Graph-based Algorithms for Linear Computation Coding

International Zurich Seminar on Information and Communication (IZS), March 6 – 8, 2024

Graph-based Algorithms for Linear Computation Coding

Hans Rosenberger*, Ali Bereyhi†, Ralf R. Müller*
*Institute for Digital Communications, Friedrich-Alexander-Universität (FAU), Erlangen, Germany
{hans.rosenberger, ralf.r.mueller}@fau.de
†Wireless Computing Lab, University of Toronto, Canada
ali.bereyhi@utoronto.ca

***Abstract*—We revisit existing linear computation coding (LCC) algorithms, and introduce a new framework that measures the computational cost of computing multidimensional linear functions, not only in terms of the number of additions, but also with respect to their suitability for parallel processing. Utilizing directed acyclic graphs, which correspond to signal flow graphs in hardware, we propose a novel LCC algorithm that controls the trade-off between the total number of operations and their parallel executability. Numerical evaluations show that the proposed algorithm, constrained to a fully parallel structure, outperforms existing schemes.**

I. Introduction

Over-parameterized neural networks (NNs) have achieved many of the recent advancements in improving inference accuracy. Many real-world applications of these very large NNs require both real-time inference and operate in a resource constrained environment. It is therefore of great importance to implement them with minimal computational complexity. Various research efforts have been directed towards improving NN efficiency, including pruning, knowledge distillation, quantization and NN-hardware co-design [1], [2].

Linear computation coding (LCC) introduces an analytical framework that invokes the idea of sparse matrix decomposition to reduce the computational cost of computing matrix-vector products, i.e. the lossy compression of a multidimensional linear function with constant coefficients. Earlier studies on LCC mainly focus on the number of additions as the metric of computational complexity [3]–[6]. Though important, this metric is not the only concern in many applications.

In this paper, we revisit the earlier LCC studies from a new perspective on computational complexity, in which not only the number of operations, but also their order matters. Our interest follows from a simple fact: optimizing the order in which the operations are carried out enables us to fully exploit the potential of *parallel processing*. We use the notion of a directed acyclic graph (DAG), closely corresponding to the signal flow graph of a hardware implementation, to develop a new LCC algorithm. The proposed scheme explicitly tunes the structure of the DAG and outperforms existing algorithms on parallel processing units.

A. Notation

Vectors and matrices are denoted by lower- and upper-case boldface letters, e.g. $\boldsymbol{x}$ and $\boldsymbol{X}$, respectively. The Euclidean and Frobenius norms are shown by $\|\cdot\|_2$ and $\|\cdot\|_{\mathrm{F}}$, respectively. The matrix transpose is denoted by $(\cdot)^{\mathrm{T}}$. The augmented identity matrix with dimension $N \times K$ is denoted by $\boldsymbol{I}_{N\times K}$, and the j-th row unit vector in K dimensions by $\mathbf{1}_{j,K}$. The function $\mathrm{supp}(\boldsymbol{x})$ returns the indices in the support of $\boldsymbol{x}$, i.e. the set of all indices i where $x_i \neq 0$.

Sets are specified by upper case caligraphic letters, e.g. $\mathcal{A}$. We use the notation $|\mathcal{A}|$ to represent the cardinality of $\mathcal{A}$. A DAG is denoted by $D = (\mathcal{C}, \mathcal{A})$, where $\mathcal{C} \subset \mathbb{R}^{1\times K}$ is the ordered set of all vertices and $\mathcal{A}$ the set of arcs (directed edges). The indegree and outdegree of a vertex $\boldsymbol{c} \in \mathcal{C}$ are denoted by $\mathrm{d}_{\mathrm{D}}^{-}(\boldsymbol{c})$ and $\mathrm{d}_{\mathrm{D}}^{+}(\boldsymbol{c})$, respectively. Given a DAG $D = (\mathcal{C}, \mathcal{A})$ and a vertex $\boldsymbol{c} \in \mathcal{C}$, $\mu_{\mathrm{D}}(\boldsymbol{c})$ denotes the depth of $\boldsymbol{c}$, i.e. the longest path from any node $\boldsymbol{c}' \in \mathcal{C}$ to node $\boldsymbol{c}$. The operator $\mathrm{mat}(\cdot)$ converts a vertex set $\mathcal{C} = \{\boldsymbol{c}_1, \ldots, \boldsymbol{c}_L\} \subset \mathbb{R}^{1\times K}$ with $|\mathcal{C}| = L$ to its corresponding matrix, i.e. $\boldsymbol{C} = \mathrm{mat}(\mathcal{C}) = [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_L] \in \mathbb{R}^{L\times K}$. Unless otherwise specified, $\boldsymbol{c}_i$ denotes the i-th element in the set $\mathcal{C}$ or the i-th row vector of the corresponding matrix $\boldsymbol{C} = \mathrm{mat}(\mathcal{C})$. The notation $[N]$ is an abbreviation for the set $\{1, \ldots, N\}$.

II. Preliminaries

Consider the matrix vector product

$$\boldsymbol{y} = \boldsymbol{T}\boldsymbol{x} \tag{1}$$

with the arbitrary, but constant, matrix $\boldsymbol{T} \in \mathbb{R}^{N\times K}$ and the arbitrary input vector $\boldsymbol{x} \in \mathbb{R}^{K\times 1}$. Our goal is to approximately compute $\boldsymbol{y} \in \mathbb{R}^{N\times 1}$ with minimum effort. Calculating the matrix-vector product straightforwardly requires NK multiplications and $N(K-1)$ additions. Using a finite-precision representation of $\boldsymbol{T}$, a multiplication can be reduced to additions and bitshifts. Quantizing the matrix entries independently, it is well known that each additional bit on average improves the signal to quantization noise ratio (SQNR) by $6\,\mathrm{dB}$ while requiring half an extra addition. Using the canonically signed digit (CSD) representation [7], i.e. allowing for subtractions as well, the SQNR even improves by $14.5\,\mathrm{dB}$ per digit. However, by quantizing the operations of a matrix-vector product jointly, far larger gains are possible [3], [8].

A. Addition as a Fundamental Operation

Definition 1 (Fundamental Operation): Let $\mathcal{C} \subset \mathbb{R}^{1\times K}$ denote a set of L vectors and be called a codebook. We define the fundamental operation as the linear combination of at most S vectors contained in $\mathcal{C}$, or, more formally:

$$\mathrm{add}_S(\boldsymbol{\omega}_S, \mathcal{C}) = \boldsymbol{\omega}_S \,\mathrm{mat}(\mathcal{C}) \tag{2}$$

This work was supported by Deutsche Forschungsgemeinschaft (DFG) under the project Computation Coding (MU-3735-/8-1).

with $\boldsymbol{\omega}_S \in \mathcal{W}_S$, where

$$\mathcal{W}_S = \left\{ \boldsymbol{\omega} = \sum_{s=1}^{S} i_s \mathbf{1}_{j_s,\,L} : i_s \in \mathcal{M} \subseteq \{0, \pm 2^{\mathbb{Z}}\}, j_s \in [L] \ \forall s \right\}. \tag{3}$$

The nonzero coefficients of $\boldsymbol{\omega}_S \in \mathcal{W}_S$ are restricted to the set of (sums of) signed powers of two, corresponding only to bitshifts in hardware, which can be considered computationally cheap.[1] The computational cost of a fundamental operation is governed by the at most $S-1$ additions needed to form the linear combination.

Given a codebook $\mathcal{C}$ and using the notion of the fundamental operation, our aim is now to approximate a target vector $\boldsymbol{t}$ by a single fundamental operation. We call this objective wiring. Mathematically we aim to solve the following least squares (LS) problem:

$$w(\boldsymbol{t}, \mathcal{C}, S) = \underset{\boldsymbol{\omega}_S \in \mathcal{W}_S}{\operatorname{argmin}} \| \boldsymbol{t} - \boldsymbol{\omega}_S \operatorname{mat}(\mathcal{C}) \|_2, \tag{4}$$

which can be equivalently seen as a sparse recovery problem [9] due to the restricted support of $\boldsymbol{\omega}_S$.

The minimization over the set of discrete vectors $\mathcal{W}_S$ in (4) is an NP-hard problem. Hence, an optimal solution is generally computationally intractable. Therefore, we resort to the following two suboptimal approaches:

- *Discrete matching pursuit (DMP)* [3]: Start with $\boldsymbol{\omega} \leftarrow \mathbf{0}$. Find the vector in $\boldsymbol{c}_i \in \mathcal{C}$ scaled by a signed power of two that reduces the error to $\boldsymbol{t}$ maximally and update $\boldsymbol{\omega}$ in the i-th component. Repeat S times.
- *Reduced state (RS) approach* [5]: Procedure similar to DMP. However, instead of choosing in each iteration the best vector minimizing the error, we retain a list of the Q best linear combinations in each iteration and choose the combination with minimum error at termination. This procedure enables a performance close to full search at a reasonable time complexity [5].

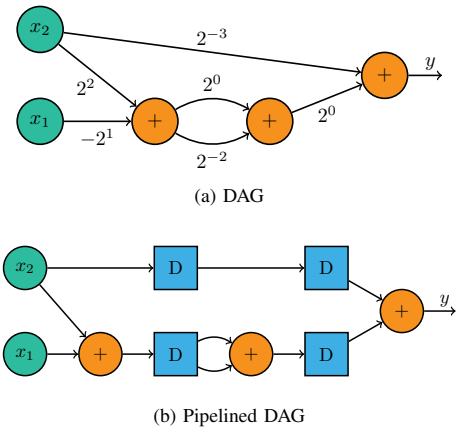
To quantify the ability of a codebook $\mathcal{C}$ to approximate the matrix $\boldsymbol{T}$ with row vectors $\boldsymbol{t}_n$, we use the SQNR defined as

$$\operatorname{SQNR}(\boldsymbol{T}, \mathcal{C}) = \frac{\|\boldsymbol{T}\|_{\mathrm{F}}^2}{\sum_{n=1}^{N} \|\boldsymbol{t}_n - w(\boldsymbol{t}_n, \mathcal{C}, 1) \operatorname{mat}(\mathcal{C})\|_2^2}. \tag{5}$$

Note that $w(\boldsymbol{t}_n, \mathcal{C}, 1) \operatorname{mat}(\mathcal{C})$ finds the vector in $\mathcal{C}$ scaled by a signed power of two, that approximates $\boldsymbol{t}_n$ best. As $S = 1$, this is only a selection and potentially a bitshift, no additions are required.

B. Constant matrix vector multiplication (CMVM)

Using the notion of a fundamental operation, any matrix-vector product with finite precision can now be expressed as a DAG with K input and N output vertices. Input vertices are all vertices with no preceding fundamental operations, i.e. $\{\boldsymbol{c} \in \mathcal{C} | \, \mathrm{d}_{\mathrm{D}}^{-}(\boldsymbol{c}) = 0\}$. Likewise, output vertices have no arcs directed to subsequent vertices ($\{\boldsymbol{c} \in \mathcal{C} | \, \mathrm{d}_{\mathrm{D}}^{+}(\boldsymbol{c}) = 0\}$). In such a graph, each vertex, except the input vertices, corresponds to one fundamental operation, and each directed arc is labeled with a signed power of two. An example of such a DAG is depicted in Fig. 1a. It is our goal, given some target matrix



(a) DAG

(b) Pipelined DAG

Fig. 1: A DAG realizing the function $y(x_1, x_2) = (21/8)x_2 - (5/4)x_1$ is depicted in (a). The same DAG is extended in (b) with delay elements to allow for pipelining.

$\boldsymbol{T}$, to find a DAG requiring a minimum of computations given some fidelity constraint. We can therefore define now a CMVM problem.

Definition 2 (CMVM Problem): For all fundamental operations assume without loss of generality $S = 2$. Given a target matrix $\boldsymbol{T}$ and some positive parameter ϵ, find a DAG $D = (\mathcal{C}, \mathcal{A})$ with vertex set $\mathcal{C} \subset \mathbb{R}^{1 \times K}$, that solves

$$\begin{aligned}
\min \quad & |\mathcal{C}| && && \text{(6a)}\\
\text{s.t.} \quad & \operatorname{SQNR}(\boldsymbol{T}, \mathcal{C}) > \epsilon && && \text{(6b)}\\
& \boldsymbol{c}_m = \mathbf{1}_{m,K} && \forall m \in [K] && \text{(6c)}\\
& \boldsymbol{c}_l = \operatorname{add}_2(\boldsymbol{\omega}_2, \{\boldsymbol{c}_i \in \mathcal{C} : i \in [l-1]\}) && \forall l > K && \text{(6d)}
\end{aligned}$$

The CMVM problem is at least NP-complete. Similar to multiple constant multiplication (MCM) [10], it is an even broader generalization of the single constant multiplication (SCM) problem,[2] which is known to be NP-complete [11], [12]. Hence, by polynomial reduction the CMVM problem has to be at least as difficult. As the optimal solution is generally computationally intractable, we focus for the remainder of this paper on the development of efficient heuristics for obtaining decomposition DAGs.

Remark 1: Throughout the paper we do not specify the set of arcs $\mathcal{A}$ of a DAG explicitly for reasons of brevity. As new vertices are created from an initial codebook, i.e. the set of unit vectors, by means of fundamental operations, implicitly $\mathcal{A}$ is defined uniquely[3] by $\mathcal{C}$ for any decomposition DAG $D = (\mathcal{C}, \mathcal{A})$ as well.

C. Computational Cost

Three terms contribute to the overall computational cost

$$C_{\text{total}} = C_{\text{add}} N_{\text{add}} + C_{\text{delay}} N_{\text{delay}} + C_{\text{inv}} N_{\text{inv}}. \tag{7}$$

The number of additions N_{add}, the number of delay elements (latches) N_{delay} and the number of sign inverters N_{inv} required. Further, C_{add}, C_{delay} and C_{inv} are the effective cost for an

[1] In this paper we consider the set of wiring coefficients to be unrestricted, i.e. $\mathcal{M} = \{0, \pm 2^{\mathbb{Z}}\}$. For some applications, it is beneficial to restrict the coefficients to a subset. Efficient strategies for such cases are investigated in [6].

[2] The optimization of the multiplication of a constant scalar to a scalar variable.

[3] Uniqueness only refers in that context to the start and endpoint of individual arcs, not their labeling. For example two different fundamental operations, differing in their labeling/bitshift, might produce the same result, i.e. $\boldsymbol{c}_2 = \boldsymbol{c}_1 - 1/4\boldsymbol{c}_1 = 1/2\boldsymbol{c}_1 + 1/4\boldsymbol{c}_1$.

addition, a delay element and an inverter, respectively. Inspired by the CMOS implementation of these basic functions, we assume for simplicity that the cost for an adder and a delay element are approximately equal and set to[4] $C_{\text{add}} = C_{\text{delay}} = 20$. For an inverter we assume a cost of $C_{\text{inv}} = 2$, since these can be easily implemented by two transistors [13].

The number of additions in computing a DAG is upper bounded, as zeros are allowed for coefficients as well, by

$$N_{\text{add}} = \sum_{i=K+1}^{|\mathcal{C}|} \left(\mathrm{d}_{\mathrm{D}}^{-}(\boldsymbol{c}_i) - 1\right) \tag{8a}$$

$$\overset{\text{(a)}}{=} (|\mathcal{C}| - K)(S-1) \tag{8b}$$

where (a) follows from the fact that the number of additions $S-1$ for all vertices is constant.

For medium to large matrices it may not be desirable to straightforwardly implement the DAG in hardware, apply a realisation of $\boldsymbol{x}$, and wait for the output $\boldsymbol{y}$ to be computed. Particularly, for a DAG with many logical operations in sequence, this may take some time and is not an optimal use of resources. Instead, a pipelined approach is desirable,[5] each adder is followed by a latch or delay element that is able to store the intermediate result produced by that adder. For example, after an addition is completed, and the result is stored, the following input realization can already be forwarded to the adder. The stored result is then forwarded to the subsequent adder. The schematic of a pipelined design is depicted in Fig. 1b. There, a pipelined signal flow graph/DAG with two inputs x_1 and x_2 computes a single output y. The second input is required for the final addition computing the output. Thus, two additional delay elements are required in the upper branch to delay the input accordingly, adding to the overall hardware cost.

Pipelining largely improves overall throughput, keeping each adder busy and reducing idle times of resources. However, to enable that, idle paths require additional delay elements that contribute to the overall hardware cost. Hence, for a practical algorithm it is desirable to not only minimize the number of adders but to find a DAG structure that limits the number of delay elements. The overall number of delay elements required for a pipelined implementation of a decomposition DAG can be computed by

$$N_{\text{delay}} = N_{\text{add}} + \sum_{\forall \tilde{\boldsymbol{c}} \in \tilde{\mathcal{C}}} \left(\max_{\boldsymbol{c} \in \mathcal{D}(\tilde{\boldsymbol{c}})} \mu_{\mathrm{D}}(\boldsymbol{c}) - \mu_{\mathrm{D}}(\tilde{\boldsymbol{c}}) - 1 \right) \tag{9a}$$

with

$$\tilde{\mathcal{C}} = \left\{ \boldsymbol{c} \in \mathcal{C} |\, \mathrm{d}_{\mathrm{D}}^{+}(\boldsymbol{c}) > 0 \right\} \tag{9b}$$

$$\mathcal{D}(\tilde{\boldsymbol{c}}) = \{ \boldsymbol{c} \in \mathcal{C} | (\tilde{\boldsymbol{c}}, \boldsymbol{c}) \in \mathcal{A} \} \tag{9c}$$

The set $\mathcal{D}(\tilde{\boldsymbol{c}})$ contains all vertices $\boldsymbol{c}$ that are connected by a directed arc in $\mathcal{A}$ from $\tilde{\boldsymbol{c}}$ to $\boldsymbol{c}$. The total number of delay elements is the sum of the number of adders, as each adder needs a buffer at the output, and for each node with outgoing arcs the longest path difference minus one that needs to be equalized.

[4]The cost of a full adder ranges around 20 transistors and can vary depending on the specific implementation used, clock speed, etc. This cost only considers a full adder for the addition of two inputs of a single bit. For larger bitwidths the cost scales accordingly and simplifications in the implementation are possible. For simplicity we only consider the cost per bit.

[5]For a detailed discussion of pipelining, refer to [14].

The number of inverters depends on the specific algorithm used. For brevity, we will not discuss inverters in detail. A reduction algorithm for the number of inverters in parallel LCC algorithms is discussed in [15].

III. Algorithmic Approaches

We now discuss two existing algorithmic approaches for LCC, namely a fully sequential and fully parallel algorithm. Utilizing the best of both worlds, we introduce a new mixed algorithm (MA) that enables us to tune the DAG structure for further analysis.

A. Fully sequential (FS) Algorithm

Given the set of all unit vectors in K dimensions as our initial codebook set $\mathcal{C} = \{\mathbf{1}_{1,K}, \ldots, \mathbf{1}_{K,K}\}$, we recursively add vertices to the DAG using the following update rule [4]:

$$\mathcal{C} \leftarrow \mathcal{C} \cup \{ w(\boldsymbol{t}_{\tilde{n}}, \mathcal{C}, S) \operatorname{mat}(\mathcal{C}) \}. \tag{10}$$

This means that we find the best linear combination of vectors in $\mathcal{C}$ that approximates $\boldsymbol{t}_{\tilde{n}}$ well and requires $S-1$ additions. We choose the row vector with index $\tilde{n}$ from $\boldsymbol{T}$ that provides us with the largest reduction of the squared error for the update:

$$\tilde{n} = \underset{n \in [N]}{\operatorname{argmin}} \Bigg(\| \boldsymbol{t}_n - w(\boldsymbol{t}_n, \mathcal{C}, S) \operatorname{mat}(\mathcal{C}) \|_2^2 + \sum_{k \neq n} \| \boldsymbol{t}_k - w(\boldsymbol{t}_k, \mathcal{C}, 1) \operatorname{mat}(\mathcal{C}) \|_2^2 \Bigg) \tag{11}$$

Although this approach shows excellent performance when looking at the tradeoff between distortion and the number of additions required, it is in many cases not suited for pipelining. This follows from the fact that any S vertices in a given codebook can be combined in each iteration, the obtained graph has an arbitrary structure (c.f. Fig. 2a). Assuming for simplicity that each fundamental operation takes time t_{f} to compute,[6] it is concluded that the delay at any node $\boldsymbol{c}$ is $\mu_{\mathrm{D}}(\boldsymbol{c}) t_{\mathrm{f}}$. Thus, if the depth $\mu_{\mathrm{D}}(\boldsymbol{c})$ varies in $\boldsymbol{c}$, delays are introduced that need to be compensated for. The additional hardware resources and overhead required by the FS algorithm are typically not acceptable, especially for large matrices. Therefore, algorithms that take these hardware constraints into account are desirable.

B. Fully parallel (FP) algorithm

Instead of performing updates sequentially, we now successively refine the codebook for all vectors of the target matrix in parallel and then forget the old codebook. Such a fully parallel algorithm can be written as a product of matrices [3]:

$$\boldsymbol{T} \approx \boldsymbol{W}_L \boldsymbol{W}_{L-1} \cdots \boldsymbol{W}_2 \boldsymbol{W}_1 \boldsymbol{C}_0. \tag{12}$$

The n-th row of the l-th matrix factor $\boldsymbol{W}_l$ is recursively obtained by

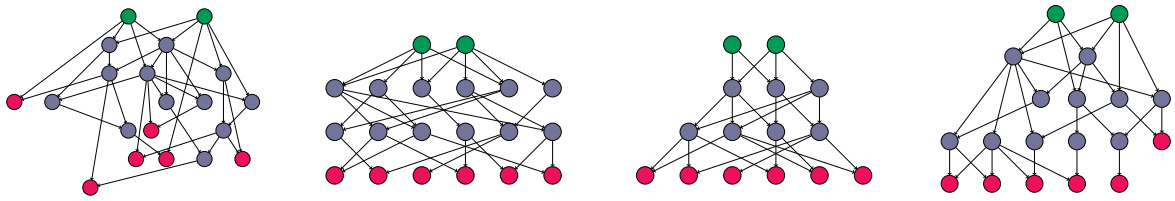
$$\boldsymbol{w}_{l,n} = w(\boldsymbol{t}_n, \boldsymbol{C}_{l-1}, S) \quad \forall n \in [N] \tag{13}$$

with

$$\boldsymbol{C}_{l-1} = \boldsymbol{W}_{l-1} \boldsymbol{W}_{l-2} \cdots \boldsymbol{W}_2 \boldsymbol{W}_1 \boldsymbol{C}_0. \tag{14}$$

Each layer l refines the approximation for each $\boldsymbol{t}_n$ using the codebook obtained in the previous iteration $l-1$. Using our DAG based interpretation, this is the same as effectively

[6]This assumption is valid as long as we use the same type of adder throughout a DAG, i.e. S is fixed.



(a) Fully Sequential Algorithm (b) Fully Parallel Algorithm (c) Mixed Alg. ($\Delta\mu_{\max} = 0$) (d) Mixed Alg. ($\Delta\mu_{\max} = 1$)

Fig. 2: Resulting graph topologies of different algorithmic approaches for decomposing a target matrix $\boldsymbol{T}$ of dimension 6×2. Green nodes represent input vertices corresponding to elements of the input vector $\boldsymbol{x}$, red nodes represent output vertices of the resulting matrix-vector product $\boldsymbol{y}$ and blue nodes are intermediary vertices of the decomposition graph.

restricting the codebook used in iteration l to the subset of vectors in $\mathcal{C}$ at depth $l-1$, i.e. the matrix $\boldsymbol{C}_{l-1}$ contains all row vectors that are also included in the set $\{\boldsymbol{c} \in \mathcal{C} | \mu_{\mathrm{D}}(\boldsymbol{c}) = l-1\}$. As for the FS algorithm, we use the set of all unit vectors as the initial codebook $\boldsymbol{C}_0 = \boldsymbol{I}_{N\times K}$. The structure of the DAG generated by this algorithm is depicted in Fig. 2b. Compared to the FS algorithm, a fully parallel implementation in hardware can be achieved, no delays by differing path lengths are introduced. However, this algorithm is not without drawbacks. First, previous work [5] showed that refinement of the initial codebook during the first few iteration comes with a drop in performance. Second, this algorithm does not scale to arbitrarily small matrices. As the effective codebook scales with the target matrix size this can lead to convergence issues when decomposing smaller matrices.

C. Mixed algorithm (MA)

Using the ideas of the FS and FP algorithms we introduce a new MA enabling us tune the structure of the computation DAG. We reuse the sequential update rule from the FS algorithm in (10) to update $\mathcal{C}$.

$$\tilde{n} = \underset{n\in[N]}{\operatorname{argmin}}\ \lambda_n \Bigg(\| \boldsymbol{t}_n - w(\boldsymbol{t}_n, \mathcal{C}, S)\, \mathrm{mat}(\mathcal{C}) \|_2^2 + \sum_{k\neq n} \| \boldsymbol{t}_k - w(\boldsymbol{t}_k, \mathcal{C}, 1)\, \mathrm{mat}(\mathcal{C}) \|_2^2 \Bigg) \tag{15a}$$

with

$$\lambda_n = \max_{j\in\mathcal{S}} \mu_{\mathrm{D}}(\boldsymbol{c}_j) \quad \text{and} \quad \mathcal{S} = \mathrm{supp}\left(w(\boldsymbol{t}_n, \mathcal{C}, S)\right).$$

To obtain the index $\tilde{n}$ for the target vector to be approximated, we amend the objective to update the approximation with the largest drop in error in (11) by a multiplicative penalty factor λ_n. This factor penalizes the absolute depth of approximations for different target vectors, i.e. updating a codeword at a higher depth leads to a larger penalty. Moreover, to be able to limit the number and depth of idle paths in the DAG, we introduce a side constraint limiting the difference in depth for any linear combination of codewords, which is

$$\max_{j\in\mathcal{S}} \left(\mu_{\mathrm{D}}(\boldsymbol{c}_j)\right) - \min_{j\in\mathcal{S}} \left(\mu_{\mathrm{D}}(\boldsymbol{c}_j)\right) \leq \Delta\mu_{\max}. \tag{15b}$$

The parameter $\Delta\mu_{\max}$ controls the maximum difference in depth for the codewords used in each update. For $\Delta\mu_{\max} \to \infty$ and $\lambda_n = 1$ the algorithm is equal to the FS algorithm. Constraining $\Delta\mu_{\max} = 0$ we obtain a parallel structure of the decomposition DAG, similar to the FP algorithm; however, codewords are added sequentially with a constraint on a parallel structure. In general, the constraint on depth lets us tune the structure of the graph with respect to parallelism. In Fig. 2c and 2d, the resulting graph structures for a graph constraint to a fully parallel structure and a depth difference of $\Delta\mu_{\max} = 1$ are depicted, respectively.

D. Related Algorithms

Most competing algorithms for CMVM have a decent time complexity for small matrices. However as they solve complex underlying problems, such as 0-1 integer linear programming [8], they do not scale well with growing matrix size and/or precision. They are hence often intractable. Instead, we use as a benchmark the best-performing MCM algorithm known, presented in [10], that has reasonable polynomial time complexity and is thus tractable for larger matrices as well. Note that MCM, the multiplication of a variable scalar to a arbitrary constant vector, is a special case of CMVM. Any CMVM problem can therefore be rewritten as a sum of K MCM problems, i.e.

$$\boldsymbol{y} = \boldsymbol{T}\boldsymbol{x} = \sum_{k=1}^{K} \boldsymbol{t}_k x_k, \tag{16}$$

that are solved independently. Here, $\boldsymbol{t}_k$ and x_k are the k-th column vector in $\boldsymbol{T}$ and k-th element in $\boldsymbol{x}$, respectively. Due to the reduced search space the benchmark MCM algorithm has excellent performance. However, the adder tree required for the summation of the K partial results, as well as a DAG structure, similar to the FS algorithm, limit the performance when pipelined.

IV. Numerical Experiments

The entries of all target matrices in the subsequent evaluations are drawn from an i.i.d. Gaussian distribution with zero mean and unit variance. We expect that for practical matrices, e.g. weight matrices of NNs, similar performance is observed for LCC algorithms [17]. A Python implementation of all algorithms discussed in this paper is available in our *github* repository: https://github.com/hansrosenberger/computationcoding.

As the first experiment, we compare the different algorithms for target matrices of dimension 64×4 in Fig. 3. The figure shows, the FS algorithm achieves the highest SQNR, considering only the cost of additions (dashed lines). However, when considering the total hardware cost, the FS performance massively deteriorates, leaving this algorithm impractical for a pipelined implementation. The overall hardware cost in this case is dominated by delay elements required to equalize path differences within the DAG. The MA constrained to a FP structure shows the best overall performance, when considering

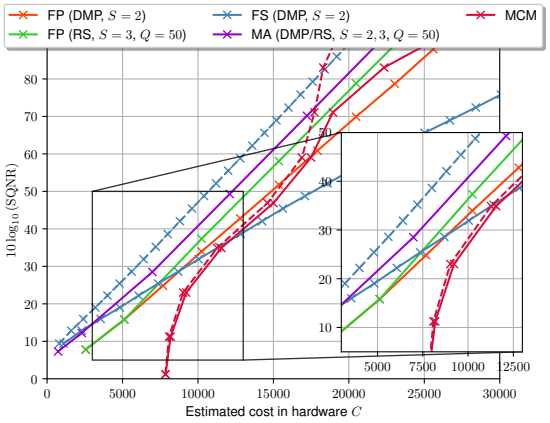


Fig. 3: Comparison of different algorithmic approaches for decomposing a 64×4 target matrix $\boldsymbol{T}$. Solid lines indicate results considering the total cost C_{total}. Dashed lines only consider the cost of adders $C_{\text{add}} N_{\text{add}}$. MCM refers to the algorithm presented in [10] (using the C++ implementation available on [16] and extended by our hardware model). The results for each algorithm are averaged over 10^5 matrix entries.

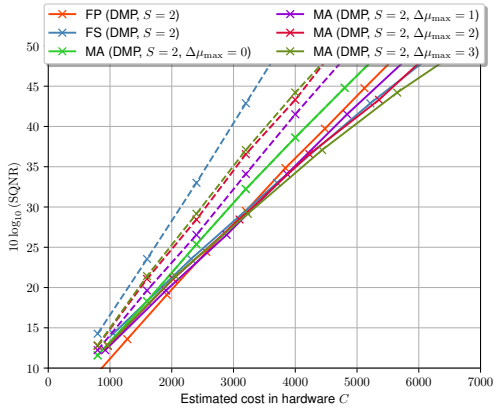


Fig. 4: Comparison of different depth parameters $\Delta\mu_{\max}$ of the MA given a 16×4 target matrix $\boldsymbol{T}$. Solid lines indicate results considering the total cost C_{total}. Dashed lines only consider the cost of adders $C_{\text{add}} N_{\text{add}}$. The results for each algorithm are averaged over 10^5 matrix entries.

the total hardware cost. It outperforms the FP algorithm, both the DMP and RS versions. Relative gains are particulary large for the low SQNR regime. This is achieved by first setting $S = 2$ and utilizing the DMP to build up a coarse codebook from the initial codebook, and then dynamically switching to $S = 3$ via the RS approach. The savings of MA to the FS result from an improved structure of the DAG for the first few layers. The FP algorithm is forced to find an approximation for each target vector separately. This creates codewords that are correlated and unnecessary for the computation. The MA eliminates this redundancy (cf. Figs. 2b and 2c).

As the second experiment we compare the performance of the MA using different depth parameters $\Delta\mu_{\max}$ for target matrices of dimension 16×4 in Fig. 4. Considering only the cost of the adders (dashed lines), we can clearly observe a tradeoff between parallelism and performance, i.e. decreasing $\Delta\mu_{\max}$ leads to a performance degradation. However, when considering the total hardware cost (solid lines) the MA performs best when constrained to a FP structure ($\Delta\mu_{\max} = 0$). For $\Delta\mu_{\max} > 0$ the MA performs worse than its FP counterpart and for some instances even worse than the FS algorithm. This result seems somewhat intuitive: Elements that incur a hardware cost that is not vanishingly small should also improve the SQNR. Hence, a fully parallel structure seems to be the best option.

Remark 2: LCC works best for matrices with an exponential aspect ratio, i.e. $K \approx \log N$. Therefore, we only consider in the evaluation matrices with that property. For approximately square matrices it is beneficial to cut these into rectangular matrices with more extreme aspect ratios and apply an LCC algorithm to each slice individually [18]. For example, to decompose a 64×64 matrix with a target SQNR of $47\,\text{dB}$, a slicing into submatrices of size 64×4 is a good choice.

V. Conclusion

By interpreting the decomposition of a matrix as a DAG, we proposed a new MA for LCC. The proposed algorithm is able to significantly outperform existing schemes. Using a realistic hardware model for pipelining, we show that in almost all cases it is best to decompose a target matrix constraining the resulting DAG to a parallel structure.

References

[1] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021. arXiv:2103.13630.

[2] J. O. Neill, "An overview of neural network compression," 2020. arXiv:2006.03669.

[3] R. R. Müller, B. M. W. Gäde, and A. Bereyhi, "Linear computation coding: A framework for joint quantization and computing," *Algorithms*, vol. 15, no. 7, p. 253, 2022.

[4] R. R. Müller, "Linear computation coding inspired by the Lempel-Ziv algorithm," in *2022 IEEE Information Theory Workshop (ITW)*, IEEE, 2022.

[5] H. Rosenberger, J. S. Fröhlich, A. Bereyhi, and R. R. Müller, "Linear computation coding: Exponential search and reduced-state algorithms," in *2023 Data Compression Conference (DCC)*, IEEE, 2023.

[6] A. Karataev, H. Rosenberger, A. Bereyhi, and R. R. Müller, "Storage constrained linear computation coding," in *2023 Data Compression Conference (DCC)*, IEEE, 2023.

[7] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[8] L. Aksoy, P. Flores, and J. Monteiro, "A novel method for the approximation of multiplierless constant matrix vector multiplication," in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, IEEE, 2015.

[9] S. Foucart and H. Rauhut, *A Mathematical Introduction to Compressive Sensing*. Springer New York, 2013.

[10] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Transactions on Algorithms*, vol. 3, no. 2, p. 11, 2007.

[11] P. Cappello and K. Steiglitz, "Some complexity issues in digital signal processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 5, pp. 1037–1041, 1984.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.

[13] U. Tietze, C. Schenk, and E. Gamm, *Halbleiter-Schaltungstechnik*. Springer Vieweg Berlin, Heidelberg, 16 ed., 2019.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, Morgan Kaufmann, 5 ed., 2012.

[15] A. Lehnert, H. Rosenberger, R. Müller, and M. Reichenbach, "More efficient CMMs on FPGAs: Instantiated ternary adders for computation coding," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pp. 275–289, Springer Nature Switzerland, 2023.

[16] "Spiral: Software/hardware generation for performance: Multiplier block generator." https://spiral.ece.cmu.edu/mcm/gen.html. Accessed: 15.04.2023.

[17] R. R. Müller, H. Rosenberger, and M. Reichenbach, "Linear computation coding for convolutional neural networks," in *Statistical Signal Processing (SSP) Workshop*, (Hanoi, Vietnam), 2023.

[18] A. Lehnert, P. Holzinger, S. Pfenning, R. Müller, and M. Reichenbach, "Most resource efficient matrix vector multiplication on FPGAs," *IEEE Access*, vol. 11, pp. 3881–3898, 2023.