# FACILITY LOCATION SUBMODULAR OPTIMIZATION ON GPU FOR TOKEN SUBSET SELECTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Token subset selection for vision-language model and in-context learning is often modeled as facility location maximization: choose $k$ of $n$ source tokens that best cover $m$ target tokens under a similarity matrix. Its objective function is monotone submodular, so the greedy is near-optimal. However, every existing solver is slow. CPU libraries submodlib and apricot run the lazy greedy as a serial chain of row reads. Moreover, PyTorch loops recompute all marginal gains on the GPU at every iteration, moving the whole similarity matrix through HBM $k$ times. To improve the execution speed, we present FLASH GREEDY, an IO-aware persistent CUDA kernel that runs the entire selection in one launch. The similarity matrix is tiled over the streaming multiprocessors (SM) and placed in SRAM, L2 cache or HBM by its size. Every SM keeps a private copy of the coverage vector and evaluates exactly one candidate token per iteration. In addition, a replicated selection loop admits several tokens per grid barrier. A bool position vector is used to certify which candidate gains are still exact, thereby selecting the multiple tokens. Most importantly, our method returns exactly the naive greedy solution, and we give its IO complexity per placement regime. In the evaluation, we use three similarity matrices from MMTok, SCOPE, and Selective Annotation; whose token features are extracted from image, video, and text. FLASH GREEDY returns the same subset as the submodlib, apricot and original PyTorch implementation. However, FLASH GREEDY is $9\times$ faster than the PyTorch loops, and $26\times$ faster than the best CPU library.

## 1 INTRODUCTION

Selecting a small, representative subset of tokens has become a standard lever for efficient inference. Vision-language models drop $80\%$ of their visual tokens before the LLM, such as SCOPE (Deng et al., 2025), CDPruner (Zhang et al., 2025a), MMTok (Dong et al., 2026), OTPrune (Chen et al., 2026) and FLoC (Cho et al., 2026). Moreover, in-context learning pipelines choose some unlabeled examples to annotate (SU et al., 2023). Many of the strongest selectors model the objective as the *facility location (FL) maximization*: given a similarity matrix $\mathbf{s} \in \mathbb{R}^{n\times m}$ between $n$ source tokens and $m$ target tokens, find the subset $S$ of $k$ source tokens that maximizes $f(S) = \sum_{j=1}^{m} \max_{i\in S} \mathbf{s}_{ij}$. This objective function is monotone submodular, so greedy algorithm can guarantee a $(1 - 1/e)$ approximation of the optimal solution (Nemhauser et al., 1978). Since the greedy algorithm is easy to implement, MMTok, SCOPE and Selective Annotation (SU et al., 2023) use the facility location function as coverage objectives.

The bottleneck is not the objective but its greedy solver. There are three common implementations: apricot (Schreiber et al., 2020) (Python, CPU) and submodlib (Kaushal et al., 2022) (C++, CPU) with naive greedy and lazy greedy (Mirzasoleiman et al., 2015); and the PyTorch loops of the token-pruning papers, which run naive greedy on the GPU. Naive greedy recomputes the marginal gains of all tokens at every iteration. Therefore, PyTorch loops move the whole matrix through GPU memory $k$ times and launch a dozen kernels per selected token. By contrast, lazy greedy reads only a few rows per iteration but is a serial chain of dependent operations. Thus, lazy greedy cannot fill the memory pipeline of a GPU and is confined to the CPU in both libraries. With $1{,}296$ tokens per image in Qwen2.5-VL (Bai et al., 2025), $2{,}880$ in LLaVA-NeXT (Li et al., 2024) and $10{,}816$ for $64$ video frames, the selection takes tens to hundreds of milliseconds on the GPU and seconds on CPU. In short, previous greedy implementations suffered from insufficient parallelism or IO bottleneck.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We build the solver the objective deserves, following the IO-aware philosophy of FlashAttention (Dao et al., 2022), Flash-KMeans (Yang et al., 2026), FLASH-MAXSIM (Pony et al., 2026) and TileMaxSim (Sharma, 2026). These works inspired us to decide where every byte lives, read every row once, never write a temporary. Our proposed FLASH GREEDY replaces the priority queue of lazy greedy by $T$ gain tiles, one tile per streaming multiprocessor (SM). By the parallel operations, every SM evaluates exactly one candidate and several candidates are admitted per synchronization. Flash Greedy is the first CUDA kernel for facility location maximization, and our contributions are:

- **IO-aware Persistent Kernel.** The whole selection is one cooperative launch with one thread block per SM. Moreover, the similarity matrix is placed in SRAM, L2 cache or HBM by its size and read from HBM once. Every SM keeps a private copy of the coverage vector, and reads the row of its single candidate with all its warps. Besides, each SM exchanges a 12-byte record and an $m$-bit bool vector through L2 cache. This kernel takes only the similarity matrix and budget $k$.
- **Exact Multi-token Admission.** The bool position vector certifies that admitting one candidate has not changed the gain of another. A simple bitwise operation makes it possible to determine whether a candidate affects the marginal gain of another candidate. Thereby, several tokens are admitted per barrier while the output stays the exact naive greedy solution. We also give the IO complexity of three tiles placement regimes.
- **Faster Execution Time.** We use the similarity matrix data from MMTok, SCOPE, and Selective Annotation. Five solvers are compared: submodlib lazy greedy (CPU, C++), apricot naive and lazy greedy (CPU, Python), the original PyTorch loops on GPU, and FLASH GREEDY. All five solvers return the same subsets, and evaluation results are averaged over the samples of each dataset. FLASH GREEDY is $9\times$ faster than the original PyTorch implementation and $26\times$ faster than the best CPU library.

## 2 RELATED WORK

### 2.1 SUBMODULAR MAXIMIZATION AND GREEDY ALGORITHM

Facility location was introduced for plant location (Cornuejols et al., 1977) and popularized for data summarization by Lin & Bilmes (2011). The $(1 - 1/e)$ greedy bound is due to Nemhauser et al. (1978) and Krause & Golovin (2014) survey this field. Lazy greedy (Mirzasoleiman et al., 2015) re-evaluates only the top of a priority queue. Threshold greedy (Badanidiyuru & Vondrák, 2014) and low-adaptivity algorithms (Balkanski & Singer, 2018; Breuer et al., 2020) trade exactness for fewer evaluations or rounds. submodlib (Kaushal et al., 2022) and apricot (Schreiber et al., 2020) implement the naive greedy and lazy greedy on the CPU. The GPU work on submodular maximization targets distributed or streaming settings (Mirzasoleiman et al., 2016), and none keeps the dependent greedy chain on GPU chip.

### 2.2 TOKEN PRUNING AND SELECTION

MMTok (Dong et al., 2026) maximizes facility location over a stacked text–vision and vision–vision similarity matrix with the PyTorch greedy loops. SCOPE (Deng et al., 2025) combines the facility location coverage gain with a saliency weight on CLIP token similarity. FLoC (Cho et al., 2026), CoverPruner (Zhu et al., 2026) and EADP (Wang et al., 2026) run the weighted FL greedy on images and videos. CDPruner (Zhang et al., 2025a) and OTPrune (Chen et al., 2026) use the other diversity objectives. Selective Annotation (SU et al., 2023) chooses the instances for in-context learning on a Sentence-BERT (Reimers & Gurevych, 2019) similarity graph with the facility-location baseline.

### 2.3 IO-AWARE KERNEL

FlashAttention (Dao et al., 2022) showed that hierarchy-aware tiling and fusion, not FLOPs, decide the speed of memory-bound primitives. Flash-KMeans (Yang et al., 2026) fuses the distances with an online argmin so that the distance matrix never reaches HBM. FLASH-MAXSIM (Pony et al., 2026) and TileMaxSim (Sharma, 2026) fuse the late-interaction MaxSim score (Khattab & Zaharia, 2020) with a running maximum over document tokens. We borrow their playbook and add what a sequential greedy needs: a persistent kernel (Osama et al., 2023; Zhang et al., 2023) with a grid barrier (Zhang et al., 2020) per iteration, replicated state, and placement in GPU memory hierarchy.

## 3 BACKGROUND

**Problem.** There are $n$ source tokens and $m$ target tokens with extracted features from a large model. A similarity matrix $\mathbf{s} \in \mathbb{R}^{n \times m}$ holds the similarity $\mathbf{s}_{ij}$ between source token $i$ and target token $j$. Ground set $V = \{i \mid 1 \leq i \leq n\}$ is the all source token indices. The goal is to select a subset $S \subset V$ with $k$ tokens that summarizes the $n$ source tokens.

**Definition 1** (Submodular Function)**.** Let $f : 2^V \to \mathbb{R}$ be a set function. For the current set $S \subset V$, $\Delta(i \mid S) = f(S \cup \{i\}) - f(S)$ is the *marginal gain* of token $i$ with $i \in V$ and $i \notin S$. Set function $f$ is *submodular* if $\Delta(i \mid A) \geq \Delta(i \mid B)$ for all set $A$ and set $B$ with $A \subseteq B \subset V$. Set function $f$ is *monotone* if $\Delta(i \mid S) \geq 0$ for all token $i$.

**Definition 2** (Facility Location Function)**.** Given a similarity matrix $\mathbf{s}$, the facility location function is $f(S) = \sum_{j=1}^{m} \max_{i \in S} \mathbf{s}_{ij}$. For each target token $j$, we compute the representative from subset $S$ which is closest to token $j$ and add these similarities for all target tokens. Facility location function $f$ is monotone and submodular, because the marginal gain $\Delta(i \mid S) \geq 0$ for all $i \in V$ and $S \subset V$.

Maximizing a monotone submodular function under $|S| = k$ is NP-hard (Cornuejols et al., 1977). But greedy algorithm repeatedly adds the token with the largest marginal gain, returns subset $S$ with $f(S) \geq (1 - 1/e) \max_{|A|=k} f(A)$ (Nemhauser et al., 1978), the best possible ratio (Feige, 1998). Generally, we find the token $v = \arg\max_{1 \leq i \leq n} \sum_{j=1}^{m} \mathbf{s}_{ij}$, and initialize the *memoized coverage vector* $\mathbf{y} \in \mathbb{R}^m$ with $\mathbf{y} = \mathbf{s}_v$. For selected token $v$ at one iteration, $\mathbf{y}$ is updated via $\mathbf{y} = \max(\mathbf{y}, \mathbf{s}_v)$. Therefore, we can conveniently compute the function value $f(S) = \sum_{j=1}^{m} \mathbf{y}_j$ and the marginal gain $\Delta(i \mid S) = \sum_{j=1}^{m} \max(\mathbf{s}_{ij} - \mathbf{y}_j, 0)$.

**Naive Greedy** repeats the following steps $k$ times:

- Recompute the marginal gain $\Delta(i \mid S)$ of every unselected token.
- Select the token $v = \arg\max_{1 \leq i \leq n} \Delta(i \mid S)$.
- Add token $v$ to current subset $S$ and update $\mathbf{y} = \max(\mathbf{y}, \mathbf{s}_v)$.

Naive Greedy loads the whole similarity matrix $\mathbf{s}$ a total of $k$ times, so there are $O(knm)$ operations for token selection.

**Lazy Greedy** (Algorithm 1) exploits the submodularity: a gain computed for an earlier set is an *upper bound* of the current one. It selects the token with the largest row sum first, then initializes the $\mathbf{y}$ and the gain vector $\delta$ with one pass over $\mathbf{s}$. After initialization, it keeps the stale gains in a priority queue $Q$. During the iteration, Lazy Greedy pops the token $v$ with the largest stale gain, and recomputes its exact gain $\delta_v$ with one row read. Algorithm 1 admits the token $v$ if $\delta_v$ is at least the largest stale gain $\delta_u$ left in $Q$ (a bound on every other exact gain), and otherwise re-inserts $(\delta_v, v)$.

**Algorithm 1** Lazy Greedy Algorithm for Facility Location Function Maximization

**Require:** Token similarity matrix $\mathbf{s} \in \mathbb{R}^{n \times m}$, selected subset budget $k$.
1: Find the token $v = \arg\max_{1 \leq i \leq n} \sum_{j=1}^{m} \mathbf{s}_{ij}$, and initialize the token subset $S = \{v\}$.
2: Initialize the memoized coverage vector $\mathbf{y} \in \mathbb{R}^m$ with $\mathbf{y} = \mathbf{s}_v$.
3: Initialize the marginal gain vector $\delta \in \mathbb{R}^n$ with $\delta_i = \sum_{j=1}^{m} \max(\mathbf{s}_{ij} - \mathbf{y}_j, 0)$.
4: Insert the element $(\delta_i, i)$ into a priority queue $Q$ for all $i \in [1, n]$ and $i \neq v$.
5: **while** $|S| < k$ **do**
6: Pop the element with maximum marginal gain: $(\delta_v, v) = Q.pop()$.
7: Recalculate the marginal gain of token $v$ with $\delta_v = \sum_{j=1}^{m} \max(\mathbf{s}_{vj} - \mathbf{y}_j, 0)$.
8: Get the element with maximum marginal gain: $(\delta_u, u) = Q.top()$.
9: **if** $\delta_v \geq \delta_u$ **then** ▷ token $v$ has a greater real gain than token $u$
10: Add the selected token $v$ to the subset: $S = S \cup \{v\}$.
11: Update the memoized coverage vector: $\mathbf{y} = \max(\mathbf{y}, \mathbf{s}_v)$.
12: **else** ▷ token $v$ cannot be selected now
13: Re-insert the element $(\delta_v, v)$ into the priority queue $Q$ for updating the marginal gain.
14: **end if**
15: **end while**
16: **return** $S$

**Algorithm 2** Flash Greedy Algorithm for Facility Location Function Maximization

**Require:** Token similarity matrix $\mathbf{s} \in \mathbb{R}^{n \times m}$, selected subset budget $k$.

1: Find the token $v = \arg\max_{1 \le i \le n} \sum_{j=1}^{m} \mathbf{s}_{ij}$, and initialize the subset $S = \{v\}$ in the L2 cache.
2: Initialize the memoized coverage vector $\mathbf{y} \in \mathbb{R}^m$ in the L2 cache with $\mathbf{y} = \mathbf{s}_v$.
3: Initialize the marginal gain vector $\mathbf{x} \in \mathbb{R}^n$ in the L2 cache with $\mathbf{x}_i = \sum_{j=1}^{m} \max(\mathbf{s}_{ij} - \mathbf{y}_j, 0)$.
4: Let $T$ be the number of SM in the GPU, $w = \lceil \frac{n}{T} \rceil$ be the tile size; and set $\mathbf{x}_v = -1$ to mark.
5: Split $\mathbf{s}$ and $\mathbf{x}$ into $T$ similarity tiles $\{\mathbf{s}^1, \ldots, \mathbf{s}^T\}$ and $T$ gain tiles $\{\mathbf{x}^1, \ldots, \mathbf{x}^T\}$ of size $w$ each.
6: If $T$ SMs can store the $T$ similarity tiles, we store them into the SRAM of $T$ SMs respectively.
7: Otherwise, if L2 cache can store $T$ similarity tiles, we store them into L2 cache; else, into HBM.
8: **while** $|S| < k$ **do**
9:  **for** $t \in [1, T]$ in parallel **do** ▷ parallel processing $T$ tiles in the $T$ SMs
10:  Find the candidate token with maximum marginal gain: $\mathbf{c}^t = \arg\max_{1 \le i \le w} \mathbf{x}_i^t$.
11:  Initialize the updated marginal gain $\mathbf{g}^t = \min(\mathbf{x}_{\mathbf{c}^t}^t, 0)$; then set $\mathbf{x}_{\mathbf{c}^t}^t = -1$ to mark.
12:  Find the upper-bound marginal gain in the $t$-th tile: $\mathbf{u}^t = \max_{1 \le i \le w} \mathbf{x}_i^t$.
13:  Recalculate the tile marginal gain with $\mathbf{z}^t = \max(\mathbf{s}_{\mathbf{c}^t}^t - \mathbf{y}, \mathbf{0})$ and $\mathbf{g}^t = \mathbf{g}^t + \sum_{j=1}^{m} \mathbf{z}_j^t$.
14:  Convert the marginal gain vector $\mathbf{z}^t$ into the bool vector; now $\mathbf{z}^t$ is an $m$-bit bool vector.
15:  **end for**
16:  Get the maximum upper-bound gain $\mathbf{u}^* = \max_{1 \le t \le T} \mathbf{u}^t$; when $\mathbf{u}^* = -1$, we set $\mathbf{u}^* = 0$.
17:  Initialize the bool position vector $\mathbf{b}$ in the L2 cache with value False and size $m$.
18:  **while** $(\max_{1 \le t \le T} \mathbf{g}^t) \ge \mathbf{u}^*$ and $|S| < k$ **do** ▷ multiple tokens may be selected
19:  Find the candidate tile with maximum marginal gain: $a = \arg\max_{1 \le t \le T} \mathbf{g}^t$.
20:  **if** bool vector $\mathbf{b}$ & $\mathbf{z}^a$ is all False **then** ▷ target token positions do not overlap
21:  Set $\mathbf{g}^a = -1$ to mark, and add the token $v = (a-1) \times w + \mathbf{c}^a$ to subset: $S = S \cup \{v\}$.
22:  Update the memoized coverage vector: $\mathbf{y} = \max(\mathbf{y}, \mathbf{s}_{\mathbf{c}^a}^a)$.
23:  Update the bool position vector: $\mathbf{b} = \mathbf{b} \mid \mathbf{z}^a$.
24:  **else** ▷ target token positions do overlap
25:  **break** ▷ stop this inner loop
26:  **end if**
27:  **end while**
28:  **for** $t \in [1, T]$ in parallel **do** ▷ parallel updating $T$ gain tiles in the $T$ SMs
29:  Update the marginal gain tile: $\mathbf{x}_{\mathbf{c}^t}^t = \mathbf{g}^t$.
30:  **end for**
31: **end while**
32: **return** $S$

**Drawbacks on the GPU.** Computing a marginal gain is *memory-bound* (3 operations per 4 bytes read), so its time is the time to move the row from GPU HBM. Naive Greedy moves the entire similarity matrix through the slowest HBM memory $k$ times. In addition, the PyTorch loops launch many kernels per iteration to write $n \times m$ temporaries. Lazy Greedy moves the little data but is a *latency-bound* dependent chain, whose single row cannot keep the memory pipeline of a GPU busy. Moreover, its priority queue confines it to the CPU. Neither exploits that the similarity matrix fits on the GPU chip.

# 4 METHOD

## 4.1 FLASH GREEDY

Our improved Flash Greedy is shown in Algorithm 2, which is divided into four phases:

**Phase 1: Initialization and Tiling (lines 1–7).** As in Lazy Greedy, the token with the largest row sum is selected first, and every other token receives its exact gain with respect to $\{v\}$. $T$ is the number of SM (108 on A100 GPU). The rows of $\mathbf{s}$ and the entries of $\mathbf{x}$ are cut into $T$ contiguous tiles of $w = \lceil \frac{n}{T} \rceil$ rows, one per SM. Moreover, $T$ similarity tiles are placed by their *data size*: a tile ($4wm$ bytes) stays in the SRAM of its SM when it fits; otherwise the all similarity tiles stay in the L2 cache (40 MB on A100 GPU) when it fits, otherwise in the HBM. This is because the bandwidth of SRAM (19 TB/s) is greater than that of L2 cache (5 TB/s). And the bandwidth of L2 cache is greater than that of HBM (1.56 TB/s).

**Phase 2: Parallel Computing Gain (lines 9–15).** Each SM picks the token $\mathbf{c}^t$ with the largest cached gain of its tile and marks it. Then, SM finds the largest cached gain $\mathbf{u}^t$ among its remaining tokens. With one row read, SM recomputes the exact gain of its single candidate against the current $\mathbf{y}$. The positions where positive gain is produced are stored as a bool vector $\mathbf{z}^t$. Overall, each SM returns a bool vector $\mathbf{z}^t$ and a 12-byte record $(\mathbf{g}^t, \mathbf{c}^t, \mathbf{u}^t)$.

**Phase 3: Admit Multiple Tokens (lines 16–27).** By the submodularity, every cached gain is an upper bound of the exact one, so $\mathbf{u}^*$ bounds the exact gain of every non-candidate. Therefore, the candidate $a$ with the largest exact gain is the true greedy choice whenever $\mathbf{g}^a \geq \mathbf{u}^*$. Furthermore, candidate $a$ is admitted and $\mathbf{y}$ is updated by $\mathbf{s}^a_{\mathbf{c}^a}$. Admitting $a$ changes the other candidates' gains only on the target token positions where $\mathbf{z}^a$ is True. The bool position vector $\mathbf{b}$ accumulates these positive positions, and a candidate whose positive positions do not intersect $\mathbf{b}$ keeps exactly its gain. Thereby, this candidate can be admitted next without reading anything.

**Phase 4: Update Marginal Gain (lines 28–30).** Finally, the updated exact gains replace the cached stale ones. The marginal gains of already selected tokens are marked as $-1$.

### 4.2 Algorithm Analysis

In Algorithm 2, we refer to the first **while** loop as the *outer iteration* and the second **while** loop as the *inner loop*. Every outer iteration reads exactly one row per SM, so $T$ SMs read $T$ rows in the time of one row. Since the candidates are the tops of $T$ independent tiles, the largest candidate gain is usually far above the bound $\mathbf{u}^*$. In other words, the test of Algorithm 2 line 18 succeeds in most outer iterations. After $a$ is admitted, bool position vector will accumulate the non-overlapping positions. When the best candidate does overlap the bool position vector $\mathbf{b}$, Algorithm 2 stops this inner loop. For the overlap candidates, Algorithm 2 will re-read their rows in subsequent outer iterations. Due to the speed of bitwise operations, bool position vector $\mathbf{b}$ enables the inner loop to select multiple tokens with minimal computation. Most importantly, the admitted sequence by flash greedy is exactly the naive greedy sequence.

### 4.3 IO Complexity

The data size of `float32` similarity matrix is $4nm$ bytes. Flash Greedy moves the $4nm$ bytes through HBM in the SRAM regime or L2 regime. Let $r$ be the total number of outer iterations. In the HBM regime, Flash Greedy loads the $4Trm$ bytes through HBM for marginal gain computation (Algorithm 2 line 13). Moreover, Flash Greedy moves the $4km$ bytes through HBM for updating the memoized coverage vector (Algorithm 2 line 22). Initialization reads the similarity matrix $\mathbf{s}$ twice. In total, Flash Greedy moves the $8nm + 4m(Tr + k)$ bytes from HBM to GPU SMs in the HBM regime. The PyTorch loops read $\mathbf{s}$, write the temporary $\max(\mathbf{s} - \mathbf{y}, \mathbf{0})$, and read it back for the reduction at every iteration. In the HBM, PyTorch loops move $12nm$ bytes for selecting a token. Therefore, there is a total of $12knm$ bytes of read and write operations. By contrast, the HBM traffic of Flash Greedy is *independent of* $k$ in the SRAM or L2 regimes. In the HBM regime, $8nm + 4m(Tr + k)$ is much smaller than $12knm$. In short, Flash Greedy no longer faces the IO bottlenecks, but the execution speed of PyTorch loops is limited by HBM bandwidth.

### 4.4 Implementation Details

The whole Algorithm 2 is one program that keeps its $T$ workers alive until the last token is selected. These workers are launched in a mode in which the GPU guarantees that all of them run at the same time, which is what allows them to wait for each other. Each worker owns a fixed set of rows (its tile) and a small private memory on its processor that holds. Rows are read with several pieces requested at once per thread, so that a whole row is in flight before any of it is used. The records that workers publish for each other are written alternately to two buffers, one for even and one for odd steps. So that, a fast worker never overwrites a record that a slow worker has not yet read. The bool vector of a worker is read by the others only in the step after its candidate has been admitted. And the worker does not write it again before the next synchronization. So, its bit order is a fixed permutation of the target token indices, which is all the AND test needs. The kernel function takes only similarity matrix $\mathbf{s}$ and budget $k$. The number of workers $T$, tile width $w = \lceil \frac{n}{T} \rceil$, and the placement follow from the GPU properties and similarity matrix size.

# 5 EXPERIMENTS

## 5.1 BENCHMARKS

We adopt the similarity matrix datasets from MMTok (Dong et al., 2026), SCOPE (Deng et al., 2025), and Selective Annotation (SU et al., 2023). Each sample is one input of their pipelines.

**MMTok and SCOPE** are visual token pruning methods for image or video understanding tasks. We use the Qwen-2.5-VL-7B (Bai et al., 2025) and LLaVA-NeXT-7B (Liu et al., 2024) models to process TextVQA (Singh et al., 2019) and POPE (Li et al., 2023) images. LLaVA-NeXT generates $n$=2,880 vision tokens from multiple images at input resolution $672 \times 672$. Moreover, Qwen-2.5-VL generates $n$=1,296 vision tokens for $1008 \times 1008$ resolution image. We also utilize the larger inputs for scaling (5,184–9,216 tokens). For long video understanding, we use the LLaVA-Video-7B (Zhang et al., 2025b) to process the videos in LongVideoBench (Wu et al., 2024). With 64 frames per video, LLaVA-Video generates $n$=10,816 vision tokens.

**Selective Annotation** is an unsupervised method for in-context learning. We use six of its datasets (MRPC (Dolan et al., 2004), SST-5 (Socher et al., 2013), MNLI (Williams et al., 2018), DBpedia (Lehmann et al., 2015), RTE (Bentivogli et al., 2009), HellaSwag (Zellers et al., 2019)) with 3,000 instances each. They are embedded via the Sentence-BERT (Reimers & Gurevych, 2019) model. We also utilize the 6,000–20,000 instances for scaling.

In the tables, *Video* is used as shorthand for the LongVideoBench dataset, and *SelAnn* is used as shorthand for the Selective Annotation method.

## 5.2 BASELINES

We compare our method (FLASH GREEDY) with the following three baselines:

- The C++ implementation of Lazy Greedy in submodlib (Kaushal et al., 2022), running on CPU.
- The Python implementations of Naive Greedy and Lazy Greedy in apricot (Schreiber et al., 2020). It uses the Numba (Lam et al., 2015) to accelerate computation on the CPU.
- The original PyTorch implementations of MMTok, SCOPE, and Selective Annotation. They are all implementations of the Naive Greedy on GPU.

GPU methods run on the NVIDIA A100 (40GB HBM memory) with CUDA 12.8 and PyTorch 2.11. The A100 GPU has 108 SMs, 40 MB L2 cache, and 1.56 TB/s HBM. CPU methods run on the AMD EPYC 7B13 (64 cores, 128 threads). Given the `float32` similarity matrix in memory, evaluation metric is inference time of the selection. The reported time of a configuration is the average over all samples of its dataset. Selected subset budget $k$ follows the above papers with $128 \leq k \leq 2{,}048$.

## 5.3 RESULTS

**Overall Comparison.** Table 1 and Table 4 cover 73 configurations, and FLASH GREEDY is the fastest solver in all of them. Last column is the speedup over the fastest baseline of each row. Overall, FLASH GREEDY is 9× faster than the PyTorch loops, 26× faster than submodlib Lazy Greedy, 35× faster than apricot Lazy Greedy and 64× faster than apricot Naive Greedy. For example, it selects 320 of 2,880 LLaVA-NeXT tokens in 3.3 ms and 1,024 of 20,000 instances in 40 ms (baselines: 9.6–63 s). FLASH GREEDY returns the same subsets as the PyTorch loops and CPU libraries. However, CPU libraries do not support the `BF16` precision.

**Runtime Analysis.** submodlib and apricot run the Lazy Greedy as a serial chain: after a serial $O(nm)$ initialization, each pop reads one row before the next decision. So their time grows slowly with $k$ but quickly with $n$ (submodlib: 266 → 494 ms for $k$=320 → 1,280 at $n$=2,880, but 31 s at $n$=20,000). And the best CPU library is always at least 26.8× slower than FLASH GREEDY. Naive Greedy in apricot recomputes all gains on 128 threads, so it grows fastest with $k$ (665 ms→ 12 s). The PyTorch loops read $\mathbf{s}$, write $\max(\mathbf{s} - \mathbf{y}, \mathbf{0})$ and read it back per token ($12nm$ HBM bytes and 8–16 launches), so they are linear in $k$. For $n$=$m$=2,880 and $k$=320, this is 32 GB; in other words, at least 20 ms at the peak bandwidth 1.56 TB/s, versus 3.3 ms for FLASH GREEDY. For the largest matrices, PyTorch loops even lose to CPU Lazy Greedy (5.4 vs. 4.6 s for 10,816 video tokens).

Table 1: Inference time (ms unless marked) of five solvers, averaged over all samples of each dataset. For PyTorch loops and Flash Greedy, placement regime is based on the data size of similarity matrix. More 59 configurations are shown in Table 4 (Appendix A).

| Setting | Dataset | $n$ | $m$ | $k$ | regime | submodlib lazy | apricot naive | apricot lazy | PyTorch naive | **ours flash** | speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MMTok | TextVQA | 1,296 | 1,301 | 128 | SRAM | 54.6 | 76.5 | 58.4 | 10.7 | **1.17** | 9.14× |
| MMTok | TextVQA | 1,296 | 1,301 | 256 | SRAM | 65.8 | 186 | 72.2 | 21.4 | **1.80** | 11.8× |
| MMTok | TextVQA | 2,880 | 2,885 | 320 | L2 | 266 | 665 | 183 | 73.3 | **3.31** | 22.1× |
| MMTok | TextVQA | 2,880 | 2,885 | 1,280 | L2 | 494 | 12 s | 1.2 s | 292 | **8.23** | 35.4× |
| MMTok | POPE | 2,880 | 2,884 | 320 | L2 | 266 | 678 | 176 | 73.3 | **3.21** | 22.8× |
| SCOPE | TextVQA | 1,296 | 1,296 | 128 | SRAM | 52.3 | 90.6 | 53.4 | 29.6 | **1.18** | 25.0× |
| SCOPE | TextVQA | 2,880 | 2,880 | 320 | L2 | 306 | 671 | 197 | 169 | **3.39** | 49.8× |
| SCOPE | TextVQA | 5,184 | 5,184 | 1,024 | HBM | 1.5 s | 10.0 s | 2.0 s | 1.4 s | **10.5** | 133× |
| SCOPE | POPE | 1,296 | 1,296 | 512 | SRAM | 82.4 | 485 | 110 | 118 | **3.07** | 26.8× |
| SCOPE | Video | 10,816 | 10,816 | 1,024 | HBM | 5.7 s | 26 s | 4.6 s | 5.4 s | **25.1** | 183× |
| SCOPE | Video | 10,816 | 10,816 | 2,048 | HBM | 7.0 s | 63 s | 17 s | 11 s | **27.6** | 253× |
| SelAnn | MNLI | 3,000 | 3,000 | 128 | L2 | 471 | 214 | 267 | 39.3 | **2.33** | 16.8× |
| SelAnn | MNLI | 3,000 | 3,000 | 512 | L2 | 637 | 1.5 s | 435 | 157 | **5.09** | 30.8× |
| SelAnn | MNLI | 20,000 | 20,000 | 1,024 | HBM | 31 s | 63 s | 14 s | 9.6 s | **40.3** | 238× |

**Why FLASH GREEDY is Faster.** Algorithm 2 keeps the few row reads of Lazy Greedy but removes its serial chain. Each outer iteration reads $T$=108 candidate rows at once (line 13), i.e., 35 thousand rows in total for MMTok with $k$=320, against 868 thousand for Naive Greedy. These rows come from the fastest memory that holds them. In the SRAM regime ($n$=1,296), each SM keeps its tile of 12 rows (62 KB) in its 164 KB SRAM. In the L2 regime, the matrix (at most 36 MB) stays in the 40 MB L2 cache after one HBM pass, so the HBM traffic is independent of $k$. In the HBM regime, FLASH GREEDY moves $8nm + 4m(Tr + k)$ bytes, 224× less than the PyTorch loops for 1,024 of 10,816 video tokens, matching the measured 215× speedup. Hence, the speedup grows from 9–32× (SRAM) to 15–65× (L2) and 25–254× (HBM).

### 5.4 ABLATION STUDIES

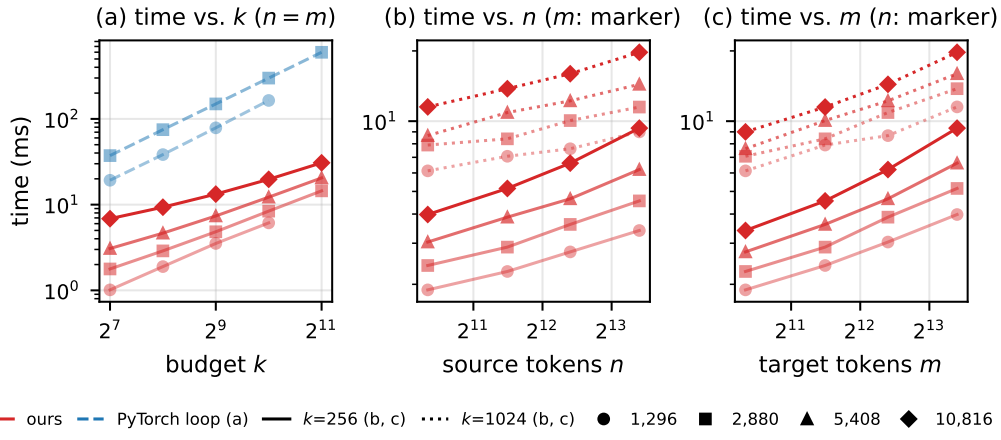



Figure 1: Inference time versus $k$ for FLASH GREEDY (solid) and PyTorch loops (dashed). Besides, inference time for different source tokens $n$ and target tokens $m$.

**Robustness to hyperparameters $n, m, k$ (Figure 1).** For the video similarity matrix covering all regimes, a 16× larger $k$ costs FLASH GREEDY only 4.5–8.1× more time for $n$=$m \geq$ 2,880 (e.g., 6.9 → 30.7 ms for $n$=$m$=10,816). While the PyTorch loops are linear in $k$ (the gap widens from 21× to 41× for $n$=$m$=2,880). 8.3× more source tokens cost 1.5–2.3× more time, as every iteration still reads $T$ rows. In addition, 8.3× more target tokens cost 1.9–2.7× more time, as every row read scales with $m$. The curves are smooth across regimes: the time is 6.1–19.7 ms at $k$=1,024, while the similarity matrix grows 70×.

Table 2: Ablation study on the number of tiles, $T \in \{14, 27, 54, 108, 216\}$ for different cases.

| Setting | $n$ | $k$ | inference time (in ms) of different $T$ | | | | | outer iterations of different $T$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 14 | 27 | 54 | **108** | 216 | 14 | 27 | 54 | 108 | **216** |
| MMTok | 2,880 | 320 | 9.33 | 6.19 | 4.17 | **3.28** | 3.58 | 1471 | 883 | 535 | 326 | **210** |
| SCOPE | 1,296 | 128 | 3.16 | 2.27 | 1.57 | **1.25** | 1.36 | 548 | 367 | 230 | 140 | **87** |
| SCOPE | 1,296 | 512 | 5.81 | 4.57 | 3.53 | **3.12** | 3.67 | 846 | 580 | 380 | 230 | **150** |
| SCOPE | 2,880 | 320 | 9.64 | 6.58 | 4.41 | **3.44** | 3.76 | 1527 | 950 | 579 | 353 | **232** |
| SCOPE | 2,880 | 1280 | 16.68 | 12.52 | 9.55 | **8.61** | 10.19 | 2184 | 1394 | 886 | 564 | **374** |
| SCOPE | 5,184 | 512 | 20.47 | 12.72 | 8.58 | **7.02** | 7.31 | 2651 | 1682 | 993 | 633 | **392** |
| SCOPE | 10,816 | 1024 | 58.68 | 34.16 | 22.34 | **18.58** | 19.62 | 5756 | 3365 | 1892 | 1161 | **765** |
| SelAnn | 3,000 | 128 | 9.24 | 5.56 | 3.28 | **2.18** | 2.32 | 1530 | 863 | 491 | 286 | **160** |
| SelAnn | 3,000 | 512 | 14.43 | 9.40 | 6.29 | **4.96** | 5.54 | 2197 | 1286 | 753 | 455 | **278** |

Table 3: The effect of bool position vector for selecting multiple tokens. At each outer iteration, we can only select one token. Our method is to select multiple tokens by bool position vector in the inner loop.

| Setting | $n$ | $k$ | inference time (in ms) | | | outer iterations | | |
|---|---|---|---|---|---|---|---|---|
| | | | multi-token | one-token | speedup | multi-token | one-token | speedup |
| MMTok | 2,880 | 320 | **3.28** | 3.88 | $1.18\times$ | **326** | 481 | $1.47\times$ |
| MMTok | 2,880 | 640 | **5.11** | 6.85 | $1.34\times$ | **410** | 801 | $1.95\times$ |
| MMTok | 2,880 | 1280 | **8.23** | 12.93 | $1.57\times$ | **503** | 1441 | $2.86\times$ |
| SCOPE | 1,296 | 128 | **2.18** | 2.61 | $1.19\times$ | **140** | 209 | $1.49\times$ |
| SCOPE | 1,296 | 256 | **3.43** | 4.50 | $1.31\times$ | **185** | 340 | $1.83\times$ |
| SCOPE | 1,296 | 512 | **3.10** | 4.71 | $1.51\times$ | **230** | 597 | $2.59\times$ |
| SCOPE | 2,880 | 320 | **3.45** | 4.03 | $1.16\times$ | **353** | 510 | $1.44\times$ |
| SCOPE | 2,880 | 640 | **5.35** | 7.06 | $1.31\times$ | **449** | 838 | $1.86\times$ |
| SCOPE | 2,880 | 1280 | **8.60** | 13.11 | $1.52\times$ | **564** | 1481 | $2.62\times$ |
| SCOPE | 5,184 | 512 | **7.03** | 7.78 | $1.10\times$ | **633** | 851 | $1.34\times$ |
| SCOPE | 5,184 | 1024 | **10.43** | 13.03 | $1.24\times$ | **794** | 1373 | $1.72\times$ |
| SCOPE | 10,816 | 1024 | **18.50** | 20.33 | $1.10\times$ | **1161** | 1617 | $1.39\times$ |
| SCOPE | 10,816 | 2048 | **27.45** | 34.25 | $1.24\times$ | **1469** | 2656 | $1.80\times$ |

**Number of tiles $T$ (Table 2).** $T$=108 (the number of SMs) is the fastest in all nine cases. Fewer tiles leave SMs idle and evaluate fewer candidates per iteration: halving $T$ costs 1.5–1.8$\times$ more iterations, and $T$=14 is 1.9–4.2$\times$ slower. $T$=216 reduces the iterations further (326 $\rightarrow$ 210 for MMTok) but is 4–18% slower. Two blocks share the SRAM, registers and load bandwidth of one SM. While every step scans twice as many records and every barrier waits for twice as many blocks. Thus, one tile per SM is the best trade-off, and it needs no tuning.

**Bool position vector (Table 3).** Without it, other candidates may be stale after an admission, so at most one token is admitted per iteration (481 iterations for 319 tokens on MMTok). With it, FLASH GREEDY needs 1.34–2.86$\times$ fewer iterations and is 1.10–1.57$\times$ faster, and the gain grows with $k$ (MMTok: from 1.18$\times$ to 1.57$\times$). An admission changes the gain of another candidate only on positions that both cover, which an AND of two $m$-bit vectors decides without a $4m$-byte row read. As $\mathbf{y}$ saturates, such overlaps become rare (3–16% of the candidate–admission pairs); the time gain is smaller, as the per-token work is unchanged.

### 5.5 ANALYSIS

**Runtime bottleneck (Figure 2).** In-kernel cycle counters show that initialization (lines 1–3) takes 1–4%, and the candidate row read (lines 13–14) takes 11–16% in the SRAM and L2 regimes (17–33% in the HBM regime). The bottleneck is synchronization: the record scan and argmax (lines 16–19), the grid barriers per iteration, and per admitted token take 56–60% of the kernel in the SRAM and L2 regimes (39–54% in the HBM regime). Since FLASH GREEDY is latency-bound once the matrix is on chip, we accelerate the fused kernel by three measures. (1) *fewer synchronizations*: $T$ candidates and several admissions per iteration, and one launch instead of 8–16 launches per token. (2) *cheaper synchronizations*: SMs exchange only 12-byte records and $m$-bit bool vectors through

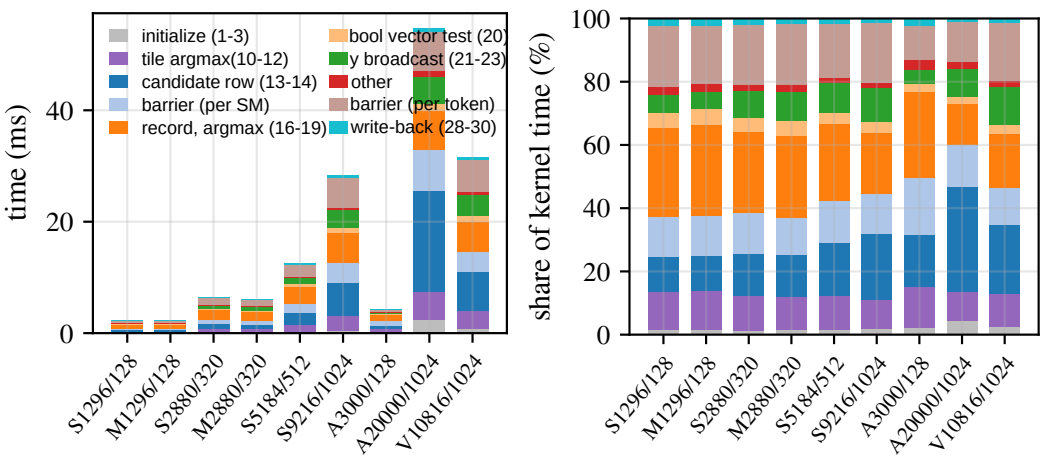



Figure 2: Runtime of each part in Algorithm 2 inside the fused kernel for different similarity matrices (S: SCOPE, M: MMTok, A: Selective Annotation, V: LongVideoBench; $n/k$). Besides, share of the kernel time of each part in Algorithm 2.

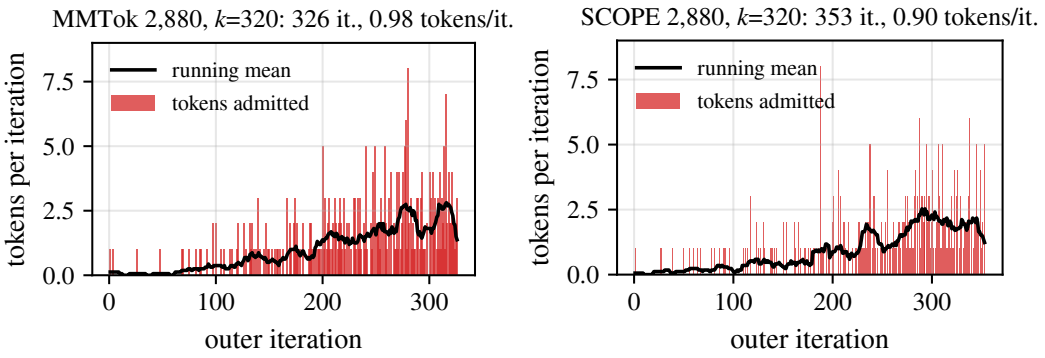



Figure 3: The number of tokens admitted per outer iteration with a running mean (line).

L2 cache, double-buffered by the step parity (one barrier per step). And each SM runs the same replicated selection loop with warp shuffles and votes. (3) *faster rows*: all warps read one row with 16-byte loads in flight, against the private $\mathbf{y}$ in SRAM.

**Tokens per outer iteration (Figure 3).** Early iterations admit few tokens (15 and 16 in the first 100). Because the initial cached gains are loose bounds, yet each refreshes 108 gains in the time of one row read. As the gains tighten and $\mathbf{y}$ saturates, the running mean reaches about two tokens per iteration (up to 8). Besides, multi-token iterations supply 73–76% of the tokens. Naive Greedy needs about as many iterations but reads all rows in each, and Lazy Greedy admits one token per serial iteration. FLASH GREEDY turns the failed lazy pops into parallel refreshes, and admits several tokens per synchronization, so its advantage grows with $k$ (2.54 tokens per iteration at $k$=1,280).

## 6 CONCLUSION

We presented the FLASH GREEDY, an exact GPU solver for greedy facility location maximization in token subset selection. FLASH GREEDY replaces the priority queue of Lazy Greedy by $T$ gain tiles, one per SM. Each SM re-evaluates only the top candidate of its tile, and the submodular upper-bound certifies the greedy choice. In addition, the bool position vector admits several tokens per iteration with a bitwise AND. Based on the data size, similarity matrix is placed in the SRAM, L2 cache or HBM. FLASH GREEDY is one IO-aware persistent kernel that returns exactly the Naive Greedy subset. Evaluations across various configurations show that FLASH GREEDY is at least 9× faster than the fastest existing solver. It applies without change to other facility location problems, such as coreset selection (Mirzasoleiman et al., 2020), active learning (Sener & Savarese, 2018), and document summarization (Lin & Bilmes, 2011).

## AI USE STATEMENT

In this work, we used generative AI tools to aid or polish writing. We have not used generative AI tools for writing code and conceptualizing the method. Additionally, we used generative AI tools to survey the literature in this field. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

## REFERENCES

Ashwinkumar Badanidiyuru and Jan Vondrák. Fast algorithms for maximizing submodular functions. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1497–1514. SIAM, 2014.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report, 2025. URL https://arxiv.org/abs/2502.13923.

Eric Balkanski and Yaron Singer. The adaptive complexity of maximizing a submodular function. In *Proceedings of the 50th annual ACM SIGACT symposium on theory of computing*, pp. 1138–1151, 2018.

Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. The fifth pascal recognizing textual entailment challenge. *TAC*, 7(8):1, 2009.

Adam Breuer, Eric Balkanski, and Yaron Singer. The fast algorithm for submodular maximization. In *International conference on machine learning*, pp. 1134–1143. PMLR, 2020.

Xiwen Chen, Wenhui Zhu, Gen Li, Xuanzhao Dong, Yujian Xiong, Hao Wang, Peijie Qiu, Qingquan Song, Zhipeng Wang, Shao Tang, Yalin Wang, and Abolfazl Razi. Otprune: Distribution-aligned visual token pruning via optimal transport. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Findings*, pp. 5849–5859, June 2026.

Janghoon Cho, Jungsoo Lee, Munawar Hayat, Kyuwoong Hwang, Fatih Porikli, and Sungha Choi. Floc: Facility location-based efficient visual token compression for long video understanding. In *International Conference on Learning Representations*, volume 2026, pp. 102973–102996, 2026.

Gerard Cornuejols, Marshall L Fisher, and George L Nemhauser. Exceptional paper—location of bank accounts to optimize float: An analytic study of exact and approximate algorithms. *Management science*, 23(8):789–810, 1977.

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

Jinhong Deng, Wen Li, Joey Tianyi Zhou, and Yang He. Scope: Saliency-coverage oriented token pruning for efficient multimodel llms. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 161527–161552, 2025. doi: 10.52202/085713-5394.

William B Dolan, Chris Quirk, and Chris Brockett. Unsupervised construction of large paraphrase corpora: Exploiting massively parallel news sources. In *COLING 2004: Proceedings of the 20th international conference on computational linguistics*, pp. 350–356, 2004.

Sixun Dong, Juhua Hu, Mian Zhang, Ming Yin, Yanjie Fu, and Qi Qian. Mmtok: Multimodal coverage maximization for efficient inference of vlms. In *International Conference on Learning Representations*, volume 2026, pp. 48075–48099, 2026.

Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4): 634–652, 1998.

Vishal Kaushal, Ganesh Ramakrishnan, and Rishabh Iyer. Submodlib: A submodular optimization library. *arXiv preprint arXiv:2202.10680*, 2022.

Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pp. 39–48, 2020.

Andreas Krause and Daniel Golovin. Submodular function maximization. *Tractability*, 3(71-104): 3, 2014.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.

Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web*, 6(2):167–195, 2015.

Bo Li, Kaichen Zhang, Hao Zhang, Dong Guo, Renrui Zhang, Feng Li, Yuanhan Zhang, Ziwei Liu, and Chunyuan Li. Llava-next: Stronger llms supercharge multimodal capabilities in the wild, May 2024. URL https://llava-vl.github.io/blog/2024-05-10-llava-next-stronger-llms/.

Yifan Li, Yifan Du, Kun Zhou, Jinpeng Wang, Xin Zhao, and Ji-Rong Wen. Evaluating object hallucination in large vision-language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 292–305, 2023.

Hui Lin and Jeff Bilmes. A class of submodular functions for document summarization. In *Proceedings of the 49th annual meeting of the association for computational linguistics: human language technologies*, pp. 510–520, 2011.

Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. LLaVA-NeXT: Improved reasoning, OCR, and world knowledge. https://llava-vl.github.io/blog/2024-01-30-llava-next/, 2024.

Baharan Mirzasoleiman, Ashwinkumar Badanidiyuru, Amin Karbasi, Jan Vondrák, and Andreas Krause. Lazier than lazy greedy. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization. *The Journal of Machine Learning Research*, 17(1):8330–8373, 2016.

Baharan Mirzasoleiman, Jeff Bilmes, and Jure Leskovec. Coresets for data-efficient training of machine learning models. In *International Conference on Machine Learning*, pp. 6950–6960. PMLR, 2020.

George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14(1):265–294, 1978.

Muhammad Osama, Duane Merrill, Cris Cecka, Michael Garland, and John D Owens. Stream-k: Work-centric parallel decomposition for dense matrix-matrix multiplication on the gpu. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 429–431, 2023.

Roi Pony, Daniel Ezer, Adi Raz Goldfarb, Idan Friedman, Oshri Naparstek, and Udi Barzelay. Flash-maxsim: Io-aware fused kernels for late-interaction retrieval. *arXiv preprint arXiv:2605.29517*, 2026.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

Jacob Schreiber, Jeffrey Bilmes, and William Stafford Noble. apricot: Submodular selection for data summarization in python. *Journal of Machine Learning Research*, 21(161):1–6, 2020.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=H1aIuk-RW.

Ashutosh Sharma. Tilemaxsim: Io-aware gpu maxsim scoring with dimension tiling and fused product quantization. *arXiv preprint arXiv:2606.26439*, 2026.

Amanpreet Singh, Vivek Natarajan, Meet Shah, Yu Jiang, Xinlei Chen, Dhruv Batra, Devi Parikh, and Marcus Rohrbach. Towards vqa models that can read. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8309–8318. IEEE, 2019.

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*, pp. 1631–1642, 2013.

Hongjin SU, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Selective annotation makes language models better few-shot learners. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=qY1hlv7gwg.

Xuehui Wang, Xuankun Yang, and Wei Shen. Combating textual noise and redundancy: Entropy-aware dense visual token pruning. In *European Conference on Computer Vision*, pp. 559–580. Springer, 2026.

Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long papers)*, pp. 1112–1122, 2018.

Haoning Wu, Dongxu Li, Bei Chen, and Junnan Li. Longvideobench: A benchmark for long-context interleaved video-language understanding. *Advances in Neural Information Processing Systems*, 37:28828–28857, 2024.

Shuo Yang, Haocheng Xi, Yilong Zhao, Muyang Li, Xiaoze Fan, Jintao Zhang, Han Cai, Yujun Lin, Xiuyu Li, Kurt Keutzer, et al. Flash-kmeans: Fast and memory-efficient exact k-means. *arXiv preprint arXiv:2603.09229*, 2026.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4791–4800, 2019.

Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. A study of single and multi-device synchronization methods in nvidia gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 483–493. IEEE, 2020.

Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. Perks: a locality-optimized execution model for iterative memory-bound gpu applications. In *Proceedings of the 37th International Conference on Supercomputing*, pp. 167–179, 2023.

Qizhe Zhang, Mengzhen Liu, Lichen Li, Ming Lu, Yuan Zhang, Junwen Pan, Qi She, and Shanghang Zhang. Beyond attention or similarity: Maximizing conditional diversity for token pruning in mllms. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 25438–25468, 2025a. doi: 10.52202/085713-0855.

Yuanhan Zhang, Jinming Wu, Wei Li, Bo Li, Zejun MA, Ziwei Liu, and Chunyuan Li. LLaVA-video: Video instruction tuning with synthetic data. *Transactions on Machine Learning Research*, 2025b. ISSN 2835-8856. URL https://openreview.net/forum?id=EElFGvt39K.

Qingchan Zhu, Weihang You, Hanqi Jiang, Changdi Yang, Tianming Liu, and Geng Yuan. Who speaks for the pruned? visual token pruning as coverage optimization. *arXiv preprint arXiv:2609.03158*, 2026.

# A Additional Results

Table 4: Experimental results of more configurations. Inference time is ms unless marked.

| Setting | Dataset | $n$ | $m$ | $k$ | regime | submodlib lazy | apricot naive | apricot lazy | PyTorch naive | **ours flash** | speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MMTok | TextVQA | 1,296 | 1,301 | 512 | SRAM | 88.2 | 477 | 110 | 42.7 | **3.03** | 14.0× |
| MMTok | TextVQA | 2,880 | 2,885 | 640 | L2 | 347 | 1.8 s | 14 s | 146 | **5.14** | 28.4× |
| MMTok | TextVQA | 5,184 | 5,188 | 512 | HBM | 1.2 s | 3.1 s | 626 | 334 | **6.95** | 48.0× |
| MMTok | TextVQA | 5,184 | 5,188 | 1,024 | HBM | 1.4 s | 10 s | 2.1 s | 667 | **10.2** | 65.3× |
| MMTok | TextVQA | 5,760 | 5,769 | 512 | HBM | 1.3 s | 3.6 s | 655 | 405 | **7.39** | 54.8× |
| MMTok | TextVQA | 5,760 | 5,769 | 1,024 | HBM | 1.5 s | 11 s | 2.3 s | 810 | **10.9** | 74.3× |
| MMTok | POPE | 1,296 | 1,300 | 256 | SRAM | 61.2 | 169 | 81.6 | 21.4 | **1.90** | 11.2× |
| MMTok | POPE | 1,296 | 1,300 | 512 | SRAM | 87.7 | 485 | 115 | 42.0 | **3.16** | 13.2× |
| MMTok | POPE | 2,880 | 2,884 | 640 | L2 | 331 | 1.8 s | 330 | 146 | **5.02** | 29.0× |
| MMTok | POPE | 2,880 | 2,884 | 1,280 | L2 | 479 | 6.0 s | 1.2 s | 292 | **8.13** | 35.9× |
| SCOPE | TextVQA | 1,296 | 1,296 | 256 | SRAM | 59.9 | 176 | 69.8 | 59.3 | **1.85** | 32.0× |
| SCOPE | TextVQA | 1,296 | 1,296 | 512 | SRAM | 82.9 | 467 | 107 | 118 | **3.05** | 27.1× |
| SCOPE | TextVQA | 2,880 | 2,880 | 640 | L2 | 387 | 1.8 s | 359 | 339 | **5.29** | 64.0× |
| SCOPE | TextVQA | 2,880 | 2,880 | 1,280 | L2 | 522 | 5.8 s | 1.1 s | 678 | **8.41** | 62.0× |
| SCOPE | TextVQA | 5,184 | 5,184 | 512 | HBM | 1.2 s | 3.1 s | 618 | 689 | **7.03** | 87.9× |
| SCOPE | TextVQA | 5,760 | 5,760 | 512 | HBM | 1.4 s | 3.6 s | 684 | 831 | **7.58** | 90.2× |
| SCOPE | TextVQA | 5,760 | 5,760 | 1,024 | HBM | 1.7 s | 11 s | 2.2 s | 1.7 s | **11.3** | 150× |
| SCOPE | POPE | 1,296 | 1,296 | 128 | SRAM | 53.1 | 78.8 | 53.8 | 29.5 | **1.19** | 24.7× |
| SCOPE | POPE | 1,296 | 1,296 | 256 | SRAM | 65.3 | 174 | 72.5 | 59.6 | **1.87** | 31.8× |
| SCOPE | POPE | 2,880 | 2,880 | 320 | L2 | 326 | 685 | 191 | 169 | **3.36** | 50.2× |
| SCOPE | POPE | 2,880 | 2,880 | 640 | L2 | 403 | 1.9 s | 363 | 339 | **5.27** | 64.3× |
| SCOPE | POPE | 2,880 | 2,880 | 1,280 | L2 | 539 | 6.0 s | 1.2 s | 678 | **8.36** | 64.4× |
| SCOPE | Video | 5,408 | 5,408 | 512 | HBM | 1.3 s | 3.6 s | 653 | 744 | **6.87** | 95.0× |
| SCOPE | Video | 5,408 | 5,408 | 1,024 | HBM | 1.6 s | 10 s | 2.1 s | 1.5 s | **10.4** | 144× |
| SCOPE | Video | 6,272 | 6,272 | 512 | HBM | 1.9 s | 4.2 s | 803 | 968 | **7.83** | 102× |
| SCOPE | Video | 6,272 | 6,272 | 1,024 | HBM | 2.2 s | 13 s | 2.2 s | 1.9 s | **11.6** | 163× |
| SelAnn | MNLI | 3,000 | 3,000 | 256 | L2 | 552 | 587 | 318 | 78.6 | **3.32** | 23.6× |
| SelAnn | MNLI | 6,000 | 6,000 | 128 | HBM | 2.2 s | 697 | 823 | 119 | **4.70** | 25.3× |
| SelAnn | MNLI | 6,000 | 6,000 | 512 | HBM | 2.7 s | 4.1 s | 1.3 s | 478 | **8.70** | 54.9× |
| SelAnn | MNLI | 6,000 | 6,000 | 1,024 | HBM | 3.2 s | 12 s | 2.9 s | 957 | **12.6** | 75.9× |
| SelAnn | MNLI | 10,000 | 10,000 | 128 | HBM | 5.8 s | 1.8 s | 2.0 s | 306 | **8.37** | 36.5× |
| SelAnn | MNLI | 10,000 | 10,000 | 512 | HBM | 7.1 s | 8.8 s | 2.9 s | 1.2 s | **14.3** | 83.9× |
| SelAnn | MNLI | 10,000 | 10,000 | 1,024 | HBM | 7.9 s | 22 s | 5.7 s | 2.4 s | **19.6** | 122× |
| SelAnn | MNLI | 20,000 | 20,000 | 128 | HBM | 25 s | 6.6 s | 7.1 s | 1.2 s | **20.7** | 57.9× |
| SelAnn | MNLI | 20,000 | 20,000 | 512 | HBM | 28 s | 29 s | 9.2 s | 4.8 s | **31.6** | 151× |
| SelAnn | MRPC | 3,000 | 3,000 | 128 | L2 | 535 | 238 | 275 | 39.4 | **2.36** | 16.6× |
| SelAnn | MRPC | 3,000 | 3,000 | 256 | L2 | 594 | 603 | 325 | 79.0 | **3.35** | 23.5× |
| SelAnn | MRPC | 3,000 | 3,000 | 512 | L2 | 693 | 1.5 s | 455 | 157 | **5.16** | 30.4× |
| SelAnn | SST-5 | 3,000 | 3,000 | 128 | L2 | 486 | 208 | 271 | 39.3 | **2.30** | 17.0× |
| SelAnn | SST-5 | 3,000 | 3,000 | 256 | L2 | 588 | 570 | 312 | 78.7 | **3.29** | 23.9× |
| SelAnn | SST-5 | 3,000 | 3,000 | 512 | L2 | 643 | 1.5 s | 433 | 159 | **5.09** | 31.2× |
| SelAnn | DBpedia | 3,000 | 3,000 | 128 | L2 | 462 | 180 | 278 | 39.4 | **2.31** | 17.0× |
| SelAnn | DBpedia | 3,000 | 3,000 | 256 | L2 | 590 | 542 | 317 | 78.6 | **3.30** | 23.8× |
| SelAnn | DBpedia | 3,000 | 3,000 | 512 | L2 | 634 | 1.4 s | 418 | 157 | **5.08** | 30.9× |
| SelAnn | DBpedia | 6,000 | 6,000 | 128 | HBM | 2.1 s | 714 | 863 | 119 | **4.76** | 25.0× |
| SelAnn | DBpedia | 6,000 | 6,000 | 512 | HBM | 2.6 s | 3.9 s | 1.3 s | 479 | **8.78** | 54.5× |
| SelAnn | DBpedia | 6,000 | 6,000 | 1,024 | HBM | 3.1 s | 11 s | 2.9 s | 957 | **12.6** | 75.9× |
| SelAnn | DBpedia | 10,000 | 10,000 | 128 | HBM | 5.5 s | 1.7 s | 2.1 s | 305 | **8.63** | 35.3× |
| SelAnn | DBpedia | 10,000 | 10,000 | 512 | HBM | 6.4 s | 8.4 s | 2.9 s | 1.2 s | **14.6** | 82.1× |
| SelAnn | DBpedia | 10,000 | 10,000 | 1,024 | HBM | 7.1 s | 20 s | 5.4 s | 2.4 s | **19.9** | 120× |
| SelAnn | DBpedia | 20,000 | 20,000 | 128 | HBM | 24 s | 6.7 s | 7.4 s | 1.2 s | **20.8** | 57.6× |
| SelAnn | DBpedia | 20,000 | 20,000 | 512 | HBM | 28 s | 29 s | 9.2 s | 4.8 s | **31.3** | 153× |
| SelAnn | DBpedia | 20,000 | 20,000 | 1,024 | HBM | 30 s | 63 s | 14 s | 9.6 s | **40.4** | 237× |
| SelAnn | RTE | 2,488 | 2,488 | 128 | L2 | 324 | 133 | 225 | 30.8 | **2.09** | 14.7× |
| SelAnn | RTE | 2,488 | 2,488 | 256 | L2 | 353 | 411 | 254 | 62.0 | **2.96** | 20.9× |
| SelAnn | RTE | 2,488 | 2,488 | 512 | L2 | 491 | 1.1 s | 361 | 124 | **4.61** | 26.8× |
| SelAnn | HellaSwag | 3,000 | 3,000 | 128 | L2 | 455 | 204 | 226 | 39.3 | **2.13** | 18.4× |
| SelAnn | HellaSwag | 3,000 | 3,000 | 256 | L2 | 519 | 534 | 264 | 78.6 | **3.13** | 25.1× |
| SelAnn | HellaSwag | 3,000 | 3,000 | 512 | L2 | 601 | 1.4 s | 387 | 157 | **4.86** | 32.3× |

## B GREEDY EXACTNESS

We show that FLASH GREEDY (Algorithm 2) admits exactly the same tokens as Naive Greedy, in the same order. The proof uses three simple facts.

**Fact 1 (gains only decrease).** If $S \subseteq S'$, then $\Delta(i \mid S') \leq \Delta(i \mid S)$. This is the submodularity of Definition 1: a gain computed for an older (smaller) subset is an upper bound of the current gain.

**Fact 2 (cached gains are upper bounds).** At the start of every outer iteration, each unselected token $i$ has $\mathbf{x}_i \geq \Delta(i \mid S)$ and $\mathbf{x}_i \geq 0$. And every selected token has $\mathbf{x}_i = -1$. This holds after initialization, because line 3 computes the exact gains for $S = \{v\}$ and line 4 marks $v$. During an outer iteration, the gain vector is changed only at the candidates (lines 11 and 29). An admitted candidate gets $\mathbf{g}^a = -1$ (line 21), so it is marked as selected. A candidate that is not admitted gets back $\mathbf{g}^t$. Its exact gain for the subset at the start of the iteration (lines 11 and 13), which is an upper bound of its gain for the larger subset by Fact 1. Every non-candidate keeps its old value, which also remains an upper bound by Fact 1.

**Fact 3 (non-overlapping positions keep the gain).** Let $a_1, \ldots, a_h$ be the tokens admitted so far in its inner loop, so that $\mathbf{b} = \mathbf{z}^{a_1} \mid \cdots \mid \mathbf{z}^{a_h}$ and the current coverage vector is $\mathbf{y}' = \max(\mathbf{y}, \mathbf{s}_{a_1}, \ldots, \mathbf{s}_{a_h})$. A coordinate $\mathbf{y}'_j$ can differ from $\mathbf{y}_j$ only if $\mathbf{s}_{a_\ell j} > \mathbf{y}_j$ for some $\ell$, i.e., only where $\mathbf{b}_j$ is True. Now take another candidate $c$ whose bool vector satisfies $\mathbf{b} \,\&\, \mathbf{z}^c = \text{False}$ everywhere. On a position $j$ where $\mathbf{z}^c_j$ is True, $\mathbf{b}_j$ is False, so $\mathbf{y}'_j = \mathbf{y}_j$ and the term $\max(\mathbf{s}_{cj} - \mathbf{y}'_j, 0)$ is unchanged. On a position $j$ where $\mathbf{z}^c_j$ is False, $\mathbf{s}_{cj} \leq \mathbf{y}_j \leq \mathbf{y}'_j$, so the term is 0 before and after. $\Delta(c \mid S \cup \{a_1, \ldots, a_h\}) = \Delta(c \mid S) = \mathbf{g}^c$: the gain of $c$ is still exact without reading its row again.

**Theorem 1.** *Algorithm 2 returns the Naive Greedy subset, and the tokens are admitted in the Naive Greedy order.*

*Proof.* Both algorithms start with the token $v$ of the largest row sum. Assume that the first $\ell$ admitted tokens coincide. Let $S$ be this subset, and consider the next admission of candidate $a$ at line 21. We show that $a$ has the largest marginal gain $\Delta(\cdot \mid S)$ among all unselected tokens.

- *The gain of $a$ is exact.* $\mathbf{g}^a$ was computed at line 13 against the coverage vector at the start of the outer iteration. If $a$ is the first admission of this inner loop, the coverage vector has not changed. Otherwise, the test of line 20 has passed, and Fact 3 gives $\mathbf{g}^a = \Delta(a \mid S)$.
- *Token $a$ beats every other candidate.* Every other unselected candidate $c$ has $\mathbf{g}^c \geq \Delta(c \mid S)$ by Fact 1, because $\mathbf{g}^c$ was computed for an older subset. Since $a$ maximizes $\mathbf{g}$ (line 19), we have $\Delta(a \mid S) = \mathbf{g}^a \geq \mathbf{g}^c \geq \Delta(c \mid S)$. Candidates that were already admitted, or whose tile has no unselected token, have $\mathbf{g} = -1 < 0 \leq \mathbf{u}^*$ and are never admitted.
- *Token $a$ beats every non-candidate.* For every unselected non-candidate $i$ in tile $t$, Fact 2 and lines 12 and 16 give $\Delta(i \mid S) \leq \mathbf{x}_i \leq \mathbf{u}^t \leq \mathbf{u}^*$. The inner loop only runs while $\max_t \mathbf{g}^t \geq \mathbf{u}^*$ (line 18), so $\Delta(a \mid S) = \mathbf{g}^a \geq \mathbf{u}^* \geq \Delta(i \mid S)$. When no unselected non-candidate is left, $\mathbf{u}^*$ is set to 0, which is below every gain of an unselected candidate.

Therefore, $a$ is the token with the largest marginal gain, i.e., exactly the token that Naive Greedy selects from $S$. And line 22 updates $\mathbf{y}$ exactly as Naive Greedy does. By induction, all $k$ admitted tokens coincide. □

**Remark 1.** If an outer iteration admits no token, then $\max_t \mathbf{g}^t < \mathbf{u}^*$. Let $i^*$ be a non-candidate with $\mathbf{x}_{i^*} = \mathbf{u}^*$, and let $c$ be the candidate of its tile. Because $c$ was chosen as the largest cached gain of the tile, $\mathbf{x}_c \geq \mathbf{x}_{i^*} = \mathbf{u}^* > \mathbf{g}^c$, so line 29 strictly lowers $\mathbf{x}_c$ to its exact gain. While $S$ does not change, an exact cached gain cannot be lowered again. So, at most $n$ consecutive outer iterations admit no token, and Algorithm 2 always terminates with $|S| = k$. In practice, it needs only 0.4–1.3 outer iterations per admitted token (Table 3). The same argument holds when the row of an overlapping candidate is read again against the current $\mathbf{y}$ before its admission, since its $\mathbf{g}$ is then exact again. This only changes how many tokens an outer iteration can admit, not which tokens are admitted. Consequently, FLASH GREEDY inherits the $(1 - 1/e)$ approximation guarantee of Naive Greedy.

## C SIMILARITY MATRIX

**MMTok** selects vision tokens by maximizing their coverage of two target sets: the text tokens of the question and all $n$ vision tokens. The text–vision similarity $\mathbf{M}^{tv} \in \mathbb{R}^{m_t \times n}$ is the cosine between the LLM input embeddings of $m_t$ question tokens and the vision tokens after multimodal projector. The vision–vision similarity $\mathbf{M}^{vv} \in \mathbb{R}^{n \times n}$ is the cosine between the penultimate-layer patch features of the vision encoder, before the projector. Equivalently, the stacked matrix $[\mathbf{P}; \alpha\mathbf{Q}] \in \mathbb{R}^{(m_t+n)\times n}$ with $\mathbf{P} = \mathrm{softmax}(\mathbf{M}^{tv}/\tau_t)/m_t$ and $\mathbf{Q} = \mathrm{softmax}(\mathbf{M}^{vv}/\tau_v)/n$ defines a single facility location function. And our $\mathbf{s}$ is its transpose with $m = m_t + n$ targets. MMTok solves it with a `torch.jit` Naive Greedy loop, which is our PyTorch baseline.

**SCOPE** scores every vision token by its saliency, the attention from the [CLS] token. It times the coverage gain $\Delta(v \mid S)$ for the set coverage $f(S) = \sum_u \max_{s\in S} \mathrm{sim}(u, s)$ over all $n$ vision tokens. Where $\mathrm{sim}$ is the cosine between the penultimate-layer patch features of the vision encoder, before the projector. The similarity matrix is therefore square, $\mathbf{s} \in \mathbb{R}^{n\times n}$, and we store the $(1 + \cos)/2$. We use the unit saliency, so that, SCOPE's PyTorch greedy loop is exactly the Naive Greedy of its coverage term.

**Selective Annotation** will choose which unlabeled examples to annotate for in-context learning from a pool of $3{,}000$ random instances. Every instance is embedded by a Sentence-BERT model. Then, the embeddings are mean-centered over the pool, and instances are compared by cosine similarity. Their facility location baseline (MFL) maximizes $\sum_i \max_{u\in S} \cos(\mathbf{x}_i, \mathbf{x}_u)$ over the $3{,}000 \times 3{,}000$ cosine matrix. And it starts the coverage at $-1$. Its PyTorch loop for Naive Greedy is our baseline.