MATRIX ADDITION, HADAMARD PRODUCTS, AND MATRIX MULTIPLICATION

1. INTRODUCTION

Matrix multiplication is a core operation in the forward pass of neural networks, underlying the computations performed by fully connected layers—where input activations are combined with learned weights to produce subsequent activations. The Hadamard product—defined as the elementwise product of two matrices of the same shape—also plays an essential role, particularly in gating mechanisms common in recurrent networks and attention modules, where it modulates information flow on a per-coordinate basis.

In this document, we develop methods for verifying integer relations involving these fundamental operations and their linear combinations within arithmetic circuits. Specifically, we consider integer-valued matrices without assuming any scaling or quantization. (Verification involving matrices scaled for fixed-point representations—known as *quantized matrix multiplication*—is treated in a separate companion document [1].)

The overarching verification strategy employed throughout this document is as follows: we begin with an integer equality that we aim to verify within an arithmetic circuit over a finite field $\mathbb{Z}/p\mathbb{Z}$. Directly enforcing integer equalities is impossible in modular arithmetic, as a congruence modulo p only ensures equality up to some multiple of p. Therefore, the core verification challenge is to guarantee—through the application of carefully chosen inequalities and range checks—that this multiple of p must be zero. Since all integers are first converted into least-residue representatives modulo p *before* entering the circuit, the circuit itself cannot distinguish between distinct integers differing by multiples of p. Consequently, establishing that inputs lie within a predetermined complete residue system modulo p necessarily relies on off-circuit justifications and assumptions. This interplay between modular constraints (enforced within the circuit) and integer range assumptions (justified off-circuit) is fundamental to all arithmetic circuit verifications.

With this verification paradigm in place, we begin by examining the simplest scenario—*matrix addition*—which establishes the basic structure and reasoning behind our verification process. Subsequently, we consider the *Hadamard product*, structurally analogous to addition but involving elementwise multiplication. Finally, we address *matrix multiplication* in its "vanilla" (non-quantized) form, as well as generalized expressions of the form

$$\alpha AB + \beta C,$$

where A, B, and C are integer matrices of compatible dimensions, and α, β are known integer scalars.

Together, these foundational operations encompass a wide variety of neural network architectures—from simple affine transformations to sophisticated gating and attention mechanisms—enabling efficient and rigorous zero-knowledge verification of neural network behavior.

2. MATRIX ADDITION: THE PROCESS

Let $A_1, \ldots, A_k$ be matrices of the same dimension $m \times n$, and let $\alpha_1, \ldots, \alpha_k$ be integer scalars. We define the weighted sum

$$B = \alpha_1 A_1 + \cdots + \alpha_k A_k,$$

with $B = [b_{ij}]$, $A_t = [a_{ij}^{(t)}]$ for $t = 1, \dots, k$, and

$$b_{ij} = \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}. \tag{2.1}$$

Verifying such a matrix addition within an arithmetic circuit amounts to checking mn independent linear constraints over $\mathbb{Z}/p\mathbb{Z}$.

This operation arises naturally in affine layers, residual blocks, and many preprocessing or postprocessing steps in neural networks. The constraints are simple to implement: we verify that

$$b_{ij} \equiv \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)} \bmod p.$$

However, we must ensure that the integer representation of each sum lies within a known range to interpret the result unambiguously.

2.1. Steps in the process.

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime.

(2) Let $A_1 = [a_{ij}^{(1)}], \dots, A_k = [a_{ij}^{(k)}], B = [b_{ij}]$ be integer matrices of dimensions $m \times n$, and let $\alpha_1, \dots, \alpha_k$ be integer scalars. Assume there is an integer h such that for all i, j,

$$h - p \leqslant \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}, b_{ij} < h. \tag{2.2}$$

(3) The goal of the circuit is to verify that $\alpha_1 A_1 + \cdots + \alpha_k A_k = B$, that is, for all i, j,

$$b_{ij} = \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}. \tag{2.3}$$

(4) Most frameworks work exclusively with least residue representations. Accordingly, we use $\bar{x}$ to denote the least residue of x mod p: for all i, j, ℓ,

$$\bar{\alpha}_\ell \equiv \alpha_\ell \bmod p, \quad \bar{a}_{ij}^{(\ell)} \equiv a_{ij}^{(\ell)} \bmod p, \quad \bar{b}_{ij} \equiv b_{ij} \bmod p, \quad 0 \leqslant \bar{\alpha}_\ell, \bar{a}_{ij}^{(\ell)}, \bar{b}_{ij} < p. \tag{2.4}$$

(5) For all i, j, impose the constraint

$$\left(\bar{\alpha}_1 \bar{a}_{ij}^{(1)} + \cdots + \bar{\alpha}_k \bar{a}_{ij}^{(k)}\right) - \bar{b}_{ij} \equiv 0 \bmod p. \tag{2.5}$$

(6) Given that the bounds in (2.2) hold and the constraints (2.5) are satisfied for all i, j, we may conclude that the original integer relation (2.3) holds entrywise [Proposition 5.1]. That is, under our assumptions, the congruences (2.5) correctly enforce the desired matrix sum

$$\alpha_1 A_1 + \cdots + \alpha_k A_k = B. \tag{2.6}$$

2.2. Commentary on each step.

(1) Typically, p is an n-bit prime satisfying

$$2^{n-1} \leqslant p < 2^n,$$

with $n \approx 256$. In practice, we use the prime field associated with the scalar field of the BN254 elliptic curve, a 254-bit prime that offers a good balance between security and efficiency. This field is widely supported in cryptographic applications and zk-SNARK frameworks due to the curve's pairing-friendly properties and efficient arithmetic over $\mathbb{Z}/p\mathbb{Z}$.

(2) The offset parameter h defines an integer interval of length p within which each weighted sum and each b_{ij} is assumed to lie. Two common conventions are:

- the *least residue range* $[0, p)$, corresponding to $h = p$;
- the *balanced residue range* $(-p/2, p/2]$, corresponding to $h = (p+1)/2$.

As with any field-based circuit, the circuit cannot distinguish between integers that differ by a multiple of p, so the bound

$$h - p \leqslant b_{ij} < h$$

must be justified off-circuit.

The same applies to the weighted sum. In many cases, we may assume the range bound

$$h - p \leqslant \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)} < h$$

holds automatically due to the size of p and small magnitudes of the scalars and matrix entries. However, if explicit justification is needed, we may enforce a bound in-circuit by verifying that

$$a_{ij}^{(\ell)} \in [-U, U) \quad (1 \leqslant \ell \leqslant k),$$

for some positive integer U satisfying

$$(|\alpha_1| + \cdots + |\alpha_k|)\,U < p/2.$$

This ensures that the sum lies in $(-p/2, p/2)$. However, as before, this reasoning assumes each $a_{ij}^{(\ell)}$ is represented within a known residue interval of length p. Without that, even a valid range check cannot be interpreted unambiguously. For further discussion of this subtlety and the implementation of range checks, see our companion documents [2, 3].

(3) In other words, we wish to verify the correctness of mn independent weighted sums.

(4) In typical usage, the prover supplies the least residues $\bar{A}_1, \ldots, \bar{A}_k$, and $\bar{B}$ as part of the witness, where each entry is the canonical representative modulo p. If any matrix $\bar{A}_\ell$ is fixed—e.g., a constant bias matrix—it may be incorporated into the circuit directly. Likewise, the scalars $\bar{\alpha}_\ell$ are the least residues of known integers α_ℓ and may be either public parameters or hardcoded constants, depending on the application.

(5) Although $\bar{\alpha}_\ell \equiv \alpha_\ell \bmod p$, $\bar{a}_{ij}^{(\ell)} \equiv a_{ij}^{(\ell)} \bmod p$, and $\bar{b}_{ij} \equiv b_{ij} \bmod p$, we state the constraint in terms of the least residues to reflect what is literally enforced in the circuit. The prover supplies $\bar{a}_{ij}^{(\ell)}$ and $\bar{b}_{ij}$ as field elements in $[0, p)$, and the constraint

$$\left(\bar{\alpha}_1 \bar{a}_{ij}^{(1)} + \cdots + \bar{\alpha}_k \bar{a}_{ij}^{(k)}\right) - \bar{b}_{ij} \equiv 0 \bmod p$$

is enforced using native field operations in the arithmetic circuit. The distinction between x and $\bar{x}$ is invisible to the circuit at the field level, but helps clarify the origin of each quantity.

(6) From (2.5) and (2.4), we have

$$\left(\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}\right) - b_{ij} \equiv 0 \bmod p,$$

so there exists an integer t such that

$$\left(\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}\right) - b_{ij} = tp.$$

Using (2.2), we know

$$h - p \leqslant \left(\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}\right), b_{ij} \leqslant h - 1.$$

(All terms involved are integers, so a strict inequality like $b_{ij} < h$ is equivalent to $b_{ij} \leqslant h - 1$.) Subtracting gives

$$(h-p) - (h-1) \leqslant \left(\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}\right) - b_{ij} \leqslant (h-1) - (h-p),$$

i.e.,

$$-(p-1) \leqslant \left(\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)}\right) - b_{ij} \leqslant p - 1.$$

Hence $tp \in (-p, p)$, implying $t \in (-1, 1)$, so $t = 0$, and therefore

$$\alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)} = b_{ij}.$$

3. Hadamard Product: The Process

Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be matrices of the same dimension $m \times n$. Their *Hadamard product*, denoted $A \odot B$, is the matrix $C = [c_{ij}]$, also of dimension $m \times n$, defined by

$$c_{ij} = a_{ij} b_{ij}. \tag{3.1}$$

This operation arises naturally in gating mechanisms and feature-wise modulations in neural networks, and is efficient to verify in constraint systems due to its lack of cross-term interactions.

Verifying a Hadamard product computation within an arithmetic circuit amounts to checking the validity of mn independent multiplication constraints. These constraints are straightforward to implement: for each i, j, we simply verify that

$$c_{ij} \equiv a_{ij} b_{ij} \bmod p$$

However, careful attention must be paid to the range and encoding of the input and output values to ensure correctness at the integer level.

We consider a slightly more general setting: given integer matrices A, B, C, and D of equal dimensions and integer scalars α, β, we verify that

$$\alpha(A \odot B) + \beta C = D.$$

This generalization incurs little additional complexity in the constraint system but captures a broader class of common structures, including residual connections and affine gates.

3.1. Steps in the process. Each step in the process has a corresponding explanatory note in Subsection 3.2 that provides additional context and details.

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime.

(2) Let $A = [a_{ij}], B = [b_{ij}], C = [c_{ij}], D = [d_{ij}]$ be integer matrices of dimensions $m \times n$. Let α and β be integers. Assume there is an integer h such that for all i, j,

$$h - p \leqslant \alpha a_{ij} b_{ij} + \beta c_{ij},\, d_{ij} < h. \tag{3.2}$$

(3) The goal of the circuit is to verify that $\alpha(A \odot B) + \beta C = D$. That is, for all i, j,

$$\alpha a_{ij} b_{ij} + \beta c_{ij} = d_{ij}. \tag{3.3}$$

(4) Most frameworks work exclusively with least residue representations. Accordingly, we use $\bar{x}$ to denote the least residue of x mod p:

$$\bar{\alpha} \equiv \alpha \bmod p, \quad \bar{\beta} \equiv \beta \bmod p, \quad 0 \leqslant \bar{\alpha}, \bar{\beta} < p,$$

and for all i, j,

$$\bar{a}_{ij} \equiv a_{ij} \bmod p, \quad \bar{b}_{ij} \equiv b_{ij} \bmod p, \quad \bar{c}_{ij} \equiv c_{ij} \bmod p, \quad \bar{d}_{ij} \equiv d_{ij} \bmod p, \quad 0 \leqslant \bar{a}_{ij}, \bar{b}_{ij}, \bar{c}_{ij}, \bar{d}_{ij} < p. \tag{3.4}$$

(5) For all i, j, impose the constraint

$$\bar{\alpha}\bar{a}_{ij}\bar{b}_{ij} + \bar{\beta}\bar{c}_{ij} - \bar{d}_{ij} \equiv 0 \bmod p. \tag{3.5}$$

(6) Given that the bounds in (3.2) hold and the constraints (3.5) are satisfied for all i, j, we may conclude that the original integer relation (3.3) holds entrywise [Proposition 5.1]. That is, under our assumptions, the congruences (3.5) correctly enforce the desired Hadamard product

$$\alpha(A \odot B) + \beta C = D. \tag{3.6}$$

3.2. Commentary on each step.

(1) See Comment (1) of Subsection 2.2.

(2) The offset parameter h provides flexibility in defining an integer range of length p to which all values are assumed to belong. Prior to entering the circuit, we typically assume that each term $\alpha a_{ij} b_{ij} + \beta c_{ij}$ and each value d_{ij} lies within such a range. Two common conventions are:

- the *least residue range* $[0, p)$, corresponding to $h = p$;
- the *balanced residue range* $(-p/2, p/2]$, corresponding to $h = (p+1)/2$.

Because arithmetic circuits over $\mathbb{Z}/p\mathbb{Z}$ cannot distinguish between integers that differ by a multiple of p, the condition

$$h - p \leqslant d_{ij} < h$$

cannot be enforced within the circuit itself. It must instead be justified through off-circuit reasoning, either by construction of the inputs or by assumption.

In most practical scenarios, the corresponding bound on the product,

$$h - p \leqslant \alpha a_{ij} b_{ij} + \beta c_{ij} < h,$$

can likely also be justified off-circuit. However, if such justification is not available, we may instead enforce the product bound in-circuit by performing a range check to verify that

$$a_{ij}, b_{ij}, c_{ij} \in [-U, U)$$

for some positive integer U satisfying $|\alpha|U^2 + |\beta|U < p/2$. In that case, it follows that

$$\alpha a_{ij} b_{ij} + \beta c_{ij} \in \left[-\left(|\alpha|U^2 + |\beta|U\right), |\alpha|U^2 + |\beta|U\right] \subseteq (-p/2, p/2),$$

so the product lies within the balanced residue range.

Still, the soundness of this range check implicitly relies on the assumption that a_{ij}, b_{ij}, c_{ij} are themselves represented in the balanced residue range $(-p/2, p/2]$, or some other known range of length p, to begin with. Without such an assumption, the result of the range check cannot be interpreted unambiguously. For further discussion of this subtlety and the implementation of range checks, see our companion documents [2, 3].

(3) In other words, the goal is to verify mn independent multiplications.

(4) In typical usage, the prover supplies the least residues $\bar{A} = [\bar{a}_{ij}]$, $\bar{B} = [\bar{b}_{ij}]$, $\bar{C} = [\bar{c}_{ij}]$, and $\bar{D} = [\bar{d}_{ij}]$ as part of the witness, where each entry is the canonical representative of its corresponding integer modulo p. Although it is possible to compute $\bar{D}$ in-circuit from $\bar{A}, \bar{B}, \bar{C}$, doing so increases circuit size and is unnecessary for most modular zkML designs.

If B or C represent fixed matrices—such as model weights or bias terms—they may be incorporated directly into the circuit as constant values. This eliminates the need for the prover to supply $\bar{B}$ or $\bar{C}$ as part of the witness and can improve both prover performance and circuit compactness.

Likewise, the scalars $\bar{\alpha}$ and $\bar{\beta}$ may either be treated as public parameters or compiled into the circuit as fixed constants. In many practical scenarios—such as residual connections or affine gates—these coefficients are known in advance and can be hardcoded to minimize input complexity.

(5) Although $\bar{\alpha} \equiv \alpha \bmod p$, and similarly for $\bar{\beta}, \bar{a}_{ij}, \bar{b}_{ij}$, $\bar{c}_{ij}$, and $\bar{d}_{ij}$, we state the constraint in terms of the least residue representatives to reflect what is literally enforced in the circuit. The prover supplies $\bar{a}_{ij}, \bar{b}_{ij}, \bar{c}_{ij}, \bar{d}_{ij} \in [0, p)$ as field elements, and the constraint $\bar{\alpha}\bar{a}_{ij}\bar{b}_{ij} + \bar{\beta}\bar{c}_{ij} - \bar{d}_{ij} \equiv 0 \bmod p$ is imposed using arithmetic over $\mathbb{Z}/p\mathbb{Z}$. The distinction between x and $\bar{x}$ is not meaningful within the field, but it helps clarify the provenance of the values and emphasizes that no signed interpretation occurs within the circuit itself.

(6) For all i, j, we have from (3.4) and (3.5) that

$$\alpha a_{ij} b_{ij} + \beta c_{ij} - d_{ij} \equiv 0 \bmod p,$$

and hence there exists an integer t such that

$$\alpha a_{ij} b_{ij} + \beta c_{ij} - d_{ij} = tp. \tag{3.7}$$

By the assumption (3.2), we know that

$$h - p \leqslant \alpha a_{ij} b_{ij} + \beta c_{ij},\, d_{ij} \leqslant h - 1.$$

(All terms involved are integers, so a strict inequality like $d_{ij} < h$ is equivalent to $d_{ij} \leqslant h - 1$.) Subtracting these bounds gives

$$(h-p) - (h-1) \leqslant (\alpha a_{ij} b_{ij} + \beta c_{ij}) - d_{ij} \leqslant (h-1) - (h-p),$$

which simplifies to

$$-(p-1) \leqslant (\alpha a_{ij} b_{ij} + \beta c_{ij}) - d_{ij} \leqslant p - 1. \tag{3.8}$$

Substituting from (3.7), we conclude that $tp \in (-p, p)$, so $t \in (-1, 1)$, and hence $t = 0$. It follows that

$$\alpha a_{ij} b_{ij} + \beta c_{ij} = d_{ij}.$$

4. Matrix Multiplication: The Process

Matrix multiplication lies at the heart of neural network computation. It is the fundamental operation performed by fully connected layers, where the output activations are computed as a linear transformation of the inputs using a learned weight matrix. Given integer matrices $A = [a_{ik}]$ of dimensions $\ell \times m$ and $B = [b_{kj}]$ of dimensions $m \times n$, their product is the matrix $C = [c_{ij}]$ of dimensions $\ell \times n$, with entries defined by

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}. \tag{4.1}$$

To verify this relation within an arithmetic circuit, we interpret the sum in (4.1) as a composition of multiplication and addition gates over a finite field. At a high level, we impose a constraint for each output entry c_{ij} that asserts it is equal to the inner product of the i-th row of A with the j-th column of B. As with the other operations in this document, we work over the finite field $\mathbb{Z}/p\mathbb{Z}$, and therefore must ensure that all inputs and outputs are appropriately encoded, and that the integer-level equality is correctly recovered from modular constraints using known bounds on the input values.

4.1. Steps in the process. Each step in the process has a corresponding explanatory note in Subsection 4.2 that provides additional context and details.

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is a prime.

(2) Let $A = [a_{ik}], B = [b_{kj}], C = [c_{ij}], D = [d_{ij}]$ be integer matrices of dimensions $\ell \times m, m \times n, \ell \times n$, respectively. Let α and β be integers.

(2a) Assume there is an integer h such that for all i, k, j,

$$h - p \leqslant \alpha \left(\sum_{k=1}^{m} a_{ik} b_{kj} \right) + \beta c_{ij}, d_{ij} < h. \tag{4.2}$$

(2b) If this assumption cannot be justified, then assume there is an integer h such that for all i, k, j,

$$h - p \leqslant a_{ik}, b_{kj}, c_{ij}, d_{ij} < h, \tag{4.3}$$

and perform a range check to ensure that

$$a_{ik}, b_{kj}, c_{ij} \in [-U, U), \tag{4.4}$$

for some positive integer U satisfying

$$|\alpha| m U^2 + |\beta| U < \min\{-(h-p), p\}. \tag{4.5}$$

(3) The goal of the circuit is to verify that $\alpha(AB) + \beta C = D$. That is, for all i, j,

$$\alpha \left(\sum_{k=1}^{m} a_{ik} b_{kj} \right) + \beta c_{ij} = d_{ij}. \tag{4.6}$$

(4) Most frameworks work exclusively with least residue representations. Accordingly, we use $\bar{x}$ to denote the least residue of x mod p:

$$\bar{\alpha} \equiv \alpha \bmod p, \quad \bar{\beta} \equiv \beta \bmod p, \quad 0 \leqslant \bar{\alpha}, \bar{\beta} < p,$$

and for all i, k, j,

$$\bar{a}_{ik} \equiv a_{ik} \bmod p, \quad \bar{b}_{kj} \equiv b_{kj} \bmod p, \quad \bar{c}_{ij} \equiv c_{ij} \bmod p, \quad \bar{d}_{ij} \equiv d_{ij} \bmod p, \quad 0 \leqslant \bar{a}_{ik}, \bar{b}_{kj}, \bar{c}_{ij}, \bar{d}_{ij} < p. \tag{4.7}$$

(5) For all i, j, impose the constraint

$$\bar{\alpha} \left(\sum_{k=1}^{m} \bar{a}_{ik} \bar{b}_{kj} \right) + \bar{\beta} \bar{c}_{ij} - \bar{d}_{ij} \equiv 0 \bmod p. \tag{4.8}$$

(6) Given that the bounds in (4.2) hold and the constraints (4.8) are satisfied for all i, j, we may conclude that the original integer relation (4.6) holds entrywise [Proposition 5.1]. That is, under our assumptions, the congruences (3.5) correctly enforce the desired generalized matrix product

$$\alpha(AB) + \beta C = D. \tag{4.9}$$

4.2. Commentary on each step.

(1) See Comment (1) of Subsection 2.2.

(2) The offset parameter h provides flexibility in defining an integer interval of length p to which all values are assumed to belong. Prior to entering the circuit, we typically assume that each term $\alpha\left(\sum_{k=1}^{m} a_{ik} b_{kj}\right) + \beta c_{ij}$ and each value d_{ij} lies within such a range. Two common conventions are:

- the *least residue range* $[0, p)$, corresponding to $h = p$;
- the *balanced residue range* $(-p/2, p/2]$, corresponding to $h = (p+1)/2$.

Because arithmetic circuits over $\mathbb{Z}/p\mathbb{Z}$ cannot distinguish between integers that differ by a multiple of p, the condition

$$h - p \leqslant d_{ij} < h$$

must be justified off-circuit.

The same applies to the term $\alpha\left(\sum_{k=1}^{m} a_{ik} b_{kj}\right) + \beta c_{ij}$. In many scenarios, such bounds may be justified off-circuit, particularly when m is small and the matrix entries are known to be bounded. However, when such justification is unavailable, we must instead enforce the bound in-circuit by verifying that

$$a_{ik}, b_{kj}, c_{ij} \in [-U, U)$$

for some positive integer U satisfying

$$|\alpha|mU^2 + |\beta|U < \min\{-(h-p), p\}.$$

This guarantees that

$$\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij} \in \left[-(|\alpha|mU^2 + |\beta|U),\, |\alpha|mU^2 + |\beta|U\right] \subseteq (h-p, h). \tag{4.10}$$

As always, the validity of the range check depends on the off-circuit assumption that a_{ik}, b_{kj}, and c_{ij} are encoded using a known, consistent set of representatives modulo p (e.g., the balanced residue range). Without such an assumption, even a successful range check does not determine the underlying integer values uniquely. For further discussion of this subtlety and the implementation of range checks, see our companion documents [2, 3].

(3) The goal is to verify ℓn constraints, one for each entry in the output matrix D. Each constraint corresponds to a generalized inner product, involving the weighted sum of a row of A and a column of B, followed by an affine shift by βc_{ij}. The structure resembles an inner product but includes both a scalar multiplier α and an additive offset.

(4) In typical usage, the prover supplies the least residues $\bar{A} = [\bar{a}_{ik}]$, $\bar{B} = [\bar{b}_{kj}]$, $\bar{C} = [\bar{c}_{ij}]$, and $\bar{D} = [\bar{d}_{ij}]$ as part of the witness, where each entry is the canonical representative of its corresponding integer modulo p. Although it is possible to compute $\bar{D}$ in-circuit from $\bar{A}, \bar{B}, \bar{C}$, doing so increases circuit size and is unnecessary in most modular zkML designs.

If B or C represent fixed matrices—such as learned model weights or constant bias terms—they may be hardcoded into the circuit. This eliminates the need for the prover to supply $\bar{B}$ or $\bar{C}$ and reduces prover-side workload.

Likewise, the scalars $\bar{\alpha}$ and $\bar{\beta}$ may be compiled into the circuit as fixed constants or supplied as public parameters, depending on the use case. In most inference settings, these coefficients are known in advance and are typically hardcoded to reduce input complexity and prover effort.

(5) Although $\bar{\alpha} \equiv \alpha \bmod p$, and similarly for $\bar{\beta}, \bar{a}_{ik}, \bar{b}_{kj}$, $\bar{c}_{ij}$, and $\bar{d}_{ij}$, we express the constraint in terms of least residues to reflect the actual field-level expressions used in circuit implementation. The prover supplies $\bar{a}_{ij}, \bar{b}_{ij}, \bar{c}_{ij}, \bar{d}_{ij} \in [0, p)$ as field elements, and the constraint

$$\bar{\alpha}\left(\sum_{k=1}^{m} \bar{a}_{ik}\bar{b}_{kj}\right) + \bar{\beta}\bar{c}_{ij} - \bar{d}_{ij} \equiv 0 \bmod p$$

is enforced using arithmetic over $\mathbb{Z}/p\mathbb{Z}$. The distinction between x and $\bar{x}$ is invisible to the circuit but clarifies how values are encoded at preprocessing time.

(6) For all i, j, we have from (4.7) and (4.8) that

$$\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij} - d_{ij} \equiv 0 \bmod p,$$

and hence there exists an integer t such that

$$\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij} - d_{ij} = tp. \tag{4.11}$$

From assumption (4.2) (alternatively, (4.10)), we know that

$$h - p \leqslant \alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij},\, d_{ij} \leqslant h-1.$$

(All terms involved are integers, so a strict inequality like $d_{ij} < h$ is equivalent to $d_{ij} \leqslant h-1$.) Subtracting gives

$$(h-p)-(h-1) \leqslant \left(\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij}\right) - d_{ij} \leqslant (h-1)-(h-p),$$

i.e.,

$$-(p-1) \leqslant \left(\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij}\right) - d_{ij} \leqslant p-1.$$

Substituting from (4.11), we conclude that $tp \in (-p, p)$, so $t \in (-1, 1)$ and hence $t = 0$. It follows that

$$\alpha\left(\sum_{k=1}^{m} a_{ik}b_{kj}\right) + \beta c_{ij} = d_{ij}.$$

That is, under our assumptions, the congruences (4.8) correctly enforce the desired generalized matrix product

$$\alpha(AB) + \beta C = D. \tag{4.12}$$

5. The Maths

Proposition 5.1. *Let p be a positive integer (not necessarily a prime), and let integers h, x, y satisfy*

$$h - p \leqslant x, y < h. \tag{5.1}$$

Let $\bar{x}, \bar{y}$ denote the least residues of x and y modulo p, respectively. Then $x - y = 0$ if and only if

$$\bar{x} - \bar{y} \equiv 0 \bmod p. \tag{5.2}$$

Remark 5.2. (i) While one may trivially replace $\bar{x} - \bar{y}$ with $x - y$ in (5.2), we state the congruence in terms of least residues to reflect the circuit-level perspective adopted throughout this document. Arithmetic circuits operate over the field $\mathbb{Z}/p\mathbb{Z}$ and thus "see" only the least residue representatives of integers.

(ii) In Subsection 2.1, Step (2) assumes that x and y lie in an interval of length p:

$$h - p \leqslant x, y < h,$$

with

$$x = \alpha_1 a_{ij}^{(1)} + \cdots + \alpha_k a_{ij}^{(k)} \quad \text{and} \quad y = b_{ij}.$$

Step (5) imposes the constraint $\bar{x} - \bar{y} \equiv 0 \bmod p$, and Step (6) concludes that $x - y = 0$, i.e. $x = y$. The same pattern appears—with corresponding substitutions—in the Hadamard product and matrix multiplication cases (Subsections 3.1 and 4.1). Thus, Proposition 5.1 provides a unified justification for the conclusions drawn in all three constructions. ■

Proof of Proposition 5.1. If $x - y = 0$, then $\bar{x} - \bar{y} \equiv x - y \equiv 0 \bmod p$. For the converse, first note that as $\bar{x} \equiv x \bmod p$ and $\bar{y} \equiv y \bmod p$ by definition, (5.2) is equivalent to

$$x - y \equiv 0 \bmod p.$$

This congruence holds if and only if there is an integer t such that

$$x - y = tp. \tag{5.3}$$

Since h, x, y are integers, the assumption (5.1) implies that x, y lie in the closed interval

$$h - p \leqslant x, y \leqslant h - 1.$$

Subtracting, we obtain

$$-(p-1) = h - p - (h-1) \leqslant x - y \leqslant (h-1) - (h-p) = p - 1.$$

If (5.3) holds, then

$$tp = x - y \in [-(p-1), p-1] \subseteq (-p, p),$$

so $t \in (-1, 1) \cap \mathbb{Z} = \{0\}$. Hence $x - y = 0$. □

6. The Code

Rust code in this section is designed for implementation within the ExpanderCompilerCollection (ECC) framework [4, 5]. This library provides a specialized interface for constructing and verifying arithmetic circuits.

6.1. Matrix Addition. We begin with the matrix addition operation, where the circuit verifies that a weighted sum of input matrices equals a target output matrix:

$$\bar{B} = \bar{\alpha}_1 \bar{A}_1 + \cdots + \bar{\alpha}_k \bar{A}_k.$$

Below, we provide the corresponding pseudocode and its implementation using the ExpanderCompilerCollection Rust API.

Algorithm 6.1 `verify_matrix_addition`: verify that $\bar{B} = \sum_{\ell=1}^{k} \bar{\alpha}_\ell \cdot \bar{A}_\ell$

```
Require: Matrices $\bar{A}_1, \ldots, \bar{A}_k : m \times n$, $\bar{B} : m \times n$, scalars $\bar{\alpha}_1, \ldots, \bar{\alpha}_k \in \mathbb{Z}/p\mathbb{Z}$
Require: All entries are circuit variables representing least residues modulo $p$
function VERIFY_MATRIX_ADDITION($\bar{A}_1, \ldots, \bar{A}_k, \bar{B}, \bar{\alpha}_1, \ldots, \bar{\alpha}_k$)
    for $i \leftarrow 0$ to $m-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            $lhs \leftarrow 0$
            for $\ell \leftarrow 1$ to $k$ do
                $lhs \leftarrow lhs + (\bar{\alpha}_\ell \cdot \bar{a}_{ij}^{(\ell)})$
            end for
            assert_is_equal($\bar{b}_{ij}, lhs$)
        end for
    end for
end function
```

```rust
const M: usize = 3; // Number of rows
const N: usize = 2; // Number of columns
const K: usize = 4; // Number of input matrices A_1, ..., A_k

declare_circuit!(Circuit {
    matrices_a: [[[Variable; N]; M]; K],  // Array of A_ell: (M x N) matrices
    matrix_b: [[Variable; N]; M],          // B: (M x N)
    alphas: [Value<C::Field>; K],          // Scalars alpha_ell
});

impl<C: Config> Define<C> for Circuit<Variable> {
    fn define<Builder: RootAPI<C>>(&self, api: &mut Builder) {
        for i in 0..M {
            for j in 0..N {
                let mut acc = api.constant(C::Field::zero());

                for ell in 0..K {
                    let prod = api.mul(self.matrices_a[ell][i][j], self.alphas[ell]);
                    acc = api.add(acc, prod);
                }

                api.assert_is_equal(self.matrix_b[i][j], acc);
            }
        }
    }
}
```

Listing 1: ECC Rust API: verify $\bar{B} = \sum \bar{\alpha}_\ell \bar{A}_\ell$

6.2. Hadamard Product. We now consider the Hadamard product, optionally combined with an affine transformation. The circuit verifies that each entry of the output matrix satisfies:

$$\bar{d}_{ij} = \bar{\alpha} \cdot (\bar{a}_{ij} \cdot \bar{b}_{ij}) + \bar{\beta} \cdot \bar{c}_{ij}.$$

The pseudocode and corresponding Rust implementation are shown below.

Algorithm 6.2 `verify_hadamard`: verify that $\bar{D} = \bar{\alpha} \cdot (\bar{A} \odot \bar{B}) + \bar{\beta} \cdot \bar{C}$

```
Require: Matrices $\bar{A},\bar{B},\bar{C},\bar{D} : m \times n$, scalars $\bar{\alpha},\bar{\beta} \in \mathbb{Z}/p\mathbb{Z}$
Require: All entries are circuit variables representing least residues modulo $p$
1: function VERIFY_HADAMARD($\bar{A},\bar{B},\bar{C},\bar{D},\bar{\alpha},\bar{\beta}$)
2:     for $i \leftarrow 0$ to $m-1$ do
3:         for $j \leftarrow 0$ to $n-1$ do
4:             $prod \leftarrow \bar{a}_{ij} \cdot \bar{b}_{ij}$
5:             $lhs \leftarrow \bar{\alpha} \cdot prod + \bar{\beta} \cdot \bar{c}_{ij}$
6:             assert_is_equal($\bar{d}_{ij}, lhs$)
7:         end for
8:     end for
9: end function
```

```rust
const M: usize = 3; // Number of rows
const N: usize = 2; // Number of columns

declare_circuit!(Circuit {
    matrix_a: [[Variable; N]; M],      // A: (M x N)
    matrix_b: [[Variable; N]; M],      // B: (M x N)
    matrix_c: [[Variable; N]; M],      // C: (M x N)
    matrix_d: [[Variable; N]; M],      // D: (M x N)
    alpha: Value<C::Field>,
    beta: Value<C::Field>,
});

impl<C: Config> Define<C> for Circuit<Variable> {
    fn define<Builder: RootAPI<C>>(&self, api: &mut Builder) {
        for i in 0..M {
            for j in 0..N {
                let prod = api.mul(self.matrix_a[i][j], self.matrix_b[i][j]);
                let alpha_prod = api.mul(prod, self.alpha);
                let beta_c = api.mul(self.matrix_c[i][j], self.beta);
                let lhs = api.add(alpha_prod, beta_c);

                api.assert_is_equal(self.matrix_d[i][j], lhs);
            }
        }
    }
}
```

Listing 2: ECC Rust API: verify $\bar{D} = \bar{\alpha} \cdot (\bar{A} \odot \bar{B}) + \bar{\beta} \cdot \bar{C}$

6.3. Matrix Multiplication. In the generalized matrix multiplication scenario, the circuit verifies that the affine combination of a matrix product and an auxiliary matrix yields a specified output:

$$\bar{D} = \bar{\alpha} \cdot \bar{A}\bar{B} + \bar{\beta} \cdot \bar{C}.$$

The following pseudocode and Rust implementation show how this computation is encoded in the circuit.

Algorithm 6.3 `verify_gen_mat_mul`: verify that $\bar{D} = \bar{\alpha} \cdot \bar{A}\bar{B} + \bar{\beta} \cdot \bar{C}$

```
Require: Matrices $\bar{A} : \ell \times m$, $\bar{B} : m \times n$, $\bar{C}, \bar{D} : \ell \times n$, scalars $\bar{\alpha}, \bar{\beta} \in \mathbb{Z}/p\mathbb{Z}$
Require: All entries are circuit variables representing least residues modulo $p$
 1: function VERIFY_GEN_MAT_MUL($\bar{A}, \bar{B}, \bar{C}, \bar{D}, \bar{\alpha}, \bar{\beta}$)
 2:     for $i \leftarrow 0$ to $\ell - 1$ do
 3:         for $j \leftarrow 0$ to $n - 1$ do
 4:             $acc \leftarrow 0$
 5:             for $k \leftarrow 0$ to $m - 1$ do
 6:                 $acc \leftarrow acc + (\bar{a}_{ik} \cdot \bar{b}_{kj})$
 7:             end for
 8:             $lhs \leftarrow \bar{\alpha} \cdot acc + \bar{\beta} \cdot \bar{c}_{ij}$
 9:             assert_is_equal($\bar{d}_{ij}, lhs$)
10:         end for
11:     end for
12: end function
```

```rust
// Example dimension constants for A, B, C, and D.
const L: usize = 3; // Number of rows in A, C, and D
const M: usize = 4; // Number of columns in A, rows in B
const N: usize = 2; // Number of columns in B, C, and D

declare_circuit!(Circuit {
    matrix_a: [[Variable; M]; L],       // A: (L x M)
    matrix_b: [[Variable; N]; M],       // B: (M x N)
    matrix_c: [[Variable; N]; L],       // C: (L x N)
    matrix_d: [[Variable; N]; L],       // D = alpha AB + beta C: (L x N)
    alpha: Value<C::Field>,             // scalar alpha (mod p)
    beta: Value<C::Field>,              // scalar beta (mod p)
});

impl<C: Config> Define<C> for Circuit<Variable> {
    fn define<Builder: RootAPI<C>>(&self, api: &mut Builder) {
        for i in 0..L {
            for j in 0..N {
                let mut acc = api.constant(C::Field::zero());

                for k in 0..M {
                    let prod = api.mul(self.matrix_a[i][k], self.matrix_b[k][j]);
                    acc = api.add(acc, prod);
                }

                let alpha_acc = api.mul(acc, self.alpha);
                let beta_c = api.mul(self.matrix_c[i][j], self.beta);
                let lhs = api.add(alpha_acc, beta_c);

                api.assert_is_equal(self.matrix_d[i][j], lhs);
            }
        }
    }
}
```

Listing 3: ECC Rust API: verify $\bar{D} = \bar{\alpha} \cdot \bar{A}\bar{B} + \bar{\beta} \cdot \bar{C}$

7. Examples

Example 7.1. We illustrate the process of verifying a matrix multiplication over the finite field $\mathbb{Z}/101\mathbb{Z}$ using two contrasting cases. We assume the balanced residue range $(-50,50]$ and adopt the least residue convention in all modular representations. In part (a), all inputs satisfy the necessary range condition, and the modular constraint implies true integer equality. In part (b), the assumption fails, and we obtain a false positive verification.

(a) Let

$$A = \begin{bmatrix} 2 & -3 \\ 4 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} -1 & 5 \\ 2 & 3 \end{bmatrix}.$$

Since each entry of A and B is bounded in absolute value by 5, it follows that each entry of AB lies in $(-2\cdot 5^2, 2\cdot 5^2) \subseteq (-50,50]$.

$$\bar{A} = \begin{bmatrix} 2 & 98 \\ 4 & 1 \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} 100 & 5 \\ 2 & 3 \end{bmatrix},$$

and, if the prover is honest,

$$\bar{C} = \begin{bmatrix} 93 & 1 \\ 99 & 23 \end{bmatrix}.$$

To see this, note that

$$\bar{A}\bar{B} \equiv \begin{bmatrix} (2)(100)+(98)(2) & (2)(5)+(98)(3) \\ (4)(100)+(1)(2) & (4)(5)+(1)(3) \end{bmatrix} \equiv \begin{bmatrix} 396 & 304 \\ 402 & 23 \end{bmatrix} \equiv \begin{bmatrix} 93 & 1 \\ 99 & 23 \end{bmatrix} \bmod 101.$$

We now replace the entries of $\bar{C}$ with their balanced residue equivalents in $(-50,50]$ to obtain

$$C = \begin{bmatrix} -8 & 1 \\ -2 & 23 \end{bmatrix}.$$

Thus, we expect that $AB = C$. Indeed,

$$AB = \begin{bmatrix} (2)(-1)+(-3)(2) & (2)(5)+(-3)(3) \\ (4)(-1)+(1)(2) & (4)(5)+(1)(3) \end{bmatrix} = \begin{bmatrix} -2-6 & 10-9 \\ -4+2 & 20+3 \end{bmatrix} = \begin{bmatrix} -8 & 1 \\ -2 & 23 \end{bmatrix}.$$

(b) Now suppose the prover presents the following matrices:

$$A = \begin{bmatrix} 20 & 25 \\ 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 3 \\ 4 & 1 \end{bmatrix}.$$

The circuit receives $\bar{A} = A$, $\bar{B} = B$ and, if the prover is honest,

$$\bar{C} = \begin{bmatrix} 39 & 85 \\ 2 & 3 \end{bmatrix}$$

To see this, note that

$$\bar{A}\bar{B} \equiv \begin{bmatrix} (20)(2)+(25)(4) & (20)(3)+(25)(1) \\ (1)(2)+(0)(4) & (1)(3)+(0)(1) \end{bmatrix} \equiv \begin{bmatrix} 140 & 85 \\ 2 & 3 \end{bmatrix} \equiv \begin{bmatrix} 39 & 85 \\ 2 & 3 \end{bmatrix} \bmod 101.$$

We now replace the entries of $\bar{C}$ with their balanced residue equivalents in $(-50, 50]$ to obtain

$$C' = \begin{bmatrix} 39 & -16 \\ 2 & 3 \end{bmatrix}.$$

However, as the calculation above confirms (since $A = \bar{A}$ and $B = \bar{B}$ in this example),

$$AB = \begin{bmatrix} 140 & 85 \\ 2 & 3 \end{bmatrix} \neq C'.$$

Thus, the circuit would accept the constraint $\bar{A}\bar{B} \equiv \bar{C} \bmod 101$, leading one to incorrectly conclude that $AB = C'$. A dishonest prover could therefore claim that $AB = C'$ and pass verification, despite the claim being false.

What enables this failure is the violation of the range condition in Step (2a) of the matrix multiplication process: the entries of AB span from 2 to 140, a range of 138, which exceeds the modulus $p = 101$. This demonstrates that such assumptions cannot be omitted. Since all integers are converted to least residues *before* entering the circuit, the circuit cannot distinguish between values that differ by a multiple of p. It is therefore essential that the relevant range conditions be justified through off-circuit reasoning. ■

References

[1] Inference Labs. *Quantized Matrix Multiplication: Arithmetic Circuit Blueprint.* https://github.com/inference-labs-inc/zkml-blueprints/blob/main/matmul/quantized_matrix_multiplication.pdf. Accessed April 14, 2025.

[2] Inference Labs. *Range Check and ReLU: Arithmetic Circuit Blueprint.* https://github.com/inference-labs-inc/zkml-blueprints/blob/main/core_ops/range_check_and_relu.pdf. Accessed April 14, 2025.

[3] Inference Labs. *Range Check, Max, Min, and ReLU: Arithmetic Circuit Blueprint.* https://github.com/inference-labs-inc/zkml-blueprints/blob/main/core_ops/range_check_max_min_relu.pdf. Accessed April 14, 2025.

[4] Polyhedra Network. *ExpanderCompilerCollection: High-Level Circuit Compiler for the Expander Proof System.* https://github.com/PolyhedraZK/ExpanderCompilerCollection. Accessed April 14, 2025.

[5] Polyhedra Network. *ExpanderCompilerCollection: Rust Frontend Introduction.* https://docs.polyhedra.network/expander/rust/intro. Accessed April 14, 2025.