Adaptive Coding for Matrix Multiplication at Edge Networks

Elahe Vedadi and Hulya Seferoglu
University of Illinois at Chicago
Email: {evedad2, hulya}@uic.edu

Abstract—**Edge computing is emerging as a new paradigm to allow processing data at the edge of the network, where data is typically generated and collected, by exploiting multiple devices at the edge collectively. However, exploiting the potential of edge computing is challenging mainly due to the heterogeneous and time-varying nature of edge devices. Coded computation, which advocates mixing data in sub-tasks by employing erasure codes and offloading these sub-tasks to other devices for computation, is recently gaining interest, thanks to its higher reliability, smaller delay, and lower communication cost. In this paper, our focus is on characterizing the cost-benefit trade-offs of coded computation for practical edge computing systems, and develop an adaptive coded computation framework. In particular, we focus on matrix multiplication as a computationally intensive task, and develop an adaptive coding for matrix multiplication (ACM^2) algorithm by taking into account the heterogeneous and time varying nature of edge devices. ACM^2 dynamically selects the best coding policy by taking into account the computing time, storage requirements as well as successful decoding probability. We show that ACM^2 improves the task completion delay significantly as compared to existing coded matrix multiplication algorithms.**

I. Introduction

Massive amount of data is generated at edge networks with the emerging Internet of Things (IoT) including self-driving cars, drones, health monitoring devices, etc. Transmitting such massive data to the centralized cloud, and expecting timely processing are not realistic with limited bandwidth between an edge network and centralized cloud. We consider a distributed computing system, where computationally intensive aspects are distributively processed at the end devices with possible help from edge servers (fog) and cloud. However, exploiting the potential of edge computing is challenging mainly due to the heterogeneous and time-varying nature of edge devices.

Coded computation is an emerging field, which studies the design of erasure and error-correcting codes to improve the performance of distributed computing through "smart" data redundancy. This breakthrough idea has spawned a significant effort, mainly in the information and coding theory communities [1], [2]. According to distributed computation, a *master* device divides computationally intensive aspects/tasks into multiple smaller sub-tasks, and offloads each of them to other devices (end devices, edge servers, and cloud), called *workers*, for computation. Coded computation (*e.g.,* by employing erasure codes such as Reed Solomon codes [3], [4]), on the other hand, encodes the data in the sub-tasks, and offloads these coded sub-tasks for computation. The next example demonstrates the potential of coded computation.

Example 1: Consider a setup where a master wishes to offload a matrix multiplication $C = A^T B$ task to three workers. Assume A and B are $K \times K$ matrices and matrix A is divided into two sub matrices A_1 and A_2, which are then encoded using a $(3, 2)$ Maximum Distance Separable (MDS) code, which is further explained in Section II, to give $Z_1 = A_1$, $Z_2 = A_2$ and $Z_3 = A_1 + A_2$, and sends each to a different worker. When the master receives the computed values (*i.e.,* $Z_i^T B$) from at least two out of three workers, it can decode its desired task, which is the computation of $A^T B$. The power of coded computations is that it makes $Z_3 = A_1 + A_2$ acts as an extra task that can replace any of the other two tasks if they end up straggling or failing. □

Significant effort is being put on constructing codes for fast and distributed matrix-vector multiplication [1], [5], matrix-matrix multiplication [6]–[9], dot product and convolution of two vectors [10], [11], gradient descent [12]–[14], distributed optimization [15], [16], Fourier transform [17], and linear transformations [18]. The trade-off between latency of computation and load of communication for data shuffling in MapReduce framework is characterized in [2], and optimum resource allocation algorithm is developed in [19]. This coding idea is extended for cellular networks [20], multistage computation [21], and heterogeneous systems [22], [23].

Our focus in this work is on matrix multiplication, where a master device divides its matrix multiplication computations into smaller tasks and assigns them to workers (possibly including itself) that can process these tasks in parallel. Product [7], polynomial [6], and MatDot (and its extension PolyDot) codes [8] are recently developed for matrix multiplication. Their main focus is to minimize/optimize the recovery threshold, which is the minimum number of workers that the master needs to wait for in order to compute matrix multiplication ($C = A^T B$ in Example 1). Although this metric is good for large scale computing systems in data centers, it fails short in edge computing, where other resources including computing power, storage, energy, networking resources are limited.

In this paper, we analyze the computing time, storage cost, and successful decoding probability of some existing codes for matrix multiplication. Then, we develop an adaptive coding for matrix multiplication (ACM^2) algorithm that selects the best coding strategy for each sub-matrix.

We note that rateless codes considered in [24], [25] also

This work was supported in parts by the Army Research Lab (ARL) under Grant W911NF-1820181, National Science Foundation (NSF) under Grants CNS-1801708 and CCF-1942878, and the National Institute of Standards and Technology (NIST) under Grant 70NANB17H188.

provide adaptive coded computation mechanisms against heterogeneous and time-varying resources. However, the coding overhead in rateless codes can be high in some scenarios, which makes adaptive selection of fixed-rate codes a better alternative. Multi-message communication by employing Lagrange coded computation is considered in [26] to reduce under-utilization due to discarding partial computations carried out by stragglers as well as over-computation due to inaccurate prediction of the straggling behavior. A hierarchical coded matrix multiplication is developed in [27] to utilize both slow and fast workers. As compared to [26], [27], we propose an adaptive code selection policy for heterogeneous and time-varying resources. A code design mechanism under a heterogeneous setup is developed in [22], [23], where matrix A is divided, coded, and offloaded to worker by taking into account heterogeneity of resources. However, available resources at workers may vary over time, which is not taken into account in [22], [23]. Thus, it is crucial to design a coded computation mechanism, which is dynamic and adaptive to heterogeneous and time-varying resources, which is the goal of this paper.

The following are the key contributions of this paper:

- We provide computing time analysis of some existing codes designed for matrix multiplication including product [7], polynomial [6], and MatDot codes [8].
- We characterize storage requirements of existing matrix multiplication codes [6]–[8] at master and workers.
- We design $\texttt{ACM}^2$ for an iterative procedure (*e.g.,* gradient descent) that selects the best code as well as the optimum number of partitions for that code at each iteration to minimize the average computing time subject to storage and successful decoding probability constraints.[1]
- We evaluate the performance of $\texttt{ACM}^2$ through simulations and show that $\texttt{ACM}^2$ significantly improves the average computing time as compared to existing codes.

II. Model, Background, and Motivation

A. Model

1) Setup: We consider a master/worker setup at the edge of the network, where the master device offloads its computationally intensive tasks (matrix multiplication computations) to workers w_n, $n \in \mathcal{N}$ $\big($where $\mathcal{N} \triangleq \{1, \dots, N\}$ $\big)$ via device-to-device (D2D) links such as Wi-Fi Direct and/or Bluetooth. The master device divides a task (matrix) into smaller sub-matrices, and offloads them to parallel processing workers.

2) Task Model: The master wants to compute *functions* of its collected data, which is determined by the applications. We will focus on computing linear functions; specifically matrix multiplication $C = A^T B$, where $A \in \mathbb{R}^{L\times K}$, $B \in \mathbb{R}^{L\times K}$. Matrix multiplication forms an essential building block of many signal processing (convolution, compression, etc.) and machine learning algorithms (gradient descent, classification, etc.) [1]. We consider an iterative procedure (gradient descent) where a matrix multiplication is calculated at each iteration.

3) Worker Model: The workers may experience: *(i) failures:* workers may fail or "sleep/die" or leave the network before finishing their assigned tasks. *(ii) stragglers:* workers will incur probabilistic delays in responding to the master.

4) Coding Model: We design and employ an adaptive coding for matrix multiplication ($\texttt{ACM}^2$) that selects the best coding strategy among repetition, MDS [1], [28], polynomial [6], MatDot [8], and product codes [7] by taking into account the computing time, storage cost, and successful decoding probability of these codes. The master device divides the matrix A into p_π partitions (sub-matrices), where $\pi \in \{\text{rep}, \text{mds}\}$ for repetition and MDS codes. Both A and B matrices are divided into p_π partitions, where $\pi \in \{\text{poly}, \text{matdot}, \text{pro}\}$ for polynomial, MatDot, and product codes.

5) Delay Model: Each sub-matrix transmitted from the master to a worker $w_n,\ n \in \mathcal{N}$, experiences the following delays: (i) transmission delay for sending the sub-matrix from the master to the worker, (ii) computation delay for computing the multiplication of the sub-matrices, and (iii) transmission delay for sending the computed matrix from the worker w_n back to the master. We model the composite delay using the shifted exponential distribution ($f(t) = \lambda e^{-\lambda(t-1)}$ for $t \geq 1$) [1], [29], with λ referred to as the straggling parameter and each sub-task with shifted-scaled exponential distribution ($f(t) = \alpha\lambda e^{-\alpha\lambda(t-\frac{1}{\alpha})}$ for $t \geq \frac{1}{\alpha}$) where the scale parameter, α is selected from $\alpha \in \{p_{\text{rep}}, p_{\text{mds}}, p_{\text{poly}}^2, p_{\text{matdot}}, p_{\text{pro}}^2\}$.

B. Background on Existing Codes for Matrix Multiplication

In this section, we provide a short background on existing coded computation mechanisms for matrix multiplication.

1) Repetition Codes: The master device divides matrix A column-wise into p_{rep} parts, where $p_{\text{rep}}|N$, and generates $\frac{N}{p_{\text{rep}}}$ copies of each sub-matrix. Sub-matrix A_i, $i = 1, \dots, p_{\text{rep}}$ is transmitted to $\frac{N}{p_{\text{rep}}}$ workers as well as matrix B. Workers calculate and return $A_i^T B$ to the master, which finishes matrix multiplication calculation when it receives $A_i^T B$, $\forall i = 1, \dots, p_{\text{rep}}$.

2) MDS Codes [1], [28]: The master device divides the matrix A column-wise to p_{mds} partitions. An (N, k_{mds}) MDS code sets $k_{\text{mds}} = p_{\text{mds}}$ and codes k_{mds} sub-matrices into N sub-matrices using existing MDS codes like Reed-Solomon codes. $A_i, \forall i = 1, \dots, N$ as well as B are transmitted to N workers. When the master device receives k_{mds} $A_i^T B$ calculations, it can decode and calculate $A^T B$.

3) Polynomial Codes [6]: The master device divides A and B column-wise into p_{poly} partitions, where $p_{\text{poly}}^2 \leq N$. The master constructs polynomials; $\alpha(n) = \sum_{j=1}^{p_{\text{poly}}} A_j^T n^{j-1}$ and $\beta(n) = \sum_{j=1}^{p_{\text{poly}}} B_j n^{(j-1)p_{\text{poly}}}$, and sends them to worker W_n, which calculates $\alpha(n)\beta(n)$. When the master receives $k_{\text{poly}} = p_{\text{poly}}^2$ $\alpha(n)\beta(n)$ multiplication, decoding is completed.

4) MatDot Codes [8]: Both matrices A and B are divided row-wise into p_{matdot} partitions, where $2p_{\text{matdot}} - 1 \leq N$. The master constructs the polynomials; $\alpha(n) = \sum_{j=1}^{p_{\text{matdot}}} A_j^T n^{j-1}$ and $\beta(n) = \sum_{j=1}^{p_{\text{matdot}}} B_j n^{p_{\text{matdot}}-j}$, and sends them to worker w_n for processing, where worker w_n calculates $\alpha(n)\beta(n)$ multiplication. When the master receives $k_{\text{matdot}} = 2p_{\text{matdot}} - 1$ results from workers, it can decode and calculate $A^T B$.

[1] We note that $\texttt{ACM}^2$ is generic enough to work with any matrix multiplication codes although we consider a subset of existing codes in this paper such as product, polynomial, and MatDot codes.

5) Product Codes [7]: Product codes extend MDS codes in a way that both A and B are partitioned column-wise and coded. In particular, both A and B are divided into p_{pro} partitions, and these partitions are put into $p_{\text{pro}} \times p_{\text{pro}}$ array. Then, every row of the array is encoded with an $(\sqrt{N}, p_{\text{pro}})$ MDS code, which results $p_{\text{pro}} \times \sqrt{N}$ array. This array is also coded with an $(\sqrt{N}, p_{\text{pro}})$ MDS code, which results into $\sqrt{N}$-by-$\sqrt{N}$ array. Each coded sub-matrix in this array is sent to a worker (out of N workers) for calculation. Product codes are decodable if at least one entry of any possible sub-array with size larger than or equal to $(\sqrt{N}-p_{\text{pro}}+1) \times (\sqrt{N}-p_{\text{pro}}+1)$ is received successfully.

C. Motivation for Adaptive Coding

Assume a canonical setup, where A and B are $K \times K$ matrices and divided into two sub-matrices A_0, A_1, B_0, and B_1. The product codes divide matrices column-wise, *i.e.,* A_i and B_i are $K \times \frac{K}{2}$ matrices, for $i \in \{0, 1\}$, and use two-level MDS codes. Considering that $A_2 = A_0 + A_1$ and $B_2 = B_0 + B_1$, nine codes are constructed by $A_i^T B_i$, for $i, j \in \{0, 1, 2\}$. In polynomial codes the master device, by dividing matrices column-wise, creates polynomials $\alpha(n) = A_0 + A_1 n$ and $\beta(n) = B_0 + B_1 n^2$ for worker w_n, which multiplies $\alpha(n)\beta(n)$. MatDot follows a similar idea of polynomial codes with the following difference: A and B are divided row-wise.

Table I shows the recovery threshold (k) [6]–[8], computing load per worker (γ) (this is the simplified analysis presented in Section III-A and further detailed in Appendix A in [30]), which shows the number of required multiplications, storage load per worker (μ), which shows the average amount of memory needed to store matrices and their multiplication (as detailed in Section III-B), and probability of successful computation (ρ), which is calculated assuming that the failure probability of workers is $\frac{1}{3}$ and independent. As seen, although MatDot is the best in terms of recovery threshold (k), it introduces more computing load per worker (because of row-wise partitioning). Also, MatDot codes perform worse than polynomial and product codes in terms of storage load per worker. Product codes require at least $N = 9$ workers due to their very design, but polynomial and MatDot codes are more flexible. As seen, there is a trade-off among $\{k, \gamma, \mu, \rho\}$, which we aim to explore in this paper by taking into account the limited edge computing resources including computing power and storage. For example, if there is no constraint on the total number of workers, but only on computing load, we will likely select product or polynomial codes. Next, we will provide a computing time and storage analysis of existing codes, and develop $\texttt{ACM}^2$ that selects the best code.

III. Adaptive Coding for Matrix Multiplication

A. Computing Time Analysis

Assuming a shifted-scaled exponential distribution as a computation delay model with λ as the straggling parameter and $\alpha \in \{p_{\text{rep}}, p_{\text{mds}}, p_{\text{poly}}^2, p_{\text{matdot}}, p_{\text{pro}}^2\}$ as the scale parameter for each worker, average computing time for repetition codes T_{rep} and MDS codes T_{mds} are expressed [1] as

$$T_{\text{rep}} \approx \frac{1}{p_{\text{rep}}}\left(1 + \frac{p_{\text{rep}}}{N\lambda}\log(p_{\text{rep}})\right), \tag{1}$$

$$T_{\text{mds}} \approx \frac{1}{p_{\text{mds}}}\left(1 + \frac{1}{\lambda}\log\left(\frac{N}{N-k_{\text{mds}}}\right)\right). \tag{2}$$

Corollary 1: The average computing time for polynomial codes T_{poly} and MatDot codes T_{matdot} is expressed as the following assuming that a shifted-scaled exponential distribution is used as a delay model.

$$T_{\text{poly}} \approx \frac{1}{p_{\text{poly}}^2}\left(1 + \frac{1}{\lambda}\log\left(\frac{N}{N-k_{\text{poly}}}\right)\right), \tag{3}$$

$$T_{\text{matdot}} \approx \frac{1}{p_{\text{matdot}}}\left(1 + \frac{1}{\lambda}\log\left(\frac{N}{N-k_{\text{matdot}}}\right)\right). \tag{4}$$

Proof: The proof is provided in Appendix B in [30]. □

The product codes have different performance in two different regimes. In the first regime the number of workers scales sublinearly with p_{pro}^2, *i.e.,* $N = p_{\text{pro}}^2 + \mathcal{O}(p_{\text{pro}})$, while in the second regime, the number of workers scales linearly with p_{pro}^2, *i.e.,* $N = p_{\text{pro}}^2 + \mathcal{O}(p_{\text{pro}}^2)$. The computing time analysis of product codes is provided in these two regimes next.

Corollary 2: Assume a shifted-scaled exponential distribution as a delay model with λ as the straggling parameter and p_{pro}^2 as the scale parameter for each worker, average computing time T_{pro} for $(p_{\text{pro}} + \frac{\tau}{2}, p_{\text{pro}})^2$ product codes and $(p_{\text{pro}} + \frac{\tau}{2})^2$ workers, where τ is an even integer, as p_{pro} grows to infinity, is expressed in the first regime as

$$T_{\text{pro}} \approx \frac{1}{p_{\text{pro}}^2}\left(1 + \frac{1}{\lambda}\log\left(\frac{p_{\text{pro}} + \frac{\tau}{2}}{c_{\tau/2+1}}\right)\right), \tag{5}$$

where $c_{\tau/2+1} \approx (1 + \tau/2) + \sqrt{(1+\tau/2)\log(1+\tau/2)}$ [31], [32]. Assuming the same delay distribution, the lower bound and upper bound of average computing time T_{pro} for $(\sqrt{1+\delta}p_{\text{pro}}, p_{\text{pro}})^2$ product codes and $(1+\delta)p_{\text{pro}}^2$ workers, for a fixed δ, as p_{pro} grows to infinity, is expressed in the second regime as

$$T_{\text{pro}}^{\text{low}} = \frac{1}{p_{\text{pro}}^2}\left(1 + \frac{1}{\lambda}\log\left(\frac{1+\delta}{\delta}\right)\right), \tag{6}$$

$$T_{\text{pro}}^{\text{up}} = \frac{1}{p_{\text{pro}}^2}\left(1 + \frac{2}{\lambda}\log\left(\frac{1+\delta+\sqrt{1+\delta}}{\delta}\right)\right). \tag{7}$$

Proof: The proof is provided in Appendix C in [30]. □

B. Storage Analysis

In this section, we provide storage requirements of the codes that we explained in Section II-B. These codes have storage requirements both at the master and worker devices. In particular, we assume that each entry of matrices A and B requires a fixed amount of memory. Our analysis, which is

TABLE I
COMPARISON OF PRODUCT, POLYNOMIAL, AND MATDOT CODES FOR MATRIX-MATRIX MULTIPLICATION.

	Recovery threshold (k)	Computing load per worker (γ)	Storage load per worker (μ)	Probability of successful computation (ρ)			
				N=6	N=7	N=8	N=9
Product	6	$\frac{K^3}{4}$	$K^2 + \frac{K^2}{4}$	N/A	N/A	N/A	0.63
Polynomial	4	$\frac{K^3}{4}$	$K^2 + \frac{K^2}{4}$	0.67	0.82	0.91	0.96
MatDot	3	$\frac{K^3}{2}$	$2K^2$	0.89	0.95	0.98	0.99

provided next, quantifies how many of these entries are needed to be stored at the master and worker devices.

1) Storage at Master: In all codes, we first store matrices A and B, where each matrix contains KL components. Also, we store the final result, $A^TB \in \mathbb{R}^{K\times K}$, which contains K^2 components. Therefore, $2KL + K^2$ entries should be stored at the master device for each code.

In repetition codes, we store the first k_{rep} results obtained from k_{rep} workers, where $k_{\text{rep}} = N - \frac{N}{p_{\text{rep}}} + 1$; *i.e.,* $S^m_{\text{rep}} = k_{\text{rep}}\frac{K^2}{p_{\text{rep}}} + 2KL + K^2$.

In MDS codes, we store $N - k_{\text{mds}}$ coded sub-matrices. Each coded matrix contains $\frac{KL}{p_{\text{mds}}}$ components. Then, we need to store the first k_{mds} results obtained from k_{mds} workers. The storage requirement of MDS codes at the master device is $S^m_{\text{mds}} = (N - k_{\text{mds}})\frac{KL}{p_{\text{mds}}} + k_{\text{mds}}\frac{K^2}{p_{\text{mds}}} + 2KL + K^2$.

In polynomial codes, we store $2N$ coded sub-matrices. Each coded matrix contains $\frac{KL}{p_{\text{poly}}}$ components. Then, we need to store the first k_{poly} results obtained from k_{poly} workers. In other words, we have $S^m_{\text{poly}} = 2N\frac{KL}{p_{\text{poly}}} + k_{\text{poly}}\frac{K^2}{p^2_{\text{poly}}} + 2KL + K^2$.

In MatDot, we store $2N$ coded sub-matrices. Each coded matrix contains $\frac{KL}{p_{\text{matdot}}}$ components. Then, we need to store the first k_{matdot} results collected from k_{matdot} workers, so $S^m_{\text{matdot}} = 2N\frac{KL}{p_{\text{matdot}}} + k_{\text{matdot}}K^2 + 2KL + K^2$.

In product codes, we store $2(N - k_{\text{pro}})$ coded sub-matrices. Each coded matrix contains $\frac{KL}{p_{\text{pro}}}$ components. Then, we need to store the first k_{pro} results collected from k_{pro} workers, where $k_{\text{pro}} = 2(p_{\text{pro}} - 1)\sqrt{N} - (p_{\text{pro}} - 1)^2 + 1$ [6], so $S^m_{\text{pro}} = 2(N - k_{\text{pro}})\frac{KL}{p_{\text{pro}}} + k_{\text{pro}}\frac{K^2}{p^2_{\text{pro}}} + 2KL + K^2$.

2) Storage at Workers: In repetition codes, each worker receives one matrix of size $\frac{KL}{p_{\text{rep}}}$ and one matrix of size KL, as we decompose only one of the matrices to p_{rep} parts. So, the size of resulting matrix is $\frac{K^2}{p_{\text{rep}}}$. Similarly, in MDS codes we decompose only one of the matrices to p_{mds} parts. Each worker receives one matrix with size of $\frac{KL}{p_{\text{mds}}}$ and one matrix with size of KL. Thus, the size of resulting matrix is $\frac{K^2}{p_{\text{mds}}}$. Thus, the storage requirement of repetition and MDS codes is expressed as $S^w_x = \frac{KL}{p_x} + KL + \frac{K^2}{p_x}, \forall x \in \{\text{rep}, \text{mds}\}$.

In polynomial and product codes, each worker receives two matrices of size $\frac{KL}{p_x}$, $x \in \{\text{poly}, \text{pro}\}$. The size of the matrix after multiplication is $\frac{K^2}{p_x^2}$. Therefore, the storage requirement of polynomial and product codes at each worker is

$$S^w_x = \frac{2KL}{p_x} + \frac{K^2}{p_x^2}, \forall x \in \{\text{poly}, \text{pro}\}. \tag{8}$$

On the other hand, the size of the matrix after computation is K^2 in MatDot as matrix partitioning is done differently (*i.e.,* row-wise, which means that $A_i \in \mathbb{R}^{\frac{L}{p_{\text{matdot}}}\times K}$ and $B_i \in \mathbb{R}^{\frac{L}{p_{\text{matdot}}}\times K}$ after partitioning and finally, $A_i^TB_i \in \mathbb{R}^{K\times K}$) as compared to polynomial and product codes (partitions column-wise, which means that we have $A_i \in \mathbb{R}^{L\times\frac{K}{p_x}}$ and $B_i \in \mathbb{R}^{L\times\frac{K}{p_x}}$ after partitioning, and the final result is $A_i^TB_i \in \mathbb{R}^{\frac{K}{p_x}\times\frac{K}{p_x}}, \forall x \in \{\text{poly}, \text{pro}\}$). Therefore, the storage requirement of MatDot is expressed as

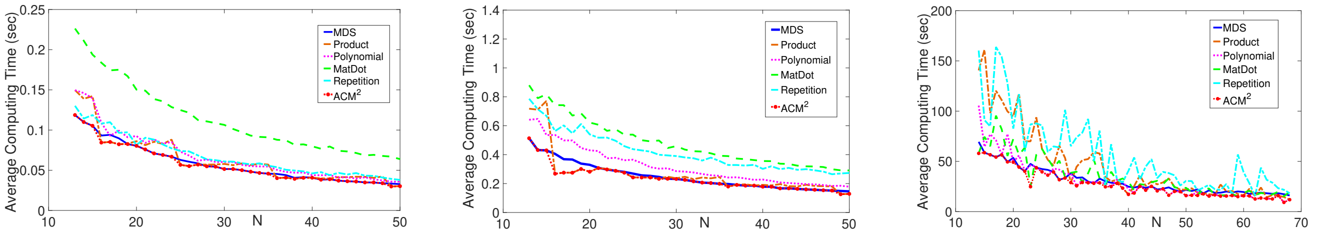
$$S^w_{\text{matdot}} = \frac{2KL}{p_{\text{matdot}}} + K^2. \tag{9}$$

C. Design of ACM^2 *Algorithm*

In this section, we present our ACM^2 algorithm. We consider an iterative process such as gradient descent, where matrix multiplications are required at each iteration. Our goal is to determine the best matrix multiplication code and the optimum number of matrix partitions by taking into account the task completion delay, *i.e.,* computing time, storage requirements and decoding probability of each code. In particular, ACM^2 solves an optimization problem at each iteration, and determines which code is the best as well as the optimum number of partitions for that code. For example, MDS codes may be good at iteration i, while polynomial codes may suit better in later iterations. The optimization problem is formulated as

$$\begin{aligned} \min_{\pi, p_\pi} \quad & T_\pi \\ \text{subject to} \quad & S^z_\pi \le S^z_{\text{thr}}, z \in \{m, w\}, \\ & \rho_\pi \ge \rho_{\text{thr}}, k_\pi \le N, p_\pi \ge 2, \\ & \pi \in \{\text{rep}, \text{mds}, \text{poly}, \text{matdot}, \text{pro}\}. \end{aligned} \tag{10}$$

The objective function selects the best code π from the set $\{\text{rep}, \text{mds}, \text{poly}, \text{matdot}, \text{pro}\}$ as well as the optimum number of partitions p_π. The first constraint is the storage constraint, which limits the storage usage at master and worker devices with thresholds S^m_{thr} and S^w_{thr}. The term $\rho_\pi \ge \rho_{\text{thr}}$ in the second constraint is the successful decoding constraint, where successful decoding probability ρ_π should be larger than the threshold ρ_{thr}. The successful decoding probability is defined as the probability that the master receives all the required results from workers. If we assume that the failure probability of each worker is $(1-\phi)$ and independent, the total number of workers is N and the number of sufficient results is k_π, one may formulate the probability of success for each coding method



(a) There are no storage or successful decoding probability constraints.

(b) Storage is constrained.

(c) Storage and success probability are constrained.

Fig. 1. Task completion delay versus number of workers.

as a binomial probability $\rho_\pi = \sum_{i=k_\pi}^{N} \binom{N}{i}(\phi)^i(1-\phi)^{N-i}$. The term $k_\pi \leq N$ in the second constraint makes sure that the recovery threshold k_π is less than the number of workers. The term $p_\pi \geq 2$ in the second constraint makes the number of partitions larger than or equal to 2, otherwise there is no matrix partitioning, which is a degenerate case.

IV. Performance Analysis of ACM^2

In this section, we evaluate the performance of our algorithm; ACM^2 via simulations. We consider a master/worker setup, where per sub-matrix computing delay λ is an i.i.d. random variable following a shifted exponential distribution. We compare ACM^2 with the baselines; repetition, MDS, polynomial, MatDot, and product codes.

Fig. 1(a) shows the average computing time versus number of workers N when there is no storage or successful decoding probability constraint in (10). $K = 2000$, $L = 5000$, $\phi = 0.95$, λ is randomly and uniformly selected from $\{2, \ldots, 10\}$. In this setup, workers are fast (since $\lambda > 1$), so all coding algorithms prefer splitting matrices to more partitions. This causes low computing load at workers, but the master needs to wait for more results, *i.e.,* recovery threshold k is large, to be able to decode computed sub-matrices. The optimum number of partitions of repetition codes is equal to N ($p^*_{\text{rep}} = N$), while for MDS codes it is close to N ($0.9 < \frac{p^*_{\text{mds}}}{N} < 1$). This means that repetition codes are the same as no coding and MDS gets closer to no coding as λ increases [1], [28]. When N is small, no coding is the best, so ACM^2 chooses MDS codes (which is close to repetition codes) as they behave as no coding. When N increases, MDS, polynomial, and product codes perform better than repetition codes. Product codes operate in the first regime, because workers are fast. Thus, the optimum number of partitions is large and close to N, so, $N = p^2_{\text{pro}} + \tau p_{\text{pro}}$. Product codes perform better in this regime [7]. When $\sqrt{N}$ is integer, product codes are the best as they can use all existing workers and choose the number of partitions as large as possible to decrease the computation load of each worker. However, when $\sqrt{N}$ is not integer, product codes may waste resources of some workers (as they only use $\lfloor\sqrt{N}\rfloor$ workers), so MDS and polynomial codes perform better. Computation time of MDS is less than or equal to polynomial, because computation load of polynomial is higher than MDS codes. The optimum number of partitions for each code increases with increasing N, which decreases the computation load at each worker. Since all workers are fast ($\lambda > 1$), all codes choose as large partitions as possible. Thus, they perform close to each other. MatDot performs worse than the other codes, because of its different way of partitioning; *i.e.,* row-wise versus column-wise. Thus, MatDot introduces almost 2 times more computation load. As seen, ACM^2 exploits the best code among all codes, so it performs the best.

Fig. 1(b) demonstrates average computing time versus number of workers when there exists storage constraint in (10). In this setup, $K = 2000$, $L = 5000$, $\phi = 0.95$, λ is selected randomly and uniformly from $\lambda \in \{\frac{1}{10}, \ldots, \frac{1}{2}\}$, and the storage constraint is set to $S^w_{\text{thr}} = 15\text{M}$ entries. As the workers are slow, *i.e.,* $\lambda < 1$, all codes prefer to choose small number of partitions. There is a trade-off between the number of partitions and storage requirement. It means that the storage requirement reduces with increasing number of partitions as smaller matrices are multiplied by each worker, so less storage is needed. Since there is a storage constraint, all codes prefer to increase the number of partitions. ACM^2 exploits this trade-off and selects the best code and optimum number of partitions.

Fig. 1(c) illustrates average computing time versus number of workers when there exists both storage and success probability constraints in (10). In this setup, $K = 2000$, $L = 5000$, $\phi = 0.9$, λ is selected randomly and uniformly from $\lambda \in \{\frac{1}{2000}, \frac{1}{1000}, \frac{1}{900}, \frac{1}{800}, \frac{1}{700}, \frac{1}{600}, \frac{1}{500}\}$, the storage constraint is set to $S^w_{\text{thr}} = 10\text{M}$ entries and the success probability constraint is set to $\rho_{\text{thr}} = 0.98$. Our proposed algorithm selects any of the MDS, product, polynomial, MatDot and repetition codes at least one time during these iterations. MatDot and polynomial codes perform better as compared with Fig. 1(a) and Fig. 1(b). Polynomial codes work better due to the tighter storage constraint and MatDot codes perform better because of the existence of success probability constraint, which has an inverse relation with the recovery threshold.

V. Conclusion

In this paper, we focused on characterizing the cost-benefit trade-offs of coded computation for practical edge computing, and develop an adaptive coded computation framework. In particular, we studied matrix multiplication as a computationally intensive task, and developed an adaptive coding for matrix multiplication (ACM^2) algorithm by taking into account the heterogeneous and time varying nature of edge devices. ACM^2 dynamically selects the best coding policy by taking into account the computing time, storage requirements as well as successful decoding probability.

References

[1] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” *IEEE Transactions on Information Theory*, vol. 64, no. 3, March 2018.

[2] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, “A fundamental tradeoff between computation and communication in distributed computing,” *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, Jan 2018.

[3] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[4] S. Lin and D. Costello, *Error-Correcting Codes*. Prentice-Hall, Inc, 1983.

[5] N. S. Ferdinand and S. C. Draper, “Anytime coding for distributed computation,” in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2016, pp. 954–960.

[6] Q. Yu, M. Maddah-Ali, and S. Avestimehr, “Polynomial codes: an optimal design for high-dimensional coded matrix multiplication,” in *Advances in Neural Information Processing Systems*, 2017.

[7] K. Lee, C. Suh, and K. Ramchandran, “High-dimensional coded matrix multiplication,” in *2017 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2017, pp. 2418–2422.

[8] M. Fahim, H. Jeong, F. Haddadpour, S. Dutta, V. Cadambe, and P. Grover, “On the optimal recovery threshold of coded matrix multiplication,” in *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 1264–1270.

[9] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Straggler mitigation in distributed matrix multiplication: Fundamental limits and optimal coding,” in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 2022–2026.

[10] S. Dutta, V. Cadambe, and P. Grover, “Short-dot: Computing large linear transforms distributedly using coded short dot products,” in *Advances In Neural Information Processing Systems*, 2016, pp. 2100–2108.

[11] ——, “Coded convolution for parallel and distributed computing within a deadline,” *arXiv preprint arXiv:1705.03875*, 2017.

[12] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, “Gradient coding: Avoiding stragglers in distributed learning,” in *International Conference on Machine Learning*, 2017, pp. 3368–3376.

[13] W. Halbawi, N. Azizan, F. Salehi, and B. Hassibi, “Improving distributed gradient descent using reed-solomon codes,” in *2018 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2018, pp. 2027–2031.

[14] N. Raviv, R. Tandon, A. Dimakis, and I. Tamo, “Gradient coding from cyclic MDS codes and expander graphs,” in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 4305–4313.

[15] C. Karakus, Y. Sun, and S. Diggavi, “Encoded distributed optimization,” in *2017 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2017, pp. 2890–2894.

[16] C. Karakus, Y. Sun, S. Diggavi, and W. Yin, “Straggler mitigation in distributed optimization through data encoding,” in *Advances in Neural Information Processing Systems*, 2017, pp. 5434–5442.

[17] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “Coded fourier transform,” in *2017 55th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2017, pp. 494–501.

[18] Y. Yang, P. Grover, and S. Kar, “Computing linear transformations with unreliable components,” *IEEE Trans. on Information Theory*, 2017.

[19] Q. Yu, S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “How to optimally allocate resources for coded distributed computing?” in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017.

[20] S. Li, Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, “A scalable framework for wireless distributed computing,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 2643–2654, 2017.

[21] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, “Coded distributed computing: Straggling servers and multistage dataflows,” in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2016, pp. 164–171.

[22] M. Kiamari, C. Wang, and A. S. Avestimehr, “On heterogeneous coded distributed computing,” in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7.

[23] A. Reisizadeh, S. Prakash, R. Pedarsani, and A. S. Avestimehr, “Coded computation over heterogeneous clusters,” *IEEE Transactions on Information Theory*, vol. 65, no. 7, pp. 4227–4242, 2019.

[24] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, “Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–40, 2019.

[25] Y. Keshtkarjahromi, Y. Xing, and H. Seferoglu, “Dynamic heterogeneity-aware coded cooperative computation at the edge,” in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE, 2018, pp. 23–33.

[26] E. Ozfatura, S. Ulukus, and D. Gündüz, “Straggler-aware distributed learning: Communication–computation latency trade-off,” *Entropy*, vol. 22, no. 5, p. 544, 2020.

[27] S. Kiani, N. Ferdinand, and S. C. Draper, “Hierarchical coded matrix multiplication,” in *2019 16th Canadian Workshop on Information Theory (CWIT)*. IEEE, 2019, pp. 1–6.

[28] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, “Speeding up distributed machine learning using codes,” in *2016 IEEE International Symposium on Information Theory (ISIT)*, July 2016, pp. 1143–1147.

[29] G. Liang and U. C. Kozat, “Tofec: Achieving optimal throughput-delay trade-off of cloud storage using erasure codes,” in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 826–834.

[30] E. Vedadi and H. Seferoglu, “Adaptive coding for matrix multiplication at edge networks,” *arXiv preprint arXiv:2103.04247*, 2021.

[31] J. Justesen and T. Hoholdt, “Analysis of iterated hard decision decoding of product codes with reed-solomon component codes,” in *2007 IEEE Information Theory Workshop*, Sep. 2007, pp. 174–177.

[32] B. Pittel, J. Spencer, and N. Wormald, “Sudden emergence of a giantk-core in a random graph,” *Journal of Combinatorial Theory, Series B*, vol. 67, no. 1, pp. 111 – 151, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0095895696900362