direct Quantization	Progressive Quantization
Bitwidth strategy	Fixed	Adaptive
Memory access	Always full bitwidth	Lower on average

Key idea: Progressive quantization saves memory by loading extra bits only when the attention distribution is uncertain.

Software Algorithm

Cascade Pruning Algorithm

Input tensors

$$P = \text{attention\_prob} \in \mathbb{R}^{h \times L_0 \times L_1}$$

$$E = \text{reshape}(\text{attention\_out}) \in \mathbb{R}^{h \times L_0 \times D}$$

$$s_t^{prev} \in \mathbb{R}^{L_1}, \quad s_h^{prev} \in \mathbb{R}^{h}$$

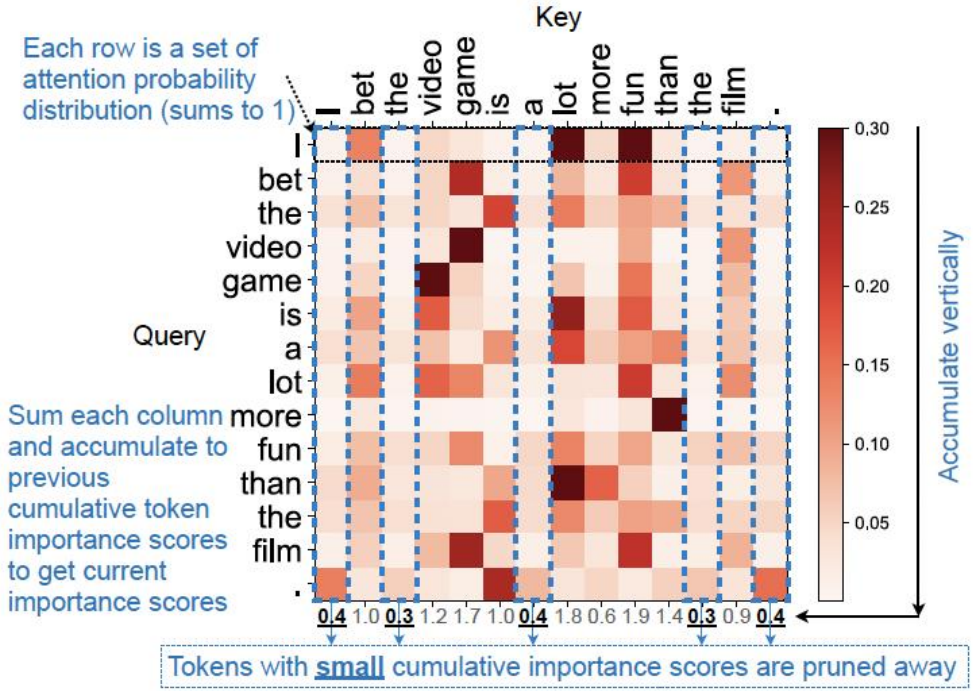
Symbols

h: current number of heads
L_0: number of query tokens
L_1: number of key/value tokens
D: feature dimension of one head

How to read the map

- Each row is one query token
- Each column is one key token
- Column-wise accumulation gives token importance
- Low-score tokens can be removed in later layers

Attention probability map



TopKIndex role

$\text{TopKIndex}(s^{new}, k)$: TopKIndex selects the indices of the k largest importance scores.

Cascade Pruning Algorithm

Token cumulative importance

1) Accumulate by key-token columns

$$s_t^{new}[j] = s_t^{prev}[j] + \Sigma_{m=1}^{h} \Sigma_{i=1}^{L_0} P_{m,i,j}, \quad j = 1, ..., L_1$$

Adds all probabilities that attend to key token j across heads and query tokens.

2) Determine how many tokens remain

$$k_t = \text{floor}(\, L_1(1 - p_t)\,)$$

p_t is the token pruning ratio; k_t is the number of token IDs to keep.

3) Select remaining tokens

$$T_{remain} = \text{TopKIndex}(\, s_t^{new}, k_t\,)$$

TopKIndex returns the k_t token IDs with the largest cumulative token scores.

Head cumulative importance

1) Accumulate by head output magnitude

$$s_h^{new}[m] = s_h^{prev}[m] + \Sigma_{i=1}^{L_0} \Sigma_{d=1}^{D} |E_{m,i,d}|, \quad m = 1, ..., h$$

A head is important if its attention output has large magnitude across tokens and features.

2) Determine how many heads remain

$$k_h = \text{floor}(\, h(1 - p_h)\,)$$

p_h is the head pruning ratio; k_h is the number of head IDs to keep.

3) Select remaining heads

$$H_{remain} = \text{TopKIndex}(\, s_h^{new}, k_h\,)$$

TopKIndex returns the k_h head IDs with the largest cumulative head scores.

Key difference: Token pruning removes rows across all heads; head pruning removes columns across all tokens.

Cascade Pruning Algorithm

Example with h = 2, L_0 = 2, L_1 = 4. The goal is to keep the two most important tokens.

Step 1. Start from attention probabilities P

Tokens:	As	the	film	perfect
Head 1, P[1,:,:]				
q_1	0.10	0.20	0.30	0.40
q_2	0.05	0.10	0.35	0.50
Head 2, P[2,:,:]				
q_1	0.20	0.10	0.40	0.30
q_2	0.10	0.05	0.50	0.35

Step 2. Accumulate token scores

	As	the	film	perfect
s_t_prev	0.20	0.10	0.40	0.30
+ $\Sigma_m \Sigma_i P$	0.45	0.45	1.55	1.55
s_t_new	0.65	0.55	1.95	1.85

Column sums are accumulated vertically across all heads and query tokens.

Step 3. Decide how many tokens remain

$p_t = 0.5$
$k_t = \text{floor}(4 \times (1 - 0.5)) = 2$

Step 4. Apply TopKIndex

TopKIndex([0.65, 0.55, 1.95, 1.85], 2)
= {film, perfect}

Result: film and perfect remain; lower-score tokens are pruned from later layers.

Cascade Pruning Algorithm

Use output magnitude to decide which attention heads remain active.

Step 1. Accumulate head scores from E

```
Head 1 E: [0.40, −0.20], [0.10, 0.30]
abs sum = 0.40+0.20+0.10+0.30 = 1.00

Head 2 E: [0.05, 0.10], [−0.10, 0.05]
abs sum = 0.05+0.10+0.10+0.05 = 0.30
```

Step 2. Add previous head importance

	Head 1	Head 2
s_h_prev	0.30	0.20
+ abs sum	1.00	0.30
s_h_new	1.30	0.50

Step 3. Decide how many heads remain

$p_h = 0.5$

$k_h = \text{floor}(2 \times (1 - 0.5)) = 1$

Step 4. Apply TopKIndex

```
TopKIndex([1.30, 0.50], 1)
= {Head 1}
```

Result: Head 1 remains; Head 2 is pruned across all tokens in later layers.

Key distinction: token pruning selects token IDs; head pruning selects head IDs.

Progressive Quantization Algorithm

Quantization reduces memory traffic, but some attention patterns need more precision.

1. Score error from quantization

Quantizing Q and K slightly perturbs the attention score.

$$\hat{s}_i = s_i + \Delta s_i$$

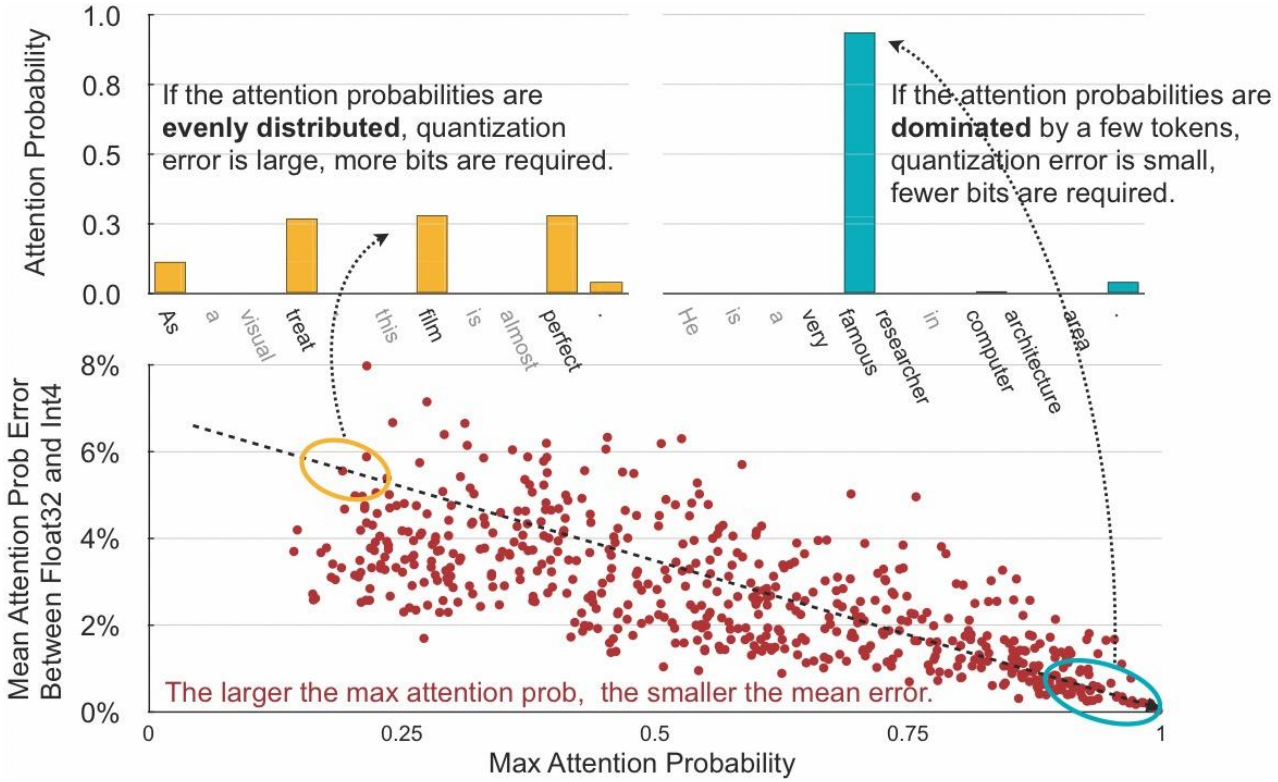
2. Softmax usually bounds the error

$$Error = 2\Delta s_0 p_0(1-p_0) \leq \frac{\Delta s_0}{2} < \Delta s_0$$

3. Error depends on the distribution

Flat distribution higher error rate → fetch LSB

Peaked distribution small error → MSB is enough



Key idea: Use MSB first. Add LSB only when attention probabilities are flat and quantization error becomes larger.

Progressive Quantization Algorithm

MSB-only estimate

$$P^{MsB} = \text{Softmax}(\, Q^{MsB} \, (K^{MsB})^{T} \,/\, \sqrt{D}\,)$$

Confidence Test

$$p_{max} = \max_j P_j^{MsB}$$

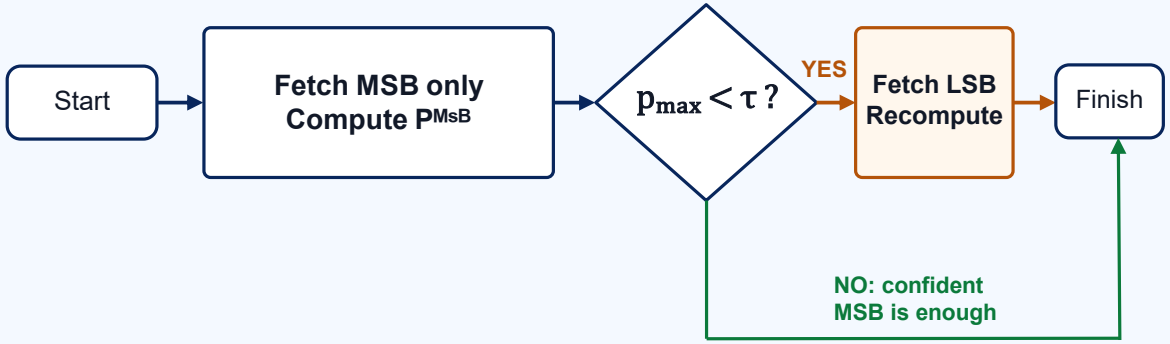
High p_{max} means one token dominates.
Low p_{max} means the distribution is flat.

Output Selection

$$P_{final} = \begin{cases} P^{MSB}, & p_{max} \geq \tau \\ P^{MSB+LSB}, & p_{max} < \tau \end{cases}$$

This is the algorithmic core of progressive quantization.

Decision Flow



Interpretation

Dominated distribution max probability is high → use MSB only

Flat distribution max probability is low → fetch LSB and recompute

Progressive Quantization Algorithm

Decision rule: start with MSB only, then fetch LSBs only when the attention distribution is uncertain.

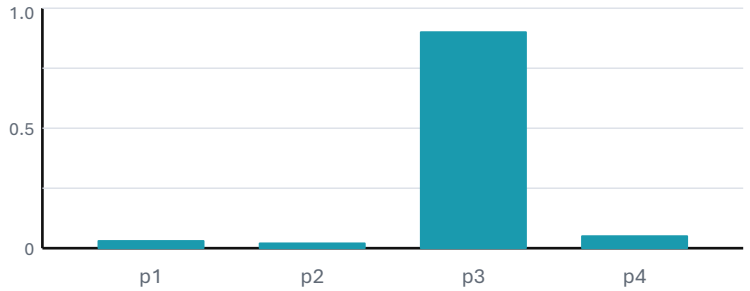
$$pmax = max(PMSB) \qquad Pfinal = PMSB \ \text{ if } \ pmax \geq \tau \qquad Pfinal = PMSB{+}LSB \ \text{ if } \ pmax < \tau$$

Illustrative threshold: $\tau = 0.30$

Case A. Peaked distribution

High variance: one token dominates.

MSB attention probabilities `[0.03, 0.02, 0.90, 0.05]`



pmax = 0.90 **0.90 ≥ τ**

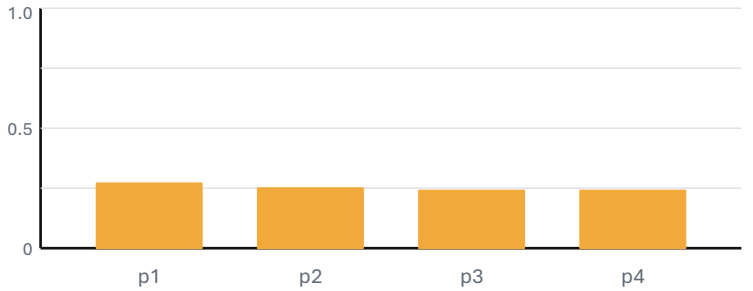
Use MSB result; no LSB fetch.

The dominant token remains clear even with low-bit quantization.

Case B. Flat distribution

Low variance: probabilities are close.

MSB attention probabilities `[0.27, 0.25, 0.24, 0.24]`



pmax = 0.27 **0.27 < τ**

Fetch LSB, combine MSB+LSB, and recompute.

Small quantization errors can change the ordering between tokens.

Key idea: flat attention distributions are more sensitive to quantization error, so SpAtten refines precision only for those cases.

Hardware Architecture

Zero Eliminator

Problem: filtering creates holes

0	1	2	3	4	5	6	7
a	0	b	0	c	d	0	e

Sparse vector after comparison

Zero Eliminator →

Output: dense stream

0	1	2	3	4	5	6	7
a	b	c	d	e	0	0	0

Valid elements keep the original relative order

Core operation

For each valid element, count how many zeros are before it, then shift it left by that count.

$$\text{new\_index}(x_i) = i - z_i$$

- Used after top-k filtering to remove zeros before pushing values into FIFO.
- Does not sort values; it only compacts valid items.
- Scores and their token/value/head IDs move together.

Zero Eliminator = order-preserving compression of nonzero elements

Zero Eliminator

For each position, compute the prefix count of zeros that appeared earlier in the vector.

Input vector

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7
a	0	**b**	0	**c**	**d**	0	**e**

Prefix zero count

0	0	1	1	2	2	2	3

zero_cnt

Element	Original index	Zeros before	Shift amount
a	0	0	0
b	2	1	1
c	4	2	2
d	5	2	2
e	7	3	3

$$z_i = \Sigma_{j=0}^{i-1} 1(x_j = 0)$$

$$\mathbf{1}(x_j = 0) = \begin{cases} 1, x_j = 0, \\ 0, x_j \neq 0. \end{cases}$$

Example reading
At position 4, the value is c. Before c, there are two zeros, so $z_4 = 2$.
At position 7, the value is e. Before e, there are three zeros, so $z_7 = 3$.

z_i tells how far each valid element must move left

Zero Eliminator

A valid element moves left by the number of zeros before it.

$$\text{new\_index}(x_i) = i - z_i$$

Mapping from original position to compacted position

Element	i	z_i	new index = $i - z_i$	Result
a	0	0	0 − 0 = 0	$y_0 = a$
b	2	1	2 − 1 = 1	$y_1 = b$
c	4	2	4 − 2 = 2	$y_2 = c$
d	5	2	5 − 2 = 3	$y_3 = d$
e	7	3	7 − 3 = 4	$y_4 = e$

Before

0	1	2	3	4	5	6	7
a	0	**b**	0	**c**	**d**	0	**e**

After

0	1	2	3	4	5	6	7
a	**b**	**c**	**d**	**e**	0	0	0

Zero Eliminator

Hardware does not need a large arbitrary shifter; it performs staged shifts of 1, 2, 4, ... positions.

Shift amount as binary

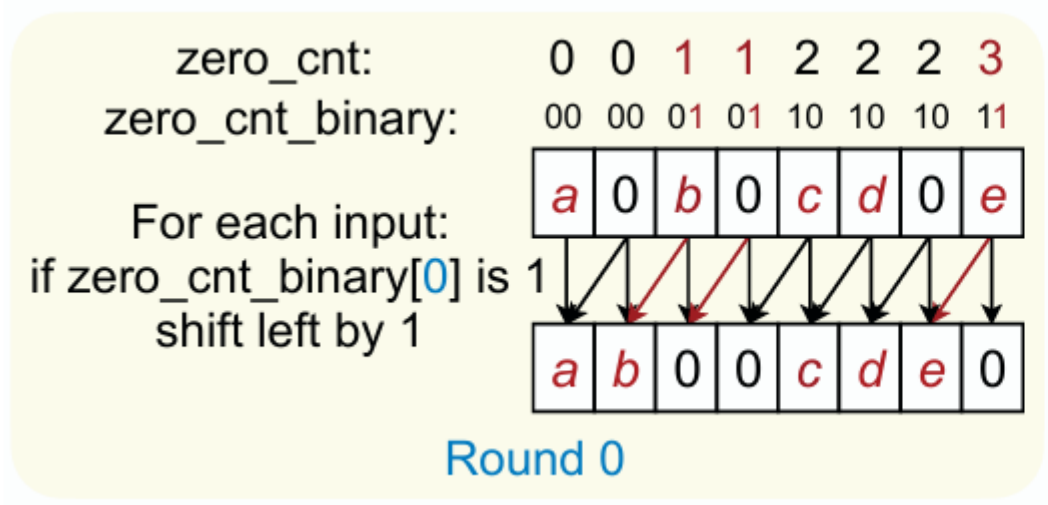
0	0	1	1	2	2	2	3
00	00	01	01	10	10	10	11

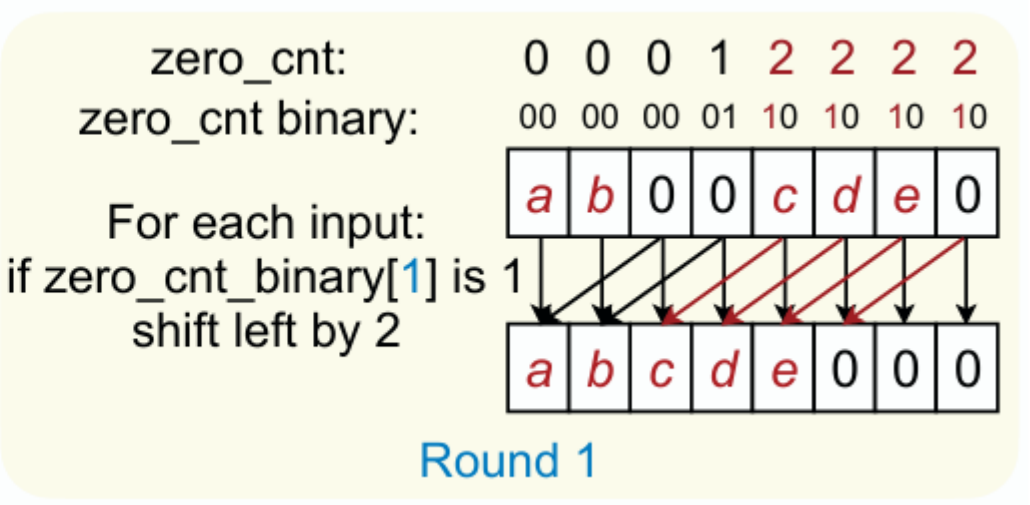
zero_cnt

binary

$$z_i = \Sigma_r \, b_{i,r} \, 2^r$$

Each bit controls one shift stage.





Zero Eliminator

The stage distances double each round, so n elements only require $\lceil \log_2 n \rceil$ shift stages.

$$N\_stage = \lceil \log_2 n \rceil$$

$$n = 8 \rightarrow \lceil \log_2 8 \rceil = 3 \text{ stages}$$

Stage distances

Round 0

bit 0

Shift left by 1 position

Round 1

bit 1

Shift left by 2 positions

Round 2

bit 2

Shift left by 4 positions

In the example, the maximum shift is 3, so only the 1-position and 2-position stages are actually used.

- A full 8-element design still includes a possible 4-position stage for the worst case.
- The hardware scales efficiently because stages grow as 1, 2, 4, 8, ... rather than one stage per possible shift distance.

Zero Eliminator uses staged parallel shifts, not serial movement

Quick select

Goal: select the top-k important tokens, heads, or values without fully sorting the whole array.

Core idea

Why this matters in SpAtten

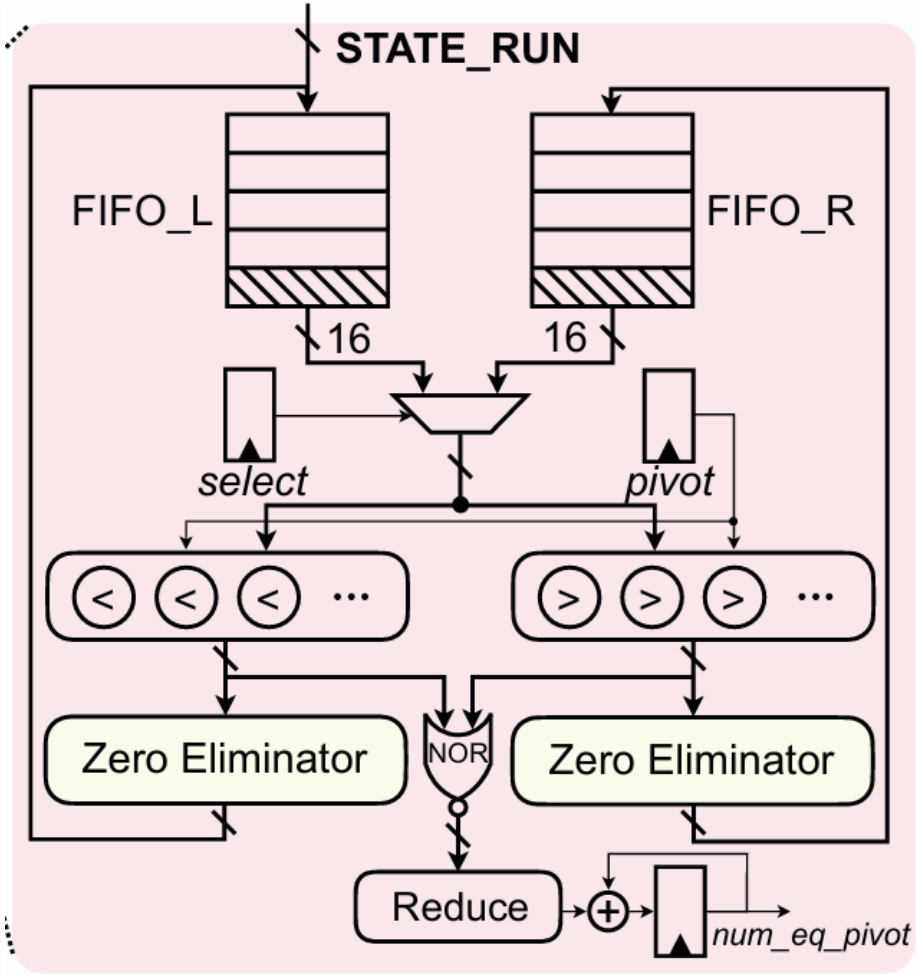
- Token pruning keeps only important token IDs.
- Head pruning keeps only important attention head IDs.
- Local value pruning keeps only high-probability value IDs.

Instead of sorting everything

- Quick Select finds the boundary value: the k-th largest score.
- The original order can be preserved when filtering.
- Hardware can process many comparisons in parallel.

Key message: Quick Select is a threshold finder for Top-k selection, not a full sorter.

Quick select



1 Select active FIFO

FIFO_L or FIFO_R is chosen depending on where the k-th value can still exist.

2 Choose a pivot

A random item from the selected FIFO becomes the current pivot.

3 Compare in parallel

16 values are compared at once: item < pivot and item > pivot.

4 Compact outputs

Zero Eliminators remove empty slots and pack valid values.

5 Count equal values

NOR + Reduce counts items equal to pivot as num_eq_pivot.

STATE_RUN performs only one partition. START decides whether to repeat or stop.

Quick select

After one partition, only one side needs to be searched again.

$$G = \{ x \mid x > p \} \quad E = \{ x \mid x = p \} \quad L = \{ x \mid x < p \}$$

Let p be the pivot, g = |G|, e = |E|, and r be the target rank.

Condition	Meaning	Next action
$r \le g$	There are already enough larger values.	Search G again
$g < r \le g + e$	The target rank lies in the equal-to-pivot block.	Stop: pivot is the threshold
$r > g + e$	Larger + equal values are not enough	Search L with $r \leftarrow r - g - e$

Output of Quick Select

A threshold θ = k-th largest value, not a fully sorted list.

Final filtering

Keep values > θ and only the needed number of values = θ.

The algorithm narrows the candidate set instead of ordering all scores.

Quick select

Selected FIFO: FIFO_L
Candidate set = [0.6, 0.1, 0.5, 1.2, 0.6]

Pivot
p = 0.1

Target
r = 4

Value x	Relation	Output
0.6	x > p	FIFO_R
0.1	x = p	Equal
0.5	x > p	FIFO_R
1.2	x > p	FIFO_R
0.6	x > p	FIFO_R

FIFO_L = [ ]

FIFO_R = [0.6, 0.5, 1.2, 0.6]

num_eq = 1

|FIFO_R| = 4 and r = 4

Decision: r ≤ |FIFO_R|, so the pivot is too small and the next search moves to FIFO_R.

Quick select

Selected FIFO: FIFO_R
Candidate set = [0.6, 0.5, 1.2, 0.6]

Pivot
p = 0.6

Target
r = 4

Value x	Relation	Output
0.6	x = p	Equal
0.5	x < p	FIFO_L
1.2	x > p	FIFO_R
0.6	x = p	Equal

FIFO_L = [0.5]

FIFO_R = [1.2]

num_eq = 2

|FIFO_R| + num_eq = 1 + 2 = 3 < r = 4

Decision: the pivot is too large. One value in FIFO_R and two equal values are already fixed, so update the target to r = 4 - 3 =1 and search FIFO_L next.

Quick select

Selected FIFO: FIFO_L
Candidate set = [0.5]

Pivot
p = 0.5

Updated target
r = 1

Value x	Relation	Output
0.5	x = p	Equal

FIFO_L = [ ]

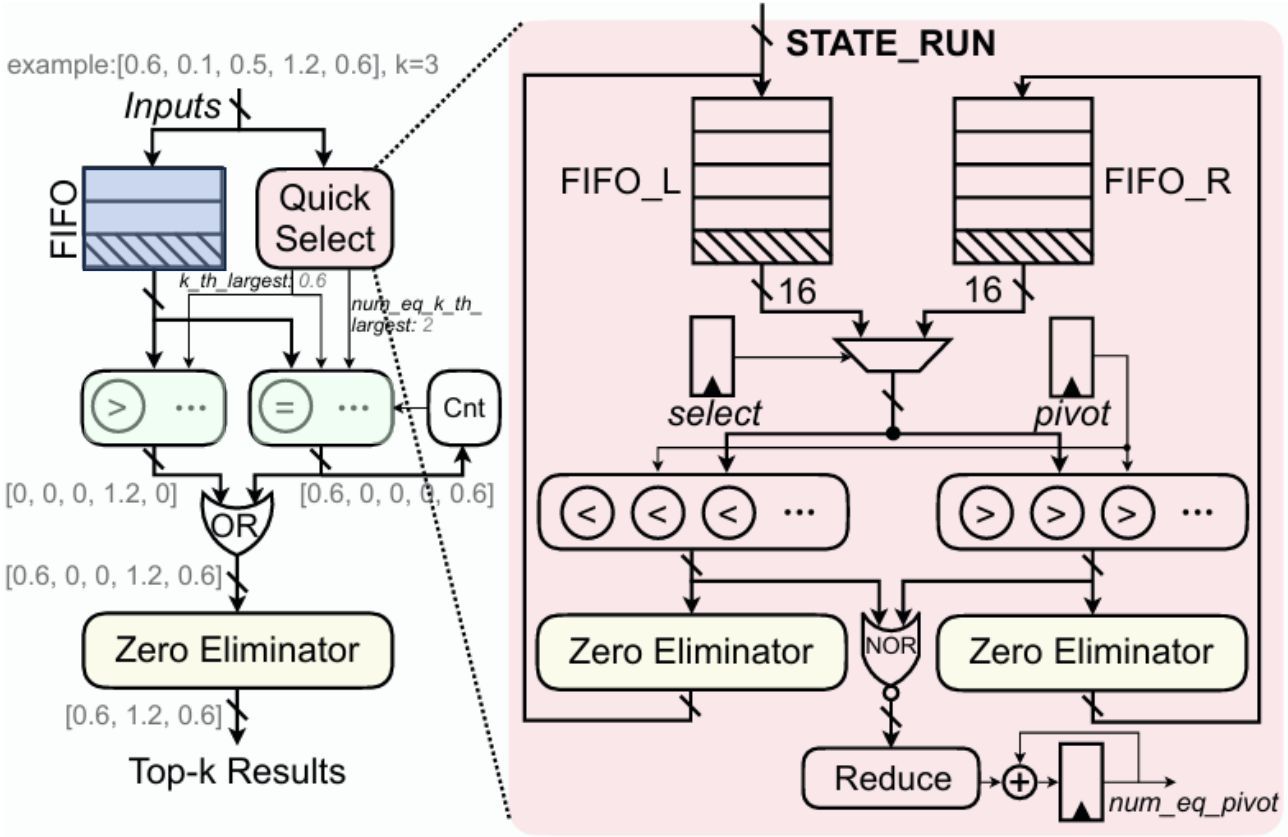
FIFO_R = [ ]

num_eq = 1

Termination condition: |FIFO_R| < r ≤ |FIFO_R| + num_eq →
0 < 1 ≤ 1

Therefore the threshold is θ = 0.5.

Top-k Engine



FIFO: [0.6, 0.1, 0.5, 1.2, 0.6], k = 3

FIFO_L: [0.6, 0.1, 0.5, 1.2, 0.6], k = 3
If, pivot = 0.6

FIFO_L: [0.1, 0.5], k = 3
| FIFO_R: [1.2] | + num_eq = 2 == k = 3
k_the_largest = 0.6

[0.6>0.6, 0.1>0.6, 0.5>0.6, 1.2>0.6, 0.6> 0.6]
= [0, 0, 0, 1.2, 0]

+

[0.6=0.6, 0.1=0.6, 0.5=0.6, 1.2=0.6, 0.6= 0.6]
= [0.6, 0, 0, 0, 0.6]

=

[0.6, 0, 0, 1.2, 0.6]
[0.6, 1.2, 0.6]

Query-Key Multiplication Module

Purpose
$S = QK^T$ produces attention scores

$$s_i = \Sigma_j K_{ij} Q_j$$

Key SRAM supplies 512 key elements per cycle.

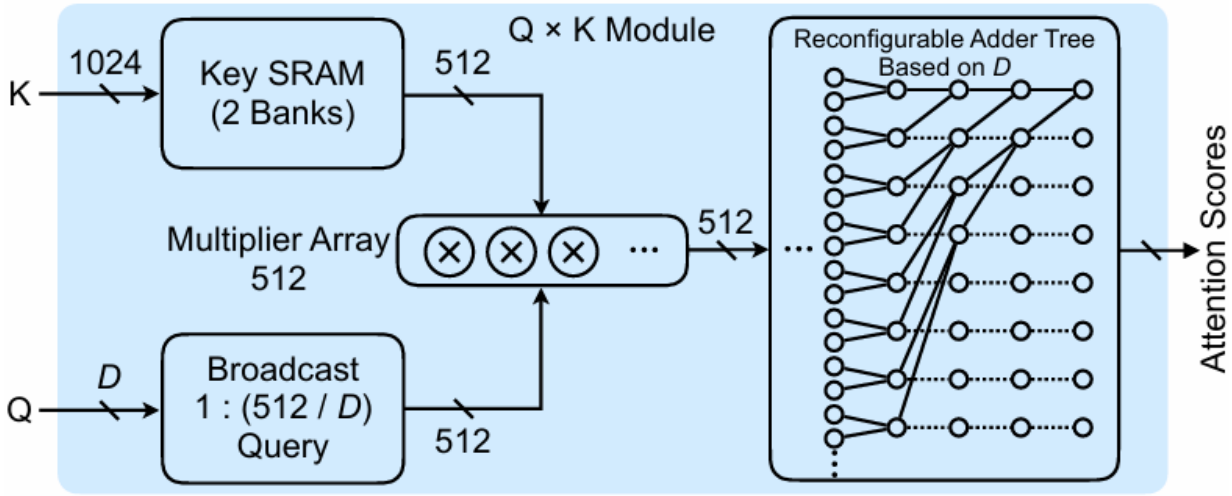
The query vector is broadcast to multiple multiplier groups.

512 multipliers compute element-wise products in parallel.

A reconfigurable adder tree reduces products into attention scores.

If head dimension is D, scores per cycle = 512 / D

$$Nscore = 512 / D$$



Softmax and Progressive Quantization

Converts fixed-point scores into probabilities and decides whether low bits are needed.

1. Dequantize + Normalize

$$\hat{s}_i = \alpha \cdot s_i^{int} / \sqrt{D}$$

2. Softmax

$$p_i = \exp(\hat{s}_i) / \Sigma_j \exp(\hat{s}_j)$$

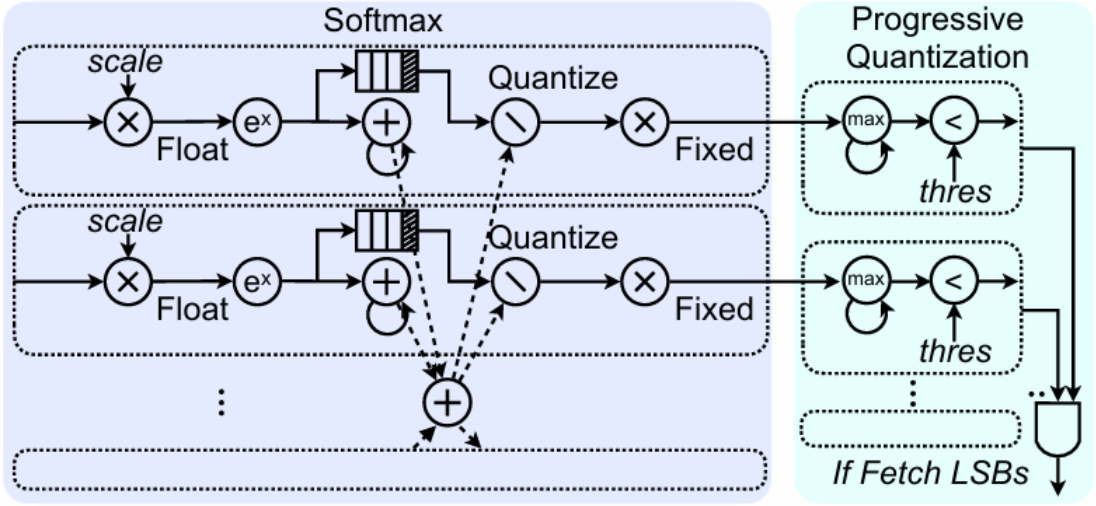
3. Precision Decision

$$pmax = \max_i p_i$$

Peaked distribution: MSB is enough.

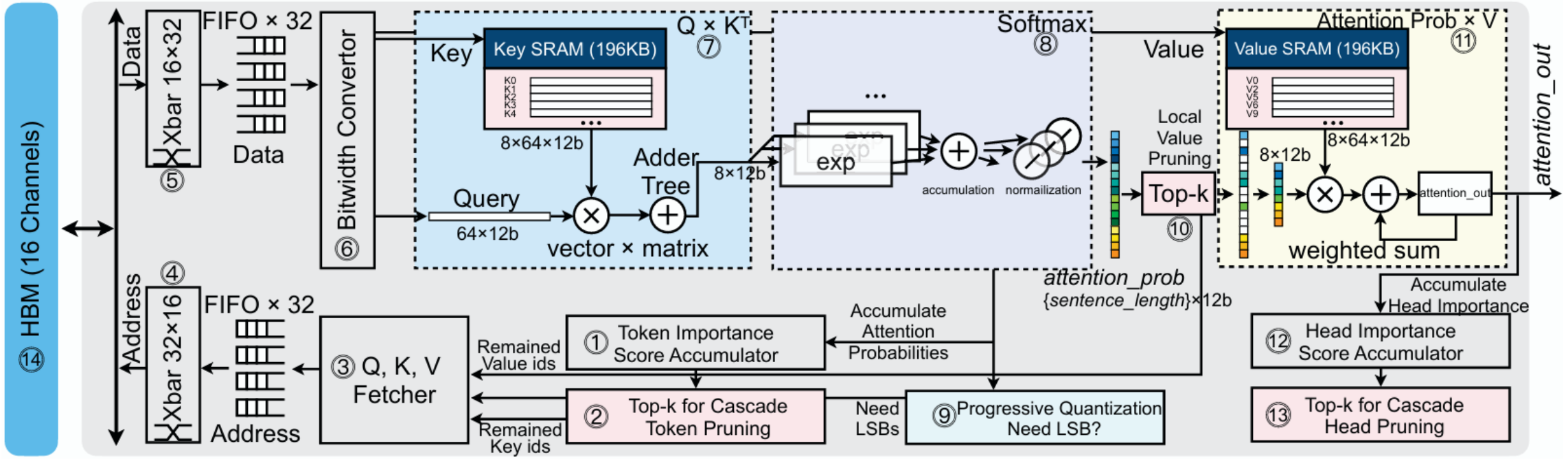
Flat distribution: fetch LSBs and recompute.

This reduces average memory traffic for GPT-style generation.



Decision rule: fetch LSBs only when pmax is below threshold τ

How One Query Flows Through SpAtten



Example Query
current token = "more"

Goal
compute attention_out for this Query

Main benefit
fetch and compute only important K/V data

How One Query Flows Through SpAtten

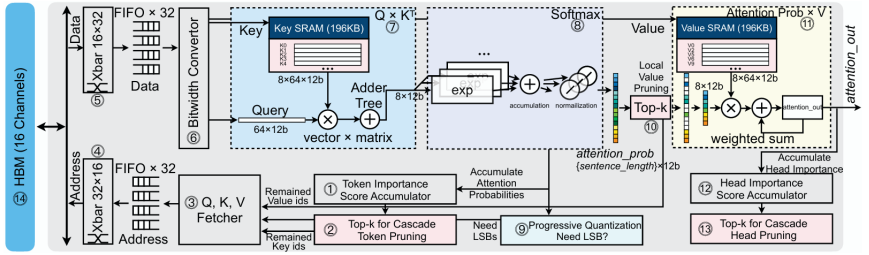
Input Query

Current Query token
"more"

Query vector for one head
Q_more = [1.0, 2.0]

Token	Prev. importance
I	0.4
bet	1.0
the	0.3
video	1.2
game	1.7
is	1.0
a	0.4
lot	1.8
more	0.6
fun	1.9
than	1.4
the	0.3
film	0.9
.	0.4

SpAtten keeps cumulative token importance from previous queries, heads, and layers.
Before fetching all Keys, it first selects high-importance Key IDs. This reduces unnecessary HBM traffic.



For this example, choose Top-6 Key IDs from the previous importance scores.

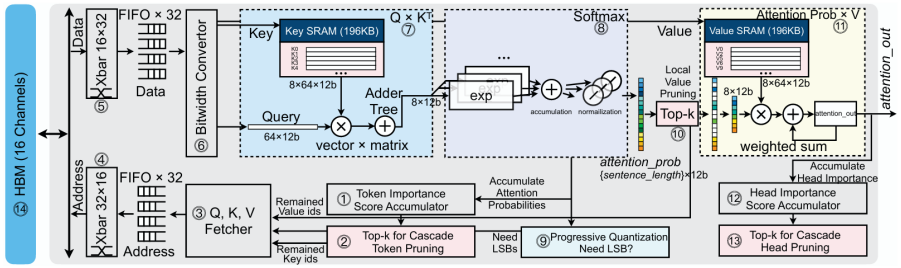
How One Query Flows Through SpAtten

Modules 1–2

Top-k input
Cumulative token scores

Top-k output
Key IDs to fetch

Rank	Selected Key	Score
1	fun	1.9
2	lot	1.8
3	game	1.7
4	than	1.4
5	video	1.2
6	is	1.0



Important: this step selects Key IDs, not the Key vectors themselves.

How One Query Flows Through SpAtten

Modules 3–6

Selected Key IDs
fun, lot, game, than, video, is

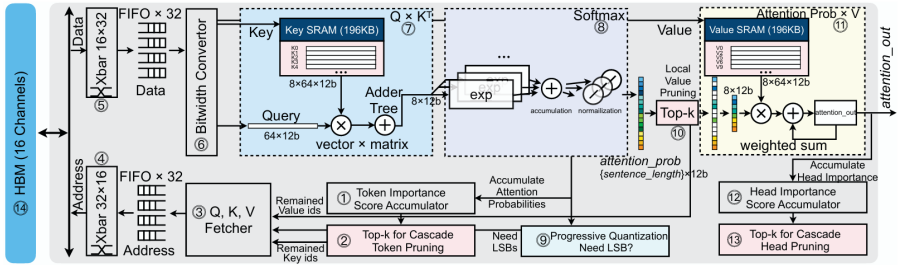
Address mapping
Key ID → HBM address

HBM Fetch selected K only

Key	HBM channel	Action
fun	Ch. 5	Fetch
lot	Ch. 9	Fetch
game	Ch. 11	Fetch
than	Ch. 2	Fetch
video	Ch. 2	Fetch
is	Ch. 7	Fetch
low-score tokens	-	Skip

MSB first
Fetch high bits only

Bitwidth Converter
internal 12-bit format



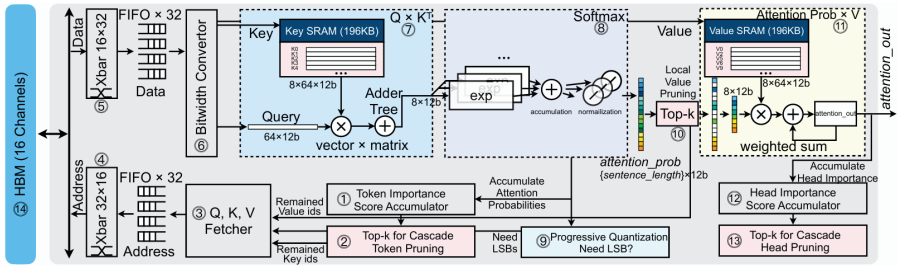
Crossbars distribute irregular memory requests across 16 HBM channels and restore the returned data order.

How One Query Flows Through SpAtten

Module 7

Formula: $S_j = (Q \cdot K_j) / \sqrt{D}$ with D = 2

Key	K_j	Dot product	Score
fun	[1.5, 1.2]	3.90	2.76
than	[1.2, 1.1]	3.50	2.47
lot	[1.0, 0.75]	2.50	1.77
game	[0.8, 0.6]	2.00	1.41
video	[0.5, 0.25]	1.00	0.71
is	[0.3, 0.1]	0.50	0.35



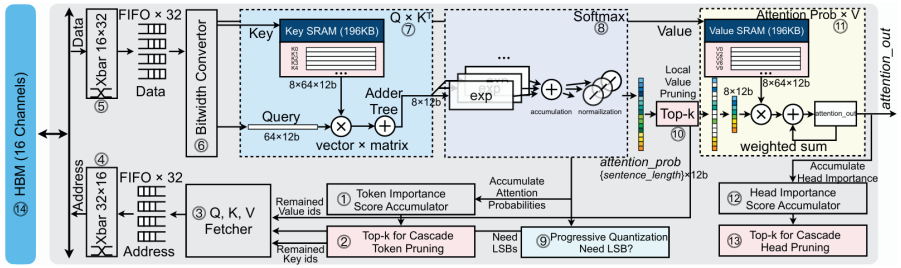
Score vector = [2.76, 2.47, 1.77, 1.41, 0.71, 0.35]

How One Query Flows Through SpAtten

Module 1

Update rule: $s_t[j] \leftarrow s_t[j] + P_j$

Token	Previous	Current P_j	New score
fun	1.90	0.384	2.284
than	1.40	0.289	1.689
lot	1.80	0.143	1.943
game	1.70	0.100	1.800
video	1.20	0.049	1.249
is	1.00	0.035	1.035



These updated scores are used by future Query/head/layer pruning decisions, not by the current Q × K calculation.

How One Query Flows Through SpAtten

Modules 8–9

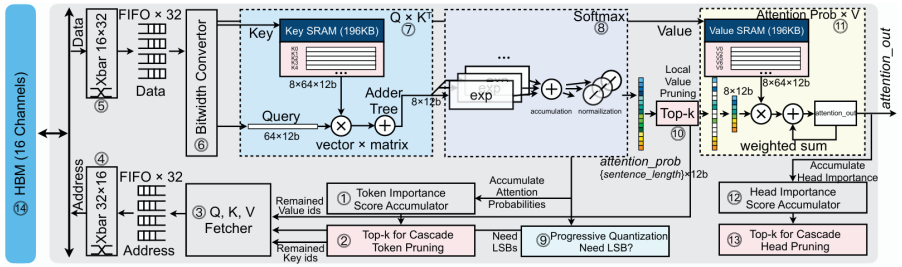
Softmax(score) = [0.38, 0.29, 0.14, 0.10, 0.05, 0.03]

Key	Probability
fun	0.384
than	0.289
lot	0.143
game	0.100
video	0.049
is	0.035

p_max = 0.385

threshold τ = 0.30

Decision: p_max ≥ τ
No LSB fetch



If the distribution were flat, SpAtten would fetch LSBs, concatenate MSB+LSB, and recompute Q × K^T and Softmax.

How One Query Flows Through SpAtten

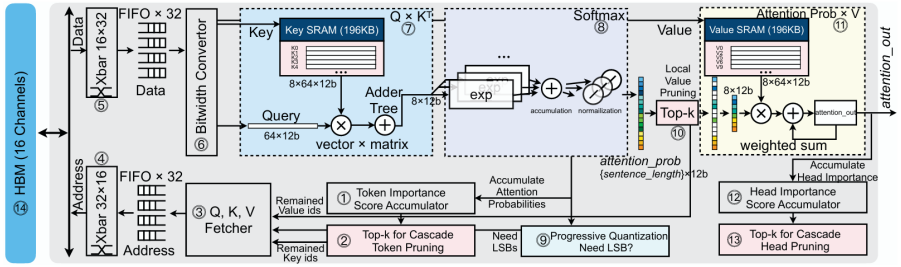
Module 10

Input to Local Value Top-k: current attention probabilities

Rank	Selected Value	Probability
1	V_fun	0.385
2	V_than	0.288
3	V_lot	0.143

Skipped Values V_game, V_video, V_is

Fetch only V_fun, V_than, V_lot



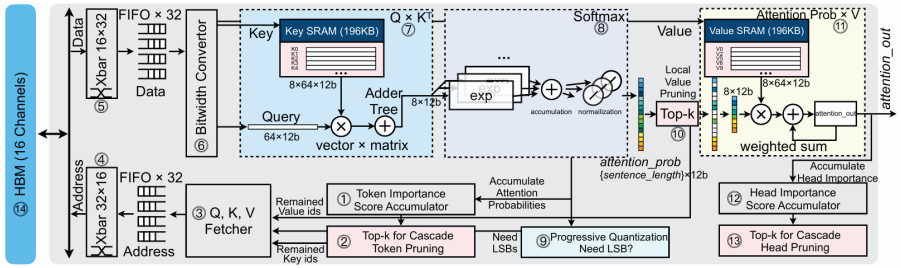
Local Value Pruning is per Query/head. It reduces V memory traffic and weighted-sum computation.

How One Query Flows Through SpAtten

Module 11

Formula: E_more = Σ P_j V_j over selected Value IDs

E_more = 0.385·V_fun + 0.288·V_than + 0.143·V_lot
V_fun=[1.0,0.8], V_than=[0.5,1.2], V_lot=[0.9,0.4]



attention_out for Query "more" = [0.66, 0.71]

This output is the final attention result for one Query in one head.
The same pipeline repeats for the next Query.

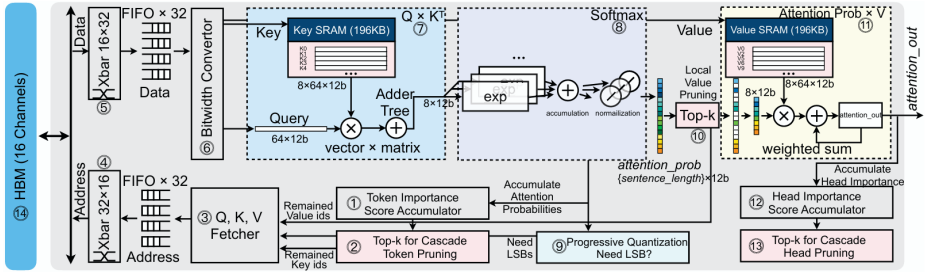
How One Query Flows Through SpAtten

Modules 12–13

Head importance update: $s_h[m] \leftarrow s_h[m] + \Sigma_d |E_{\{m,Q,d\}}|$

For this Query/head
Σ|attention_out| = 1.37

Accumulated over all Queries
then used for Head Pruning

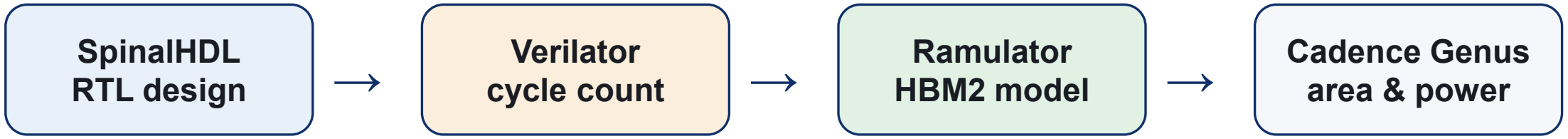


Layer-level decision
After all Queries are processed, Top-k for Cascade Head Pruning keeps high-importance heads and prunes low-importance heads.

Token pruning is updated during attention.
Head pruning is decided after accumulating head importance across the layer.

Evaluation

Evaluation



Benchmarks

BERT-Base / BERT-Large
GLUE + SQuAD

GPT-2 Small / Medium
WikiText, PTB, One Billion Word

Baselines

TITAN Xp GPU
Jetson Nano GPU
Xeon CPU
Raspberry Pi ARM CPU
A^3 and MNNFast

Accuracy control

Pruning ratios and quantization bitwidths are tuned per task.
Fine-tuning is applied after pruning.

Evaluation

TABLE III
COMPARE SPATTEN$_{1/8}$ WITH PRIOR ART A^3 AND MNNFAST.

	MNNFast	A^3	**SpAtten$_{1/8}$**
Cascade Head Pruning	✗	✗	✓
Cascade Token Pruning	✗	✗	✓
Interpretable Pruning	✗	✗	✓
Local Value Pruning	✓	✓	✓
Progressive Quantization	✗	✗	✓
Preprocessing Overhead	✗	✓	✗
Reduce Computation of	Attention only	Attention only	Attention and FFN
Accelerate	BERT only	BERT only	BERT & GPT-2
Technology	FPGA (28nm)	ASIC (40nm)	ASIC (40nm)
Frequency	1GHz (projected)	1GHz	1GHz
Area (mm^2)	-	2.08 mm^2	1.55 mm^2
Throughput (GOP/s)	120 (1×)	221 (1.8×)	360 (3.0×)
Energy Effi. (GOP/j)	120 (1×)	269 (2.2×)	382 (3.2×)
Area Effi. (GOP/s/mm^2)	-	106 (1×)	238 (2.2×)

What Table III shows

Only SpAtten supports cascade token pruning and cascade head pruning.

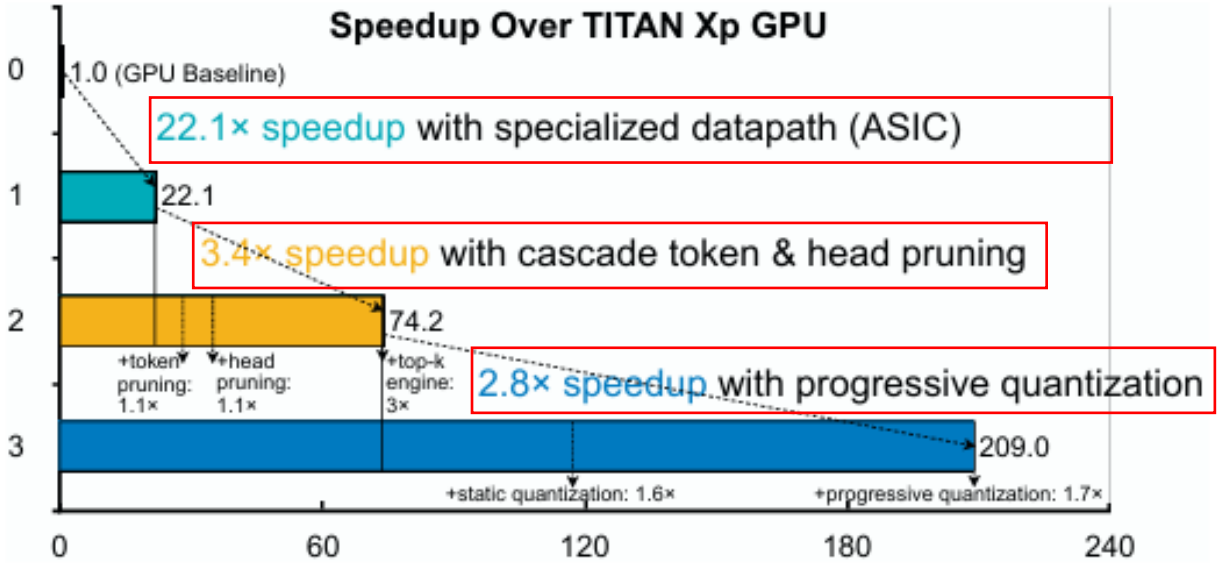
Only SpAtten combines interpretable pruning with progressive quantization.

SpAtten1/8 reaches 360 GOP/s throughput and 382 GOP/J energy efficiency.

SpAtten accelerates both BERT and GPT-2, while prior designs target BERT only.

Key idea
SpAtten wins through algorithm–hardware co-design, not only more compute units.

Evaluation



Breakdown interpretation

Dedicated attention datapath gives the first large jump: 22.1×.

Cascade pruning reduces work, but Top-k itself can become the bottleneck.

A high-throughput Top-k engine removes that bottleneck and adds a 3× jump.

Progressive quantization further reduces average input bitwidth and DRAM traffic.

Final message
Specialized datapath + fast Top-k + adaptive precision are all necessary.

Pruning alone is not enough: without fast Top-k selection, pruning decisions can become the new bottleneck.

Evaluation

Computation
2.1× average reduction
Token/head pruning removes unnecessary attention work.

DRAM Access
10× average reduction
Pruning and quantization avoid unnecessary Q/K/V fetches.

BERT Throughput
1.61 TFLOPS
BERT is closer to computation-bound execution.

GPT-2 Throughput
0.43 TFLOPS
GPT-2 generation is memory-bound, so reducing K/V traffic matters.

Evaluation takeaway: SpAtten is effective because it reduces both arithmetic and off-chip memory movement.

Q&A