Available Online: 31 July 2026 DOI: 10.54254/2977-3903/2026.35767

A resource accounting baseline for low-weight coded sparse MVM under edge storage constraints

Haoyu Zhu

School of Communication and Information Engineering, Shanghai University, Shanghai, China

theodorauc@gmail.com

Abstract. Coded Matrix-Vector Multiplication (MVM) is often evaluated by recovery latency, but in storage-constrained edge settings the preferred operating point can change when pre-storage feasibility, materialized sparsity, amortized communication, decoding overhead, and failure modes are accounted for together. This paper studies a narrow resource-accounting question: when a per-worker storage budget cannot accommodate a fixed dense coded design, can systematic low-weight redundancy provide a feasible reliability-latency compromise for sparse MVM? The simulator separates offline encoded-block placement from online worker-level vector broadcast and reports storage feasibility, unassigned tasks, amortized communication, worker nonzero operations, source-accumulation encoding cost, materialized encoded nonzeros, decoding cost, timeout-capped latency, and failure reasons. In the configured simulator, dense coding remains latency-best when storage is unconstrained, but it requires 150.9 KiB of pre-stored blocks and about 3.07×10^4 source-accumulation operations in the baseline setting. The low-weight design obtains 0.955 recovery success, 0.1632 s success-only mean latency, 0.2003 s timeout-capped mean latency, 68.5 KiB of pre-storage, and full feasibility after storage-constrained placement under an 8 KiB stress budget where the fixed dense and budgeted-dense placements cannot place all parity blocks. Workload, placement, support, and scaling diagnostics show that the low-weight point is useful but not optimal: placement is heuristic, support construction matters, and fixed $w = 3$ with $R/K \approx 1/3$ weakens as K grows. The contribution is therefore a reproducible resource-accounting benchmark and lightweight reference baseline, not a new straggler-optimal or literature-faithful coded-computation scheme.

Keywords: edge computing, coded distributed computing, matrix-vector multiplication, sparse coding, storage constraints

1. Introduction

Edge and Internet-of-Things (IoT) platforms move computation closer to data sources, but the devices available near the edge are often heterogeneous, intermittently connected, and constrained in memory and energy [1-3]. Collaborative computation can reduce the burden on a single edge server or gateway, yet the completion time of synchronous distributed jobs is frequently dominated by slow or failed workers. Coded distributed computing addresses this straggler problem by adding redundant coded tasks so that the master can recover after receiving a sufficient subset of results [4-6]. For sparse Matrix-Vector Multiplication (MVM),

however, dense coded tasks are not always the most useful operating point. Dense linear combinations can improve recovery power, but they also enlarge pre-stored encoded blocks, increase worker nonzero operations, and require heavier decoding. This issue is well known: sparse coded linear transforms, sparse MVM, low-weight encodings, rateless MVM, sparse/flexible coded learning, and sparsity-preserving edge encodings have all been studied [7-13]. Therefore, the present manuscript deliberately avoids a broad novelty claim. The narrower question studied here is: in a pre-stored sparse MVM setting, when the per-worker storage budget is too small for all dense coded blocks, can systematic low-weight redundancy trade latency for storage feasibility and recovery reliability? This question is practical rather than coding-theoretic. It is relevant when an implementer wants to know whether dense coding's latency advantage survives a tight placement budget, and whether a simple low-weight baseline is worth considering before implementing a stronger straggler-optimal or rateless construction.

The contributions are as follows.

• This paper defines a resource-accounting protocol for pre-stored sparse MVM with offline/online separation, worker-level vector broadcast, explicit per-worker storage feasibility, amortized communication, timeout metrics, and failure-reason accounting.

• The benchmark implements a systematic low-weight reference baseline with bounded-row-weight support, storage-constrained placement, peeling recovery, and same-support dense-rank diagnostics.

• The evaluation compares uncoded, replication, fixed dense-coded, storage-aware dense-budgeted, LT-like ablation, and dense-decoded low-weight ablation methods without claiming faithful reproduction of the original rateless or Das-style algorithms.

• The reported experiments include 1000-seed sweeps over storage, straggler, sparsity, reuse, support, placement, scale, and workload settings, plus a reduced $K/R/w$ design grid, to expose both useful tradeoffs and negative scaling behavior.

2. Related work and positioning

2.1. Coded computing and heterogeneous workers

Foundational coded-computing work shows how redundancy can mitigate stragglers in distributed learning and matrix computation [4-6]. Heterogeneity-aware coded computation further recognizes that workers may have different computation speeds and should not always receive equal load [14]. In mobile and edge settings, coded computing has also been combined with resource allocation and learning models [15]. These works motivate coded execution but also make clear that neither straggler mitigation nor heterogeneous assignment is new.

2.2. Sparse, rateless, and low-weight computation

Sparse coded transforms and sparse MVM have been investigated in several forms. Short-Dot computes large linear transforms using coded short dot products [7]. Universal-decodable-matrix designs target sparse MVM recoverability [9]. Lin et al. study coded multiplication for matrices with different sparsity levels and include structured assignment ideas [10]. Zhang et al. propose sparse and flexible coded computation for distributed learning [11]. Karami et al. study minimum-cost encoding for coded distributed computing systems [16]. Numerical stability is another separate concern [17]. Rateless MVM is especially important because it already uses LT-like ideas and peeling to balance load across workers [8, 18]. Low-weight encodings by Das et al. are an even closer neighbor: they study sparse distributed matrix computations with low-weight encodings, straggler-resilience guarantees, partial-computation extensions, and AWS experiments [12]. Das and

Ramamoorthy also treat partial stragglers and sparse matrices together, showing that slow workers need not be discarded as pure erasures [19]. Most importantly for this manuscript, Das et al. directly study sparsity-preserving straggler-optimal distributed matrix computations at the edge and report AWS validation [13].

2.3. Resulting boundary

This work is positioned as a reproducible resource-accounting study of a pre-stored systematic design under edge-like storage and execution constraints. It does not introduce a new sparse coded MVM method, a novel low-weight encoding scheme, or a straggler-optimal edge coding design, nor does it replicate or extend rateless or Das-style baselines. Instead, the focus is on evaluating system behavior under explicit resource constraints rather than improving coding-theoretic constructions. Table 1 summarizes the positioning of this work relative to representative prior studies across resource-accounting dimensions rather than coding-family novelty. The comparison highlights which system-level constraints are explicitly modeled and evaluated in this study.

Table 1. Positioning of this work under resource accounting dimensions

Work	Off/on split	Worker budget	Amort. comm.	Enc. accum.	Enc. nnz	Failure reasons	Timeout metric	CSV/ config	Boundary characterization
Mallick et al. [8]	partial	--	--	--	partial	--	partial	--	Rateless MVM baseline; LT-like component used only as internal ablation
Das et al. [12]	partial	partial	--	partial	√	partial	partial	--	Low-weight coding theory with empirical validation; dense decoding used only as ablation
Das & Ramamoorthy [19]	--	--	--	partial	√	partial	partial	--	Partial-straggler model; only completion events are tracked
Das et al. [13]	partial	√	--	partial	√	partial	partial	--	Edge sparsity-preserving code; used here as simplified baseline
This work	√	√	√	√	√	√	√	√	Resource-accounting benchmark for fixed systematic designs under edge constraints

Note: "Partial" indicates that the corresponding aspect appears in prior work but is not a primary evaluation axis. "--" indicates that the metric is not reported as a central benchmark dimension.

3. System and resource model

3.1. Sparse MVM and pre-stored encoded blocks

The master must compute $y = Ax$, where $A \in R^{Kb \times d}$ is partitioned into K row blocks $A_k \in R^{b \times d}$. The corresponding block outputs are:

$$y_k = A_k x, \quad k = 1, \ldots, K. \tag{1}$$

The master constructs $M = K + R$ encoded blocks using a sparse generator matrix $G \in R^{M \times K}$:

$$\widetilde{A}_j = \sum_{k \in S_j} g_{j,k} A_k \tag{2}$$

where $S_j = \{k : g_{j,k} \neq 0\}$. The row weight is $w_j = |S_j|$, with $w_j \leq w$ for the low-weight design. Coefficients are drawn from $\{+1, -1\}$ in the implemented simulator.

In the offline phase, encoded blocks are pre-stored at workers. In the online phase, the master broadcasts the input vector x , workers compute $y_j = A_j x$, and the master decodes once a sufficient number of results are collected. In this work, edge storage constraints refer to finite per-worker pre-stored encoded blocks, heterogeneous worker speeds and communication links, reusable offline placement, broadcast-based vector dissemination, and deadline-constrained execution. They do not include modeling of wireless contention, queueing effects, mobility, packet loss, or device-level energy heterogeneity.

3.2. Storage and amortized communication

Let $n_j = nnz(A_j)$. With sparse storage using a value and an index for each nonzero, the encoded block footprint is:

$$B_j^{store} = n_j(b_v + b_i) \tag{3}$$

where b_v and b_i are the bytes for a stored value and sparse index. If worker i stores task set J_i , then

$$B_i^{store} = \sum_{j \in J_i} B_j^{store}, \quad B_i^{store} \leq B_i^{max} \tag{4}$$

Tasks that cannot be placed under the budget are marked unassigned; the trial is not hidden. For online communication, the vector is counted once per assigned worker, not once per task. If W_a is the set of workers with at least one placed task, $D_x = db_v$, and $D_y = bb_v$, then

$$B^{on} = |W_a| D_x + m_{return} D_y \tag{5}$$

where m_{return} is the number of returned coded results. If a placement is reused for U vectors, the amortized communication is

$$B^{amort} = B^{on} + \frac{\sum_i B_i^{store}}{U} \tag{6}$$

3.3. Worker time, failures, and energy proxy

Worker i has compute rate c_i , downlink rate $r_i^{\downarrow}$, and uplink rate $r_i^{\uparrow}$. In the simulator these are log-normal random variables around configured base rates. A worker-level straggler delay is drawn as

$$S_i = Z_i E_i, \quad Z_i \sim Bernoulli(p_s), \quad E_i \sim Exp(1/\lambda_s) \tag{7}$$

and worker failure is $F_i(p_f)$. A failed worker returns no tasks. For a nonfailed worker processing its assigned tasks in order, task j finishes at

$$T_{i,j} = S_i + \frac{D_x}{r_i^{\downarrow}} + \sum_{l \preccurlyeq_i j} \frac{nnz(\tilde{A}_l)}{c_i} + \frac{D_y}{r_i^{\uparrow}} \tag{8}$$

For a worker assigned multiple tasks, the vector downlink term is paid once at the beginning of that worker's timeline; it appears in each task finish time only because each finish time is measured from job start. It is not charged once per task. Let $t^{\star}$ be the first collection time at which the decoder succeeds using the returned prefix. The reported job latency includes local decoding time,

$$T_{total} = T_{coll} + T_{dec} = t^{\star} + \frac{C_{dec}}{c_0} \tag{9}$$

where c_0 is the configured master decoding rate. If no returned prefix decodes, $T_{total} = \infty$. For a deadline D , the evaluation also reports the deadline capped time

$$T_D = \begin{cases} T_{\text{total}}, & \text{if decoding succeeds and } T_{\text{total}} \le D, \\ D, & \text{otherwise.} \end{cases} \tag{10}$$

This metric is a timeout capped mean, not an infinite penalty for failed jobs. The evaluation therefore also reports ($T_{total} \le D, success$). The simulator also records a normalized energy proxy for internal resource accounting only. This proxy is not calibrated to a measured hardware platform and is therefore not used as a main result.

$$E = e_{op} \sum_j nnz(\widetilde{A}_j) + e_{\downarrow} |W_a| D_x + e_{\downarrow} \frac{\sum_i B_i^{store}}{U} + e_{\uparrow} m_{return} D_y. \tag{11}$$

The current coefficients are simulation parameters, not measurements from a specific device platform. This proxy is therefore not used as a main result.

3.4. Resource accounting protocol

Every method is evaluated with the same accounting protocol. The reported vector includes recovery, latency, storage, communication, computation, decoding, and failure-mode quantities:

$$z_{acct} = (P_{succ}, T_D, B^{store}, B^{amort}, C_{enc}, N_{enc}, C_{worker}, C_{dec}, F_{reason}). \tag{12}$$

where C_{enc} is source-accumulation encoding work, N_{enc} is the materialized encoded-block nonzero count, and F_{reason} records storage infeasibility, insufficient returns, coverage failure, peeling stopping sets, rank failure, or other failure. This work does not solve a global multi-objective optimization. Instead, fixed operating points are evaluated under this protocol so that feasibility and negative results remain visible.

4. Low-weight mechanism

4.1. Systematic low-weight generator

The implemented low-weight code is systematic:

$$G = [\begin{matrix} I_K \\ P \end{matrix}], \tag{13}$$

where each row of P has at most w nonzeros. When $R_w \ge K$, the default constructor first allocates coverage slots so each source block appears in at least one redundant row, then fills the remaining row supports by random sampling. The benchmark also evaluates balanced cyclic and greedy low-overlap support designs. Coefficients are independently selected from $\{+1, -1\}$. This is a simple construction and does not meet the lower bound in [13].

Algorithm 1. Systematic bounded row weight support construction

Require: K, R, row weight w, support design.
Ensure: Generator rows $G = [I_K; P]$.
1. Initialize K systematic rows and R empty parity supports.
2. **If** $R_w \geq K$:
3. Assign each source block to one parity support for first-pass coverage.
4. **End if**
5. Fill every parity support to size w using random, balanced, or greedy support design.
6. Draw each nonzero coefficient from $\{+1, -1\}$.
7. **Return** $G = [I_K; P]$.

4.2. Ability-aware placement

Each worker receives a non-oracle score:

$$q_i = \frac{0.7(c_i/\bar{c})+0.3(r_i/\bar{r})}{1+\hat{s}_i}, \tag{14}$$

where $r_i = (r_i^{\downarrow} + r_i^{\uparrow})/2$ and s_i is an estimated historical risk score generated independently of the realized straggler delay. The simulator converts scores into integer quotas by flooring proportional allocations and assigning remaining tasks to the highest residual scores. Score and residual-quota ties are broken deterministically by ascending worker index. Encoded tasks are sorted by decreasing nonzero count, with equal-size tasks ordered by encoded row ID, and placed into quota-expanded worker order. If the preferred worker exceeds its storage budget, the task is tried on the remaining workers in deterministic order; otherwise it is counted as unassigned.

Algorithm 2. Ability-aware storage-constrained placement

Require: Encoded block sizes B_j^{store} , worker scores, storage budgets.
Ensure: Placement map J_i and unassigned task set.
1. Compute non-oracle worker scores q_i .
2. Convert scores into integer quotas by proportional flooring.
3. Distribute remaining quota slots to the largest residual scores; break ties by ascending worker index.
4. Sort encoded tasks by decreasing $n_j = nnz(\widetilde{A}_j)$; break equal-size ties by encoded row ID.
5. **For** each task j in sorted order:
6. Try the quota-preferred worker first, then remaining workers in deterministic score order with worker-index tie-breaking.
7. Place j at the first worker whose storage remains within budget.
8. If no worker fits, mark j as unassigned.
9. **End for**

This placement rule is an engineering heuristic. To avoid attributing all gains to it, the simulator also supports random, speed-only, no-risk, risk-only, storage-first, and a clairvoyant lower-bound diagnostic. The clairvoyant diagnostic uses realized worker failure and delay information and is reported only as an unattainable lower bound on latency.

4.3. Peeling decoder and stopping sets

The decoder first records returned systematic singleton equations. For a redundant equation, it subtracts all known source block results. If only one unknown remains,

$$y_k = \frac{1}{g_{j,k}} \left(\tilde{y}_j - \sum_{l \neq k} g_{j,l} y_l \right) \tag{15}$$

where a coefficient of -1 is explicitly divided out. The process repeats until all K source results are known or no singleton equation remains. A peeling failure corresponds to a stopping set: a nonempty group of missing source blocks such that every received redundant equation touching the group has residual degree at least two.

Algorithm 3. Online execution and peeling recovery

Require: Placement map, vector x , encoded rows, returned task stream.
Ensure: Recovered $y = Ax$ or a failure reason.
1. Broadcast x once to every worker with at least one placed task.
2. Workers compute placed tasks sequentially and return finished results unless failed.
3. Sort returned task events by finish time.
4. **For** each returned prefix:
5. Record systematic singleton results.
6. Subtract known terms from redundant equations.
7. If an equation has one unknown block, divide by its coefficient and recover that block.
8. Stop when all K blocks are recovered or no singleton remains.
9. **End for**

4.4. Heuristic stopping-set diagnostic

For a missing source set U with $|U| = s$, a received parity row of fixed weight w is immediately useful to peeling if it intersects U in exactly one block. Under an idealized random-support approximation, the probability of this event is:

$$p_1(s) = \frac{\binom{s}{1}\binom{K-s}{w-1}}{\binom{K}{w}}. \tag{16}$$

If q independent parity rows are received and placement correlations are ignored, the probability that none reveals a singleton is approximately $(1 - p_1(s))^q$. Let p_s^{sub} denote the probability assigned to a particular missing subset of size s under this idealized diagnostic model. A loose union-bound-style interpretation is therefore

$$P_{stop} \lesssim \sum_{s=1}^{K} p_s^{sub} \binom{K}{s} (1 - p_1(s))^q. \tag{17}$$

This expression is not a theorem for the implemented placement because systematic arrivals, worker failures, storage budgets, and support balancing create dependencies. It is used only as a diagnostic explanation, and p_s^{sub} should not be read as the aggregate probability $Pr(|U| = s)$. In the CSV output, residual stop blocks are defined as the number of unrecovered source blocks when peeling halts with no residual singleton equation. This is not a stopping-set probability; it is an empirical failure-pattern statistic that can be compared across support designs and row weights, not a predictor of exact recovery probability. The row-weight, support-design, and same-support decoder diagnostics compare this statistic with observed recovery behavior; these checks support trend interpretation but do not calibrate an exact success-probability model.

4.5. Complexity accounting

Let $n_k = nnz(A_k)$ and $n_j = nnz(A_j)$. Offline encoding cost for a coded row is modeled by the number of nonzero source entries accumulated:

$$C_{enc} \approx \sum_{j:|S_j|>1} \sum_{k \in S_j} n_k. \tag{18}$$

Thus, a dense parity row scales with $\sum_{k=1}^{K} n_k$, while a low-weight row scales only with the selected support. Worker cost is:

$$C_{worker} = \sum_{j \in returned} nnz(\widetilde{A}_j). \tag{19}$$

The two quantities are intentionally different. If three sparse source blocks are accumulated into one parity block, C_{enc} counts all traversed source nonzeros in the three blocks. The materialized encoded nonzero count n_j counts the remaining nonzeros after combining equal coordinates and possible $\{+1, -1\}$ cancellation. This is why a dense-coded row can have much larger source-accumulation cost than its final encoded nonzero count. Peeling cost is

$$C_{peel} = O(b \sum_{j \in R} |S_j|), \tag{20}$$

where R is the received equation set. Dense decoding over m received equations, where m is the number of equations used by the decoder and should not be confused with $M = K + R$, is counted as

$$C_{dense} = O(Km(K + b)), \tag{21}$$

which reduces to $O(K^3 + K^2 b)$ when $m = K$.

5. Simulation design

5.1. Configuration and metrics

The baseline configuration is summarized in Table 2. Unless otherwise stated, each manuscript-facing scenario is repeated for 1,000 seeds. All reported results are generated from the same archived benchmark record set. The 8, 16, 32, and 64 KiB storage budgets are tight-storage stress levels, not measured limits from a specific hardware platform. Matrix Market cases are structural stress tests: each coordinate (r, c) is mapped to the configured $Kb \times d$ shape by ($r \cdot Kb$, c, d), and empty target rows are filled with one random nonzero. This preserves some row-degree and clustering stress, but it changes the original dimensions and is not a faithful application trace. The $K/R/w$ design grid is reported as a reduced 300-seed diagnostic due to computational cost and is used only for design interpretation.

Table 2. Main simulation parameters

Parameter	Value
Workers and blocks	N = 20, K = 30, R = 10, M = 40
Block and vector size	b = 16, d = 64
Default row weight	w = 3
Matrix model	Bernoulli sparse, density 0.10
Base compute/link rates	4.0×10^6 ops/s; 8.0×10^6 bytes/s
Straggler/failure	p_s = 0.20, p_f = 0.02
Straggler delay scale	Exponential scale 0.20 s

Table 2. Continued

Heterogeneity	log-normal scale 0.60
Placement score weights	0.7 compute + 0.3 link, with risk penalty
Reuse/deadline	U = 100, D = 1.0 s
Storage unit	8-byte value + 4-byte index
Storage stress budgets	8, 16, 32, 64 KiB
Master decode rate	2.0× 10^7 ops/s

The methods compared in this study include uncoded systematic execution and simple replication. It also includes dense systematic coded execution with dense decoding. In addition, dense-budgeted execution is considered, which uses the same dense systematic code but places systematic rows first and then assigns only dense parity rows that satisfy the storage budget. This is treated as a fixed-code placement heuristic rather than an optimized redesign under storage constraints. The comparison further includes the proposed systematic low-weight design with peeling. An LT-like block-level ablation is also included, which does not correspond to the exact Mallick et al. rateless MVM. Finally, a dense-decoded low-weight ablation is considered, which does not reproduce the construction in Das et al.

Success rates are computed over all seeds. Mean and P95 latency are reported over successful finite trials only. Timeout mean latency uses T_D with $D = 1.0s$ over all trials, and success by deadline is reported separately. A lower T_D must therefore be interpreted jointly with success rate. Bootstrap intervals are computed for mean/P95 latency; Wilson intervals are computed for success rates.

5.2. Baseline results

Tables 3 to 6 summarize the baseline results at $w = 3$. In this setting, dense coded computation achieves the lowest success-only latency because decoding is triggered after exactly 30 successful returns. However, this advantage comes at the cost of significantly higher storage consumption, worker-side nonzero operations, and encoding accumulation. By contrast, the proposed low-weight design increases latency moderately but substantially reduces worker computation overhead.

In addition to latency behavior, communication patterns remain largely consistent across coded methods. This is because encoded blocks are pre-stored offline, and online traffic is mainly composed of a single vector broadcast per active worker together with returned computation results. Therefore, communication differences are primarily driven by placement rather than encoding transmission.

Table 3. Performance metrics for key methods (w = 3)

Method	Succ. [CI]	Succ. by D	Mean T [CI]	P95 T [CI]	T_D
Uncoded	0.657 [0.627,0.686]	0.642	0.3956 [0.3790,0.4142]	0.8649 [0.8168,0.9268]	0.6010
Dense coded	1.000 [0.996,1.000]	1.000	0.0189 [0.0166,0.0214]	0.1012 [0.0873,0.1206]	0.0189
Low-weight	0.955 [0.940,0.966]	0.953	0.1632 [0.1533,0.1736]	0.5086 [0.4738,0.5351]	0.2003

Note: Latencies are success-only except T_D , which is computed under a 1.0 s timeout across all trials.

Table 4. Resource usage ($w = 3$)

Method	Worker nnz	Enc. accum.	Enc. nnz	Dec. ops	Store KiB
Uncoded	3,009.7	0.0	0.0	624.8	36.0
Dense coded	12,609.7	30,731.1	9,805.5	41,400.0	150.9
Low-weight	5,727.2	3,073.1	2,776.3	1,262.8	68.5

Note: Encoding accumulation reflects source-level nonzero traversal, while encoded nnz reflects post-cancellation sparsity.

Table 5. Communication and decoding diagnostics ($w = 3$)

Method	Store KiB	Max KiB	Online KiB	Amort. $U = 1$	Amort. $U = 100$
Uncoded	36.0	3.5	13.7	49.7	14.0
Replication	48.0	2.7	14.9	62.9	15.4
Dense coded	150.9	12.9	14.9	165.8	16.4
Low-weight	68.5	9.1	14.9	83.4	15.6

Besides, online communication is similar across coded methods because encoded matrix blocks are pre-stored. The online traffic is dominated by one vector broadcast per active worker and returned block results, not by transmitting encoded matrices.

Table 6. Decode-time diagnostics

Method	$T_$coll	$T_$dec	Returned	Used	Critical nnz
Uncoded	0.3956	0.000024	29.4	29.4	151.9
Replication	0.3675	0.000031	39.2	38.5	166.1
Dense coded	0.0168	0.002070	39.2	30.0	894.6
Low-weight	0.1631	0.000062	39.2	35.7	268.0
LT-like abl.	0.2156	0.000084	39.2	36.3	268.9
Dense-dec. LW	0.1631	0.002461	39.2	35.7	268.0

Note: "LT-like" is a block-level ablation and does not correspond to Mallick et al.; "Dense-dec. LW" uses dense decoding over the same low-weight placement.

5.3. Edge resource results

Table 7 separates the unconstrained storage setting from the 8 KiB per-worker budget scenario in order to avoid conflating maximum-capacity behavior with feasibility under tight storage limits. Under the unconstrained setting at $w = 3$, dense coding requires about 150.9 KiB total storage with a 12.9 KiB maximum per worker, whereas the low-weight design uses 68.5 KiB total storage with a 9.1 KiB maximum per worker.

When the storage budget is reduced to 8 KiB per worker, the low-weight design can still be placed under the same deterministic placement rule after minor reallocation, which keeps all tasks feasible while slightly reducing the effective per-worker footprint. In contrast, dense and dense-budgeted configurations cannot fully place dense parity blocks under this constraint. As a result, unassigned tasks appear and the system falls back

to partial systematic recovery behavior, with an observed success rate of 0.657. This is therefore treated as a stress-test outcome of a fixed design rather than a general failure of dense coding under all configurations.

Table 7. Storage slice ($w = 3$, unconstrained vs. 8 KiB budget)

Method	Total KiB	Max KiB	8 KiB total	8 KiB max	8 KiB unassigned	All placed	8 KiB success
Uncoded	36.0	3.5	36.0	3.5	0.0	1.000	0.657
Replication	48.0	2.7	48.0	2.7	0.0	1.000	0.711
Dense coded	150.9	12.9	36.0	2.6	10.0	0.000	0.657
Dense-budgeted	150.9	12.9	36.0	2.6	10.0	0.000	0.657
Low-weight	68.5	9.1	68.5	7.2	0.0	1.000	0.955
LT-like abl.	79.5	9.2	67.7	7.7	1.5	0.206	0.818
Dense-dec. LW	68.5	9.1	68.5	7.2	0.0	1.000	0.955

Energy-related quantities are not included in the main results tables. Though the simulator records a normalized energy proxy for debugging purposes, its coefficients are not calibrated to any specific hardware platform such as ESP32, Raspberry Pi, Wi-Fi, BLE, or LoRa. Therefore, it should not be interpreted as a physically measured energy consumption value.

5.4. Failure accounting

Table 8 reports failure modes for the unconstrained baseline setting at $w = 3$, and it is separate from the 8 KiB storage-stress results in Table 7. In this setting, uncoded execution fails mainly due to insufficient returned results caused by worker failures or slow returns, while replication failures are dominated by coverage loss where at least one source block is never observed. In contrast, low-weight decoding and the LT-like ablation primarily fail due to peeling stopping sets, and the dense-decoded low-weight variant fails due to rank deficiency under the same received support pattern.

Table 8. Failure reasons (1,000 seeds, $w = 3$)

Method	Worker/insuff.	Coverage	Stop set	Rank	Storage
Uncoded	343	0	0	0	0
Replication	0	289	0	0	0
Dense coded	0	0	0	0	0
Low-weight	0	0	45	0	0
LT-like abl.	0	0	117	0	0
Dense-dec. LW	0	0	0	45	0

5.5. Placement ablation

Table 9 isolates the effect of the placement rule for the low-weight design while keeping the support structure and workload unchanged. The ability-aware heuristic improves feasibility compared with naive baselines, but the differences among variants remain small. In particular, no-risk and risk-only variants slightly reduce timeout latency, while speed-only, random, and storage-first rules introduce minor degradations in either success rate or tail latency. The clairvoyant lower-bound setting performs best but relies on realized worker outcomes and therefore cannot be used in an online setting. The results indicate that placement mainly affects

marginal latency and robustness rather than the fundamental behavior of the coding scheme, so it is treated as an engineering heuristic rather than an optimized heterogeneous allocation method.

Table 9. Low-weight placement ablation (1,000 seeds)

Placement	Succ.	Succ. by D	Mean T	T_D	Resid. stop
Ability-aware	0.955	0.953	0.1632	0.2003	0.090
No-risk	0.955	0.953	0.1595	0.1968	0.090
Risk-only	0.963	0.961	0.1639	0.1944	0.073
Speed-only	0.958	0.953	0.1625	0.1964	0.088
Random	0.949	0.945	0.1725	0.2139	0.080
Storage-first	0.941	0.940	0.1718	0.2207	0.117
Clairvoyant LB	1.000	1.000	0.0007	0.0007	0.000

5.6. Decoder diagnostics

The decoder diagnostics evaluate peeling recovery and dense rank recovery on the same support structure, placement, and returned execution traces, as shown in Table 10. And this avoids inconsistencies caused by comparing independently generated runs. Across all tested settings, whenever peeling decoding succeeds, the corresponding structural rank condition is also satisfied, and no cases are observed where peeling succeeds under rank deficiency. The remaining failures correspond to genuine stopping-set effects in low-weight decoding under identical support patterns. The smaller diagnostic sweeps over different (K, ρ, w) configurations show the same consistency pattern, indicating that the observed relationship is stable across scaling regimes.

Table 10. Same-support decoder diagnostics (low-weight)

Scenario	Seeds	Peel succ.	Dense-rank succ.	Rank viol.
$w = 3$ baseline	1,000	955	955	0
$K = 30$, $\rho = 1/3$, $w = 3$	300	288	288	0
$K = 60$, $\rho = 1/3$, $w = 3$	300	277	277	0
$K = 120$, $\rho = 1/3$, $w = 3$	300	259	259	0

5.7. Parameter sweeps

Figure 1 shows the row-weight tradeoff. Increasing w improves success and latency in this configuration, but also increases worker nonzero operations and encoding accumulation. Thus, the phrase "low-weight" should not be interpreted as "as sparse as possible"; the useful region depends on recovery needs and resource budgets.

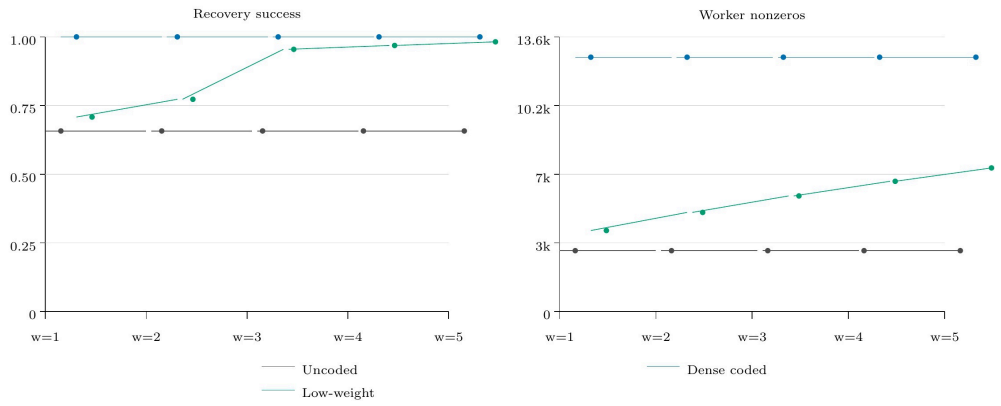

Figure 1. Effect of row weight on recovery success and worker nonzero operations

Figure 2 summarizes straggler and storage budget stress dimensions. Success falls as straggler probability rises, but low-weight coding retains higher success than uncoded, replication, and the LT-like ablation across the shown range. Storage budget feasibility highlights the main advantage over dense coding under tight per-worker placement limits.

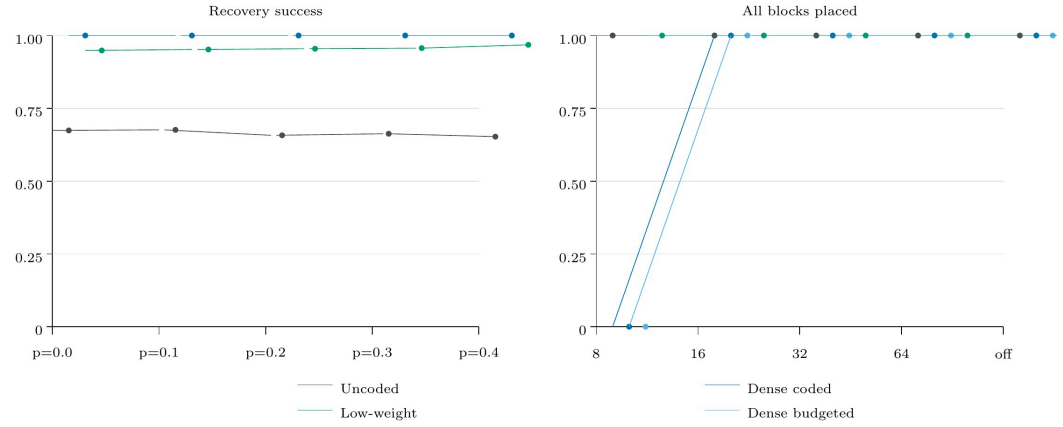

Figure 2. Straggler and storage budget sweeps

Figure 3 presents the support-design ablation and workload sensitivity, with workload definitions and matrix statistics listed in Table 11. A greedy low-overlap support construction improves baseline low-weight success from 0.955 to 0.964. Across synthetic and Matrix Market projected stress cases, success varies between 0.950 and 0.983, indicating that workload structure and support construction jointly affect performance, while the overall behavior of the scheme remains stable as a baseline design rather than an optimized code family.

Figure 3. Support-design ablation and workload sweep

W-labels are defined in Table 11.

Table 11. Low-weight workload slice

ID	Workload	Type	Density	Row mean	Row p95	Succ.	T_D
W1	Bernoulli	synthetic	0.100	6.4	10.5	0.955	0.200
W2	Banded graph	synthetic	0.094	6.0	6.0	0.978	0.165
W3	Block diagonal	synthetic	0.094	6.0	6.0	0.983	0.164
W4	Power-law graph	synthetic	0.106	6.8	19.5	0.950	0.209
W5	Clustered sparse	synthetic	0.094	6.0	6.0	0.977	0.165
W6	bcsstk01	Matrix Market	0.021	1.4	4.0	0.959	0.193
W7	nos4	Matrix Market	0.024	1.5	6.0	0.977	0.163
W8	west0067	Matrix Market	0.023	1.5	5.0	0.976	0.163
W9	gre_115	Matrix Market	0.025	1.6	4.0	0.974	0.166
W10	arc130	Matrix Market	0.049	3.1	5.0	0.957	0.200
W11	ash219	Matrix Market	0.023	1.5	2.0	0.982	0.158
W12	494_bus	Matrix Market	0.035	2.2	4.0	0.951	0.207
W13	1138_bus	Matrix Market	0.076	4.9	8.0	0.959	0.189

Matrix Market cases are projected structural stress tests rather than actual application traces. Source files are listed in the repository source log.

Figure 4 shows sparsity and reuse effects. As matrix sparsity decreases, worker nonzero operations rise for all coded methods, which is expected because each encoded block becomes less sparse. With one reuse, dense coding has much larger amortized communication than low-weight coding because the offline placement term is not amortized; with 1,000 reuses, both methods are dominated by online vector and result traffic.

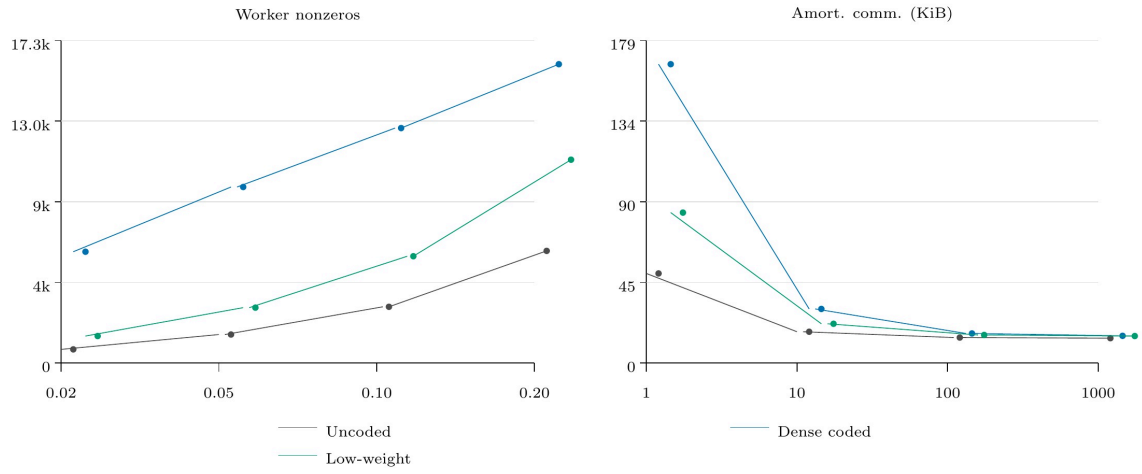

Figure 4. Sparsity and reuse-count effects

5.8. Scaling and $K/R/w$ diagnostics

Table 12. Scaling checks with zero encoding accumulation for uncoded execution

Sweep	Method	Succ.	Mean T	T_D	Worker nnz	Enc. accum.
$K = 30$	Uncoded	0.657	0.3956	0.6010	3,009.7	0.0
$K = 30$	Dense coded	1.000	0.0189	0.0189	12,609.7	30731.1
$K = 30$	Low-weight	0.955	0.1632	0.2003	5,727.2	3073.1
$K = 60$	Uncoded	0.657	0.3957	0.6010	6,015.0	0.0
$K = 60$	Dense coded	1.000	0.0306	0.0306	26,033.8	122872.0
$K = 60$	Low-weight	0.912	0.2177	0.2858	11,451.9	6143.6
$K = 120$	Uncoded	0.657	0.3958	0.6011	12,033.4	0.0
$K = 120$	Dense coded	1.000	0.1153	0.1153	52,140.7	491588.3
$K = 120$	Low-weight	0.865	0.2641	0.3610	22,911.2	12289.7
$d = 256$	Uncoded	0.657	0.3959	0.6012	12,033.1	0.0
$d = 256$	Low-weight	0.953	0.1674	0.2057	22,898.7	12286.7
$d = 256$	Dense coded	1.000	0.0200	0.0200	50,436.3	122866.8
$d = 1024$	Uncoded	0.657	0.3972	0.6020	48,137.5	0.0
$d = 1024$	Low-weight	0.953	0.1720	0.2103	91,601.7	49152.1
$d = 1024$	Dense coded	1.000	0.0245	0.0245	201,743.2	491521.0

Table 12 demonstrates 1,000-seed scaling checks. Increasing the vector dimension d primarily increases worker nonzero operations, while low-weight success remains close to the baseline. Increasing K is more challenging: with fixed $w = 3$ and $R/K \approx 1/3$, low-weight success decreases from 0.955 at $K = 30$ to 0.865 at $K = 120$. In addition, Table 13 shows a reduced 300-seed diagnostic grid focusing on low-weight configurations. Larger R/K ratios and higher row weights w improve success, but also increase worker nonzero operations. These results highlight a tradeoff: the current random or balanced support construction

should not be interpreted as a fully scalable recovery design without additional redundancy or optimized support selection.

Table 13. Reduced *K/R/w* low-weight diagnostic grid with 300 seeds and success, timeout-capped mean, and worker nonzero counts

Setting	$w = 2$	$w = 3$	$w = 5$
K =30, ρ = 1/3	.783 / .452 / 4906	.960 / .193 / 5714	.973 / .139 / 7097
K =60, ρ = 1/3	.767 / .472 / 9823	.923 / .277 / 11440	.973 / .195 / 14210
K = 120, ρ = 1/3	.743 / .505 / 19638	.863 / .364 / 22875	.940 / .253 / 28409
K = 120, ρ = 1/2	.917 / .308 / 23454	.957 / .238 / 28324	.983 / .129 / 36650

6. Discussion

The current evidence supports a restrained claim. Dense coded computation is the fastest implemented method when storage and dense decoding cost are acceptable; however, this advantage comes with higher storage demand and significantly larger encoding and decoding overhead. In contrast, the low-weight systematic design occupies a more resource-efficient region, since it reduces pre-storage, encoded nonzero operations, source-accumulation encoding cost, and decoding complexity, while trading off higher latency and lower success rate compared with dense coding under the same configuration.

At the same time, the achievable claims are constrained by prior work. Das et al. already provide sparsity-preserving straggler-optimal edge encodings with AWS validation [13] and also provide low-weight encodings with theoretical and experimental evidence [12]. Mallick et al. present rateless coded MVM with a more faithful load-balancing design than the LT-like ablation used here [8]. As a result, the comparisons in this manuscript should be interpreted as controlled internal baselines and stress tests rather than evidence that those prior methods are inferior or underperforming.

The placement and scaling results further refine this interpretation. Although ability-aware placement provides a reasonable heuristic under heterogeneous workers, it is not consistently optimal across simple alternatives, which suggests it should be treated as a transparent baseline rather than a core contribution. Similarly, the dense-budgeted variant does not recover the performance of dense coding under tight storage constraints because parity blocks remain infeasible to place once budgets are enforced, which reflects a fixed-design limitation rather than a general failure of dense coding. Moreover, scaling experiments show that a fixed configuration such as $w = 3$ and $R/K \approx 1/3$ is insufficient to maintain recovery performance as problem size increases, since success degrades noticeably with larger K even when other parameters are unchanged. Taken together, these results indicate that improvements are likely to come from stronger support design, additional redundancy, or closer alignment with prior optimized constructions.

Besides, several validity constraints should be noted in a unified view. The energy model is purely a simulation proxy and is not calibrated to specific hardware platforms such as embedded or wireless IoT devices, and therefore it is not used as a primary quantitative conclusion. Internal consistency is supported through deterministic seeds, confidence intervals, and same-support decoding diagnostics, although modeling assumptions and implementation artifacts may still influence outcomes. External validity is also limited because the workloads are projected structured matrices rather than full production traces, even though they provide more diversity than purely synthetic Bernoulli cases. In addition, several baselines are intentionally simplified or ablated versions of prior work rather than faithful reproductions, and the system model does not

include network contention, queueing effects, mobility, or partial computation completion, which further limits direct real-world generalization.

7. Conclusion

This paper evaluates a simple systematic bounded row-weight coded MVM mechanism under edge storage constraints. In the simulated settings, the method is not latency-optimal compared with dense coding, but it reduces pre-storage footprint, encoded nonzero operations, source-accumulation encoding work, and decoding effort, while remaining feasible under tight storage stress scenarios where dense parity blocks cannot be placed. Ablation studies further highlight the baseline's limitations: ability-aware placement is a heuristic, support design influences recovery, and success declines as K increases under fixed low weight and redundancy ratios. The main contribution is thus the resource-accounting framework and a reproducible lightweight baseline, rather than a new optimal sparse code family. Future work should incorporate faithful rateless and low-weight literature implementations, partial straggler modeling, device-calibrated energy and wireless parameters, and analyses of recovery under enhanced support constructions.

References

[1] Singh, R., & Gill, S. S. (2023). Edge AI: A survey. *Internet of Things and Cyber-Physical Systems, 3*, 71–92.

[2] Aouedi, O., Vu, T.-H., Sacco, A., Nguyen, D. C., Piamrat, K., Marchetto, G., & Pham, Q.-V. (2024). A survey on intelligent internet of things: Applications, security, privacy, and future directions. *IEEE Communications Surveys & Tutorials, 27*(2), 1238–1292.

[3] Zhu, G., Lyu, Z., Jiao, X., Liu, P., Chen, M., Xu, J., & Zhang, P. (2023). Pushing AI to wireless network edge: An overview on integrated sensing, communication, and computation towards 6G. *Science China Information Sciences, 66*(3), 130301.

[4] Lee, K., Lam, M., Pedarsani, R., Papailiopoulos, D., & Ramchandran, K. (2017). Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory, 64*(3), 1514-1529.

[5] Yu, Q., Maddah-Ali, M. A., & Avestimehr, A. S. (2020). Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding. *IEEE Transactions on Information Theory, 66*(3), 1920-1933.

[6] Yu, Q., Li, S., Raviv, N., Kalan, S. M. M., Soltanolkotabi, M., & Avestimehr, S. A. (2019, April). Lagrange coded computing: Optimal design for resiliency, security, and privacy. In *The 22nd International Conference on Artificial Intelligence and Statistics* (pp. 1215–1225). PMLR.

[7] Dutta, S., Cadambe, V., & Grover, P. (2019). "Short-Dot": Computing large linear transforms distributedly using coded short dot products. *IEEE Transactions on Information Theory, 65*(10), 6171–6193. https://doi.org/10.1109/TIT.2019.2927558

[8] Mallick, A., Chaudhari, M., Sheth, U., Palanikumar, G., & Joshi, G. (2020, June). Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems* (pp. 95–96).

[9] Cao, H., Yan, W., Lin, S.-J., & Zhang, W. (2022). A new coding scheme for matrix-vector multiplication via universal decodable matrices. In *2022 IEEE International Symposium on Information Theory (ISIT)* (pp. 572–577).

[10] Lin, J.-A., Huang, Y.-C., Lee, M.-C., & Chen, P.-N. (2024). Coded distributed multiplication for matrices of different sparsity levels. *IEEE Transactions on Communications, 72*(2), 633–647.

[11] Zhang, J., He, X., & Dai, H. (2025). Speeding up distributed learning via sparse and flexible coded computing. *IEEE Transactions on Information Theory, 71*(4), 3167–3180.

[12] Das, A. B., Ramamoorthy, A., Love, D. J., & Brinton, C. G. (2023). Distributed matrix computations with low-weight encodings. *IEEE Journal on Selected Areas in Information Theory, 4*, 363–378.

[13] Das, A. B., Ramamoorthy, A., Love, D. J., & Brinton, C. G. (2024). Sparsity-preserving encodings for straggler-optimal distributed matrix computations at the edge. *IEEE Internet of Things Journal, 11*(21), 34455-34470.

[14] Reisizadeh, A., Prakash, S., Pedarsani, R., & Avestimehr, A. S. (2019). Coded computation over heterogeneous clusters. *IEEE Transactions on Information Theory, 65*(7), 4227-4242.

[15] Qiu, H., Zhu, K., Niyato, D., & Tang, B. (2024). Resilient, secure, and private coded distributed convolution computing for mobile-assisted metaverse. *IEEE Transactions on Mobile Computing, 23*(12), 12892–12906.

[16] Karami, M., Ardakani, M., Ebrahimzad, H., & Zhang, Z. (2026). Minimum cost encoding for coded distributed computing systems. *IEEE Transactions on Communications, 74*, 6643–6656.

[17] Ramamoorthy, A., & Tang, L. (2021). Numerically stable coded matrix computations via circulant and rotation matrix embeddings. *IEEE Transactions on Information Theory, 68*(4), 2684–2703.

[18] Guo, Z., Ji, X., You, W., Guo, H., Zhang, Y., Zhao, Y., Xu, M., & Bai, Y. (2025). Workload distribution with rateless encoding: A low-latency computation offloading method within edge networks. *Computer Networks, 269*, 111381.

[19] Das, A. B., & Ramamoorthy, A. (2022). A unified treatment of partial stragglers and sparse matrices in coded matrix computation. *IEEE Journal on Selected Areas in Information Theory, 3*(2), 241–256.