QUANTIZED MATRIX MULTIPLICATION VIA FREIVALDS' ALGORITHM

1. INTRODUCTION

Arithmetic circuits do not natively support floating-point operations. To accommodate real-valued data (such as neural network weights or activations), we adopt a fixed-point approximation scheme. Each real-valued input is scaled by a constant factor α and then rounded to an integer. This transformation enables us to represent real-valued matrices as integer matrices that can be embedded within a circuit over a finite field.

However, fixed-point scaling introduces a subtle issue when performing arithmetic operations. For example, suppose X and Y are real matrices, and we compute integer approximations $A = \lfloor \alpha X \rfloor$ and $B = \lfloor \alpha Y \rfloor$. The matrix product AB approximates $\alpha^2 XY$, not αXY. To restore the correct scale, we must divide the product AB by α, which is a potentially expensive operation in a constrained environment.

To address this, we design the circuit to verify that

$$AB = \alpha Q + R,$$

where Q is the downscaled output matrix (a proxy for $\lfloor \alpha(XY) \rfloor$) and R is an integer matrix of remainders. Crucially, all entries of R must lie in the interval $[0, \alpha)$, ensuring that Q represents the quotient and R the residual from dividing AB by α entrywise.

This verification reduces to two tasks: verifying that $AB = C$ for some intermediate matrix C; verifying that each entry of $C - \alpha Q$ lies in $[0, \alpha)$. This approach avoids costly in-circuit division by delegating the division to the witness (outside the circuit), while constraining the residuals to ensure correctness. For a broader discussion of fixed-point scaling and its implications for circuit design, see Subsection 3.1.

2. THE PROCESS

2.1. Steps in the process. Each step in the process has a corresponding explanatory note in Subsection 2.2 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 α be a positive integer. Let $A = [a_{ik}]$, $B = [b_{kj}]$, $C = [c_{ij}]$, and $Q = [q_{ij}]$ be integer matrices of dimensions $\ell \times m$, $m \times n$, $\ell \times n$, and $\ell \times n$, respectively.

(3) The goal of the circuit is to verify that $AB = \alpha Q + R$ for some integer matrix R whose entries all lie in $[0, \alpha)$. This verification is divided into the following two sub-goals.

(3a) Verify that $AB = C$, i.e. for all i, j,

$$\sum_{k=1}^{m} a_{ik} b_{kj} = c_{ij}. \tag{2.1}$$

(3b) Verify that each entry of $C - \alpha Q$ lies in $[0, \alpha)$, i.e. for all i, j,

$$0 \leqslant c_{ij} - \alpha q_{ij} < \alpha. \tag{2.2}$$

(4) Assume there exists an integer h such that the following bounds hold:

(4a) For all i, j,

$$h - p \leqslant \sum_{k=1}^{m} a_{ik} b_{kj}, c_{ij} < h. \tag{2.3}$$

(4b) For all i, j,

$$h - p \leqslant \alpha q_{ij} < h - (\alpha - 1). \tag{2.4}$$

(5) If the assumption of Step (4a) (respectively, (4b)) cannot be justified off circuit, it can be replaced by a weaker assumption and a range check, as described in Step (5a) (respectively, (5b)).

(5a) Assume that for all i, k, j,

$$h - p \leqslant a_{ik}, b_{kj}, c_{ij} < h. \tag{2.5}$$

Perform a range check to ensure that

$$0 \leqslant a_{ik} + T_1, b_{kj} + T_1 \leqslant T_1 + T_2, \tag{2.6}$$

where T_1, T_2 are nonnegative integers such that

$$h - p \leqslant -mT_1T_2 \quad \text{and} \quad m \cdot \max\{T_1^2, T_2^2\} \leqslant h - 1. \tag{2.7}$$

(5b) Assume that for all i, j,

$$h - p \leqslant q_{ij} < h. \tag{2.8}$$

Perform a range check to ensure that

$$0 \leqslant q_{ij} + U_1 \leqslant U_1 + U_2, \tag{2.9}$$

where U_1, U_2 are nonnegative integers such that

$$h - p \leqslant -\alpha U_1 \quad \text{and} \quad \alpha(U_2 + 1) \leqslant h. \tag{2.10}$$

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

$$\bar{a}_{ik} \equiv a_{ik} \bmod p, \quad \bar{b}_{kj} \equiv b_{kj} \bmod p, \quad \bar{c}_{ij} \equiv c_{ij} \bmod p, \quad \bar{q}_{ij} \equiv q_{ij} \bmod p, \quad 0 \leqslant \bar{a}_{ik}, \bar{b}_{kj}, \bar{c}_{ij}, \bar{q}_{ij} < p. \tag{2.11}$$

We similarly define the matrices

$$\bar{A} = [\bar{a}_{ik}], \quad \bar{B} = [\bar{b}_{kj}], \quad \bar{C} = [\bar{c}_{ij}], \quad \bar{Q} = [\bar{q}_{ij}]. \tag{2.12}$$

(7) Goals (3a) and (3b) admit the following equivalent formulations [Corollaries 3.3, 3.4].

(7a) Under assumption (4a), $AB = C$ if and only if $\bar{A}\bar{B} \equiv \bar{C} \bmod p$. That is, goal (3a) holds if and only if, for all i, j,

$$\sum_{k=1}^{m} \bar{a}_{ik}\bar{b}_{kj} \equiv \bar{c}_{ij} \bmod p. \tag{2.13}$$

(7b) Define $\bar{r}_{ij}$ to be the least residue modulo p of $\bar{c}_{ij} - \alpha\bar{q}_{ij}$:

$$\bar{r}_{ij} \equiv \bar{c}_{ij} - \alpha\bar{q}_{ij} \bmod p, \quad 0 \leqslant \bar{r}_{ij} < p. \tag{2.14}$$

Under assumptions (4a) (in particular, $c_{ij} \in [h - p, h)$) and (4b), the goal of Step (3b) holds if and only if, for all i, j,

$$0 \leqslant \bar{r}_{ij} < \alpha. \tag{2.15}$$

(8) For a deterministic verification that $AB = C$, impose the constraint (2.13) for all i, j.

(9) Alternatively, perform a probabilistic verification using Freivalds' technique. If (2.13) holds for all i, j, the circuit accepts (`True`). If not, then the circuit rejects (`False`) *with high probability*.

(9a) Let X be a vector chosen uniformly at random from $(\mathbb{Z}/p\mathbb{Z})^n$. Let $\bar{X} = (\bar{x}_1, \ldots, \bar{x}_n)$, where each $\bar{x}_j \in \{0, 1, \ldots, p-1\}$ is the least residue representative of the j-th coordinate of X.

(9b) The circuit computes the product $\bar{B}\bar{X}$ and reduces all entries modulo p to obtain $\bar{U} = (\bar{u}_1, \ldots, \bar{u}_m)$. That is,

$$\bar{u}_k \equiv \bar{b}_{k1}\bar{x}_1 + \cdots + \bar{b}_{kn}\bar{x}_n \bmod p, \quad 0 \leqslant \bar{u}_k < p \quad (1 \leqslant k \leqslant m). \tag{2.16}$$

(9c) The circuit computes the product $\bar{C}\bar{X}$ and reduces all entries modulo p to obtain $\bar{V} = (\bar{v}_1, \ldots, \bar{v}_\ell)$. That is,

$$\bar{v}_i \equiv \bar{c}_{i1}\bar{x}_1 + \cdots + \bar{c}_{in}\bar{x}_n \bmod p, \quad 0 \leqslant \bar{v}_i < p \quad (1 \leqslant i \leqslant \ell). \tag{2.17}$$

(9d) For $1 \leqslant i \leqslant \ell$, impose the constraint

$$(\bar{a}_{i1}\bar{u}_1 + \cdots + \bar{a}_{im}\bar{u}_m) - \bar{v}_i \equiv 0 \bmod p. \tag{2.18}$$

(9e) If $AB = C$, then the constraints (2.18) are satisfied for all i. If $AB \neq C$, and assuming the bounds in (2.3) hold, the probability that the constraints (2.18) are satisfied for all i is at most $1/p$:

$$\mathbb{P}((2.18) \text{ holds for all } i \mid AB \neq C) \leqslant \frac{1}{p}. \tag{2.19}$$

(9f) To reduce the soundness error, repeat Steps (9a) through (9e) independently s times using fresh random vectors $\bar{X}_1, \ldots, \bar{X}_s$. In this case, the bound in (2.19) may be replaced by $1/p^s$.

(10) Perform a range check to ensure that (2.15) holds for all i, j. **If** there exist integers $\beta \geqslant 2$ and $\kappa \geqslant 0$ such that

$$\alpha = \beta^\kappa, \tag{2.20}$$

a range check may be performed as follows.

(10a) The prover computes the κ least significant base-β digits of $\bar{r}_{ij}$, and supplies the ordered tuple $(d_{\kappa-1}, \ldots, d_0)$ to the circuit as a (public or private) witness:

$$\bar{r}_{ij} = \beta^\kappa D_\kappa + \beta^{\kappa-1} d_{\kappa-1} + \cdots + \beta^0 d_0, \quad D_\kappa \in \mathbb{Z}, \quad d_\ell \in \{0, 1, \ldots, \beta-1\}, \quad 0 \leqslant \ell < \kappa. \tag{2.21}$$

(10b) Impose constraints in the arithmetic circuit:

- For $0 \leqslant \ell < \kappa$, ensure $d_\ell \equiv e_\ell \bmod p$ with $e_\ell \in \{0, 1, \ldots, \beta-1\}$, either by enforcing

$$d_\ell(d_\ell - 1)\cdots(d_\ell - (\beta-1)) \equiv 0 \bmod p, \tag{2.22}$$

  or by enforcing membership in a lookup table (LUT) containing the valid digits $\{0, 1, \ldots, \beta-1\}$.
- Require

$$\bar{r}_{ij} \equiv \beta^{\kappa-1} d_{\kappa-1} + \cdots + \beta^0 d_0 \bmod p. \tag{2.23}$$

(10c) Assuming $0 \leqslant \bar{r}_{ij} < p$ in the first place and $\beta^\kappa \leqslant p$, these constraints guarantee

$$\bar{r}_{ij} = \beta^{\kappa-1} d_{\kappa-1} + \cdots + \beta^0 d_0, \tag{2.24}$$

provided each d_j is taken as a least residue, and hence that $\bar{r}_{ij}$ lies in the range $[0, \beta^\kappa) = [0, \alpha)$.

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 integer scaling factor α is a circuit constant.

In typical usage, the prover supplies the least residue matrices $\bar{A} = [\bar{a}_{ik}]$, $\bar{B} = [\bar{b}_{kj}]$, $\bar{C} = [\bar{c}_{ij}]$, and $\bar{Q} = [\bar{q}_{ij}]$ as part of the witness, where each entry is the canonical representative of its corresponding integer modulo p (see Definition 3.2, Step (6), and Comment (6)).

If B represents a fixed matrix—such as a set of learned model weights—it may be hardcoded into the circuit. This eliminates the need for the prover to supply $\bar{B}$ as part of the witness and can significantly reduce prover-side workload. More fundamentally, from a circuit design perspective, B is part of the structure of the computation itself, just like the scaling factor α. Hardcoding B ensures that the circuit faithfully encodes the intended computation, rather than treating B as a free input.

The matrix $\bar{C}$ must be supplied explicitly, rather than computed inside the circuit, to allow efficient probabilistic verification via Freivalds' technique (see Step (9)). Recomputing C inside the circuit would be prohibitively expensive and would defeat the purpose of using a probabilistic check.

The matrix $\bar{Q}$ must also be supplied. It is computed outside the circuit, during witness generation, by dividing each c_{ij} by α and rounding down to an integer q_{ij}. This can be performed using ordinary code, by calling unconstrained APIs, or by

using a "Hint". In any case, $\bar{Q}$ is treated as part of the witness, and the circuit does not reconstruct it internally. Instead, the circuit imposes constraints on $\bar{Q}$ indirectly by checking that each residual $c_{ij} - \alpha q_{ij}$ lies in the range $[0, \alpha)$ (see Step (10)).

In EXPANDERCOMPILERCOLLECTION [6], a *Hint* refers to a computation performed during witness generation that injects a value into the witness without incurring any constraint cost. Thus, if desired, $\bar{Q}$ could be generated automatically from $\bar{C}$ and α via a Hint, rather than being explicitly supplied by the prover. In either case, the circuit verifies only that the supplied values satisfy the required constraints, not how they were obtained.

This flexibility is particularly useful when the matrix multiplication is part of a larger computation (e.g., a neural network inference), where intermediate values are computed outside the circuit and only selectively verified within it.

(3) The circuit verifies that $AB = \alpha Q + R$ for some matrix R whose entries lie in $[0, \alpha)$. However, the prover does not need to explicitly supply R; it is implicitly defined through the supplied matrices A, B, and Q. The verification proceeds in two stages: first, checking that AB is computed correctly—either deterministically [Step (8)] or probabilistically [Step (9)]—and second, checking that all entries of $AB - \alpha Q$ lie in $[0, \alpha)$ [Step (10)]. These checks are formulated in terms of the least residue matrices $\bar{A}$, $\bar{B}$, and $\bar{Q}$ [Step (7)], using equivalences that hold under the range assumptions established in Step (4).

(4) The assumptions of Step (4) ensure that all relevant quantities involved in the matrix multiplication and scaling operations lie within a clean interval of width p, namely $[h-p, h)$, prior to reduction modulo p. This alignment guarantees that congruences modulo p behave as expected and that least residue representatives can be interpreted unambiguously.

The first bound, (2.3), ensures that both the matrix product entries $\sum_{k=1}^{m} a_{ik} b_{kj}$ and the matrix C entries c_{ij} fall within $[h-p, h)$.

The second bound, (2.4), may appear slightly peculiar: it requires that αq_{ij} lie not just in $[h-p, h)$, but in the narrower subinterval $[h-p, h-(\alpha-1))$. This additional margin ensures that when we later add a remainder term $r_{ij} \in [0, \alpha)$ (corresponding to the residual in the decomposition $c_{ij} = \alpha q_{ij} + r_{ij}$), the resulting sum $\alpha q_{ij} + r_{ij}$ still lies within $[h-p, h)$, preserving the modular consistency required for our verifications.

Note that this constraint also implies that $h - p \leqslant h - (\alpha - 1)$, i.e. $\alpha \leqslant p$. As α is also assumed to be a positive integer, we have

$$1 \leqslant \alpha \leqslant p. \tag{2.25}$$

In practice, α will be much smaller than p.

(5) The assumptions of Step (4) ideally can be justified off-circuit, based on the natural size of the quantities involved. However, if these assumptions cannot be justified externally, Step (5) provides a mechanism to replace them with explicit range checks inside the circuit.

The basic idea is to assume simple bounds on individual entries a_{ik}, b_{kj}, and c_{ij} [Step (5a)], and to perform range checks on their shifted versions to guarantee the required constraints. A similar approach applies to q_{ij} [Step (5b)], where explicit verification is more likely to be needed.

In the case of Step (4a), we may choose any T_2 satisfying $mT_2^2 \leqslant h - 1$, with:

$$T_1 = \begin{cases} 0 & \text{if } h = p, \\ T_2 & \text{if } h = (p+1)/2. \end{cases}$$

In the case of Step (4b), we may choose any U_2 satisfying $\alpha(U_2 + 1) \leqslant h - 1$, with:

$$U_1 = \begin{cases} 0 & \text{if } h = p, \\ \text{any } U_1 \text{ satisfying } \alpha U_1 \leqslant h-1 & \text{if } h = (p+1)/2. \end{cases}$$

Our standard range check process is based on shifted base-β digit decomposition. To illustrate, consider Step (4b) in the case $h = (p+1)/2$. Given a suitable base $\beta \geqslant 2$, we add a shift $\beta^{\kappa-1}(\beta-1)$ to the least residue of the value, and decompose the least residue of the result into κ base-β digits $d_{\kappa-1}, \ldots, d_0$ satisfying

$$q_{ij} + \beta^{\kappa-1}(\beta-1) \equiv \beta^{\kappa-1} d_{\kappa-1} + \cdots + \beta^0 d_0 \pmod{p},$$

where each $d_\ell \in \{0, 1, \ldots, \beta-1\}$. This ensures that $q_{ij} + \beta^{\kappa-1}(\beta-1)$ lies in $[0, \beta^\kappa)$ and, hence, that αq_{ij} itself lies within a controlled range.

In practice, the success of this approach depends on the magnitude of α relative to p, and the choice of base β and number of digits κ must be adapted accordingly to fit the size of the values encountered. Specifically, we require that there exists an integer n for which

$$2(\beta-1)\beta^{n-2} < p < \beta^n,$$

and that $\kappa \leqslant n-1$. For a detailed discussion of these conditions and the full range check process, see [3].

We also perform a range check in Step (10).

(6) As discussed in Comment (2), the prover supplies the least residue matrices $\bar{A} = [\bar{a}_{ik}]$, $\bar{C} = [\bar{c}_{ij}]$, $\bar{Q} = [\bar{q}_{ij}]$, and possibly $\bar{B} = [\bar{b}_{kj}]$, as part of the witness, where each entry is the canonical representative of its corresponding integer modulo p.

As noted, the circuit's goal (3) admits an equivalent formulation (7) in terms of congruences modulo p. This equivalence follows from Proposition 3.1, but it is essential that the assumptions of Step (4) hold. For details, see Proposition 3.1, Corollary 3.3, and Corollary 3.4.

(7) (7a) See Corollary 3.3.

(7b) See Corollary 3.4. Also see Listing 1 for code that computes $\bar{q}_{ij}$ and $\bar{r}_{ij}$ (given α and $\bar{c}_{ij}$) in an unconstrained environment with EXPANDERCOMPILERCOLLECTION [6].

(8) See Algorithm 4.1 and Listing 2.

(9) See Algorithm 4.2 and Listing 3. Refer to [2] for further details on Freivalds' technique. Briefly, the verification fails unless the random vector $\bar{X}$ lies in the null space of $\bar{A}\bar{B} - \bar{C}$, which, if $\bar{A}\bar{B} \neq \bar{C}$, is a subspace of $(\mathbb{Z}/p\mathbb{Z})^n$ of dimension at most $n-1$. It follows that the probability of passing verification of $\bar{A}\bar{B} \neq \bar{C}$ is at most $1/p$.

(10) See Algorithms 4.3, 4.4, and Listings 4, 5. Refer to [4] for further details.

3. THE MATHS

Proposition 3.1. *Let p be a positive integer (not necessarily a prime), and let integers h, x, y satisfy*

$$h - p \leqslant x, y < h. \tag{3.1}$$

Then $x = y$ if and only if

$$x \equiv y \bmod p. \tag{3.2}$$

Proof. If $x = y$ then $x \equiv y \bmod p$. For the converse, note that the congruence (3.2) holds if and only if there is an integer t such that

$$x - y = tp. \tag{3.3}$$

Since h, x, y are integers, the assumption (3.1) 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 (3.3) holds, then

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

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

Definition 3.2. Let $M = [m_{ij}]$ and $M' = [m'_{ij}]$ be integer matrices of the same dimensions. Given a positive integer p, we write

$$M \equiv M' \bmod p$$

to mean that $m_{ij} \equiv m'_{ij} \bmod p$ for all i, j. We denote by $\bar{M}$ the matrix of least residues modulo p, that is, $\bar{M} = [\bar{m}_{ij}]$, where each $\bar{m}_{ij}$ is the unique integer in $\{0, 1, \ldots, p-1\}$ congruent to $m_{ij} \bmod p$. ■

Corollary 3.3. *Let p be a positive integer (not necessarily a prime). Let $A = [a_{ik}]$, $B = [b_{kj}]$, and $C = [c_{ij}]$ be integer matrices of dimensions $\ell \times m$, $m \times n$, and $\ell \times n$, respectively. Assume there exists an integer h such that, for all i, j,*

$$h - p \leqslant \sum_{k=1}^{m} a_{ik} b_{kj}, c_{ij} < h.$$

Then

$$AB = C \quad \Longleftrightarrow \quad \bar{A}\bar{B} \equiv \bar{C} \bmod p.$$

Proof. For all i, j, both

$$x = \sum_{k=1}^{m} a_{ik} b_{kj} \quad \text{and} \quad y = c_{ij}$$

lie in the same interval of length p:

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

Therefore, by Proposition 3.1, we have $x = y$ if and only if

$$x \equiv y \bmod p,$$

which is equivalent to

$$\sum_{k=1}^{m} \bar{a}_{ik} \bar{b}_{kj} \equiv \bar{c}_{ij} \bmod p.$$

The result follows, as the (i, j) entry of AB is $\sum_{k=1}^{m} a_{ik} b_{kj}$, while that of $\bar{A}\bar{B}$ is $\sum_{k=1}^{m} \bar{a}_{ik} \bar{b}_{kj}$. □

Corollary 3.4. *Let p be a positive integer (not necessarily a prime). Let $\alpha \geqslant 1$, c, and q be integers satisfying*

$$h - p \leqslant c < h \quad \textit{and} \quad h - p \leqslant \alpha q < h - (\alpha - 1).$$

Let r be the least residue modulo p of $c - \alpha q$, i.e.

$$r \equiv c - \alpha q \bmod p \quad \textit{and} \quad 0 \leqslant r < p.$$

We have

$$0 \leqslant c - \alpha q < \alpha \quad \iff \quad 0 \leqslant r < \alpha.$$

Proof. First, note that the assumptions

$$h - p \leqslant \alpha q < h - (\alpha - 1)$$

imply that $\alpha \leqslant p$. Indeed, since $h - p < h - (\alpha - 1)$, we have $\alpha - 1 < p$, and as α and p are integers, this is equivalent to $\alpha \leqslant p$.

Now, suppose first that $0 \leqslant c - \alpha q < \alpha$. Then $c - \alpha q$ lies in $[0, \alpha)$, which is contained in $[0, p)$. Since r is defined as the least residue of $c - \alpha q$ modulo p, and $c - \alpha q \in [0, p)$, it follows that $r = c - \alpha q$. In particular, $r \in [0, \alpha)$.

Conversely, suppose that $r \in [0, \alpha)$. Then $0 \leqslant r \leqslant \alpha - 1$, and so

$$h - p \leqslant \alpha q \leqslant \alpha q + r \quad \text{and} \quad \alpha q + r \leqslant \alpha q + (\alpha - 1) < h.$$

Thus, $\alpha q + r$ lies in $[h - p, h)$. Since c also lies in $[h - p, h)$ and $\alpha q + r \equiv c \bmod p$ (because $r \equiv c - \alpha q \bmod p$ by definition), Proposition 3.1 implies that $\alpha q + r = c$. Rearranging gives $c - \alpha q = r$, and hence $c - \alpha q \in [0, \alpha)$. □

3.1. Fixed-point multiplication. Suppose we have real numbers x, y, and z, and we fix a positive integer scaling factor α. In fixed-point arithmetic we represent these numbers by integers that approximate the scaled real values. For example, we choose integers

$$a, b, c \in \mathbb{Z}$$

that approximate αx, αy, and αz, respectively; that is,

$$a \approx \alpha x, \quad b \approx \alpha y, \quad c \approx \alpha z.$$

Then a, b, and c form the fixed-point representations of x, y, and z (with scaling α). One way to quantify the error is to introduce discrepancy terms:

$$\delta_x = \alpha x - a, \quad \delta_y = \alpha y - b, \quad \delta_z = \alpha z - c.$$

In many quantization schemes these errors are bounded by a constant. For example, if we use the floor function we have

$$0 \leqslant \delta_x, \delta_y, \delta_z < 1.$$

A key observation is that if

$$a \approx \alpha x \quad \text{and} \quad b \approx \alpha y,$$

then their product satisfies

$$ab \approx (\alpha x)(\alpha y) = \alpha^2 xy.$$

Likewise, since

$$c \approx \alpha z,$$

we have

$$\alpha c \approx \alpha^2 z.$$

Thus, if the fixed-point multiplication is consistent, then

$$\frac{ab}{\alpha} \approx c,$$

which in turn implies

$$xy \approx z.$$

This is the basic idea behind fixed-point multiplication: one computes the product ab (which is at scale α^2) and then "rescales" it by dividing by α to obtain an approximation for the fixed-point representation of xy.

Below are three standard examples of how one might define the fixed-point representations:

(1) *Using the floor function:* Define

$$a = \lfloor \alpha x \rfloor, \quad b = \lfloor \alpha y \rfloor, \quad c = \lfloor \alpha z \rfloor.$$

Then the discrepancies are given by

$$\delta_x = \alpha x - \lfloor \alpha x \rfloor, \quad \delta_y = \alpha y - \lfloor \alpha y \rfloor, \quad \delta_z = \alpha z - \lfloor \alpha z \rfloor,$$

with

$$0 \leqslant \delta_x, \delta_y, \delta_z < 1.$$

(2) *Using the ceiling function:* Define

$$a = \lceil \alpha x \rceil, \quad b = \lceil \alpha y \rceil, \quad c = \lceil \alpha z \rceil.$$

In this case we may write

$$a = \alpha x + \delta'_x, \quad b = \alpha y + \delta'_y, \quad c = \alpha z + \delta'_z,$$

where

$$-1 < \delta'_x, \delta'_y, \delta'_z \leqslant 0.$$

(3) *Using rounding to the nearest integer:* Define the fixed-point representations of x, y, z as follows:

$$a = \mathrm{round}(\alpha x), \quad b = \mathrm{round}(\alpha y), \quad c = \mathrm{round}(\alpha z).$$

Then, setting

$$\delta''_x = \alpha x - a, \quad \delta''_y = \alpha y - b, \quad \delta''_z = \alpha z - c,$$

we have

$$-\frac{1}{2} \leqslant \delta''_x, \delta''_y, \delta''_z \leqslant \frac{1}{2}.$$

Proposition 3.5. *Let α be a positive integer, and let $X = [x_{ik}]$ and $Y = [y_{kj}]$ be real matrices of dimensions $\ell \times m$ and $m \times n$, respectively. Define*

$$Z = [z_{ij}] = XY.$$

Let $A = [a_{ik}]$, $B = [b_{kj}]$, and $C = [c_{ij}]$ be integer matrices of the same dimensions as X, Y, and Z, respectively. Assume that there exist real matrices $\Delta_X = [\delta_{x,ik}]$, $\Delta_Y = [\delta_{y,kj}]$, and $\Delta_Z = [\delta_{z,ij}]$, whose entries all lie in $[-1, 1]$, such that

$$A = \alpha X - \Delta_X, \quad B = \alpha Y - \Delta_Y, \quad C = \alpha Z - \Delta_Z.$$

Consider the product $AB = [(AB)_{ij}]$. For each i, j, apply the division algorithm with divisor α to express

$$(AB)_{ij} = \alpha q_{ij} + r_{ij}, \quad 0 \leqslant r_{ij} < \alpha.$$

Thus, the matrix product can be rewritten as

$$AB = \alpha Q + R,$$

where $Q = [q_{ij}]$ and $R = [r_{ij}]$ are the quotient and remainder matrices, respectively. Then,

$$C = Q + E,$$

where $E = [\varepsilon_{ij}]$ is an error matrix such that, for all i, j,

$$|\varepsilon_{ij}| \leqslant 2 + \frac{m-1}{\alpha} + \sum_{k=1}^{m} \left(|x_{ik}| + |y_{kj}| \right).$$

Remark 3.6. In a typical scenario, we have[1] $m, x_{ik}, y_{kj}, z_{ij} \asymp 1$, and α is large. Thus,

$$\varepsilon_{ij} \ll \frac{m}{\alpha} + m \ll 1, \quad c_{ij} = \alpha z_{ij} - \delta_{z,ij} \gg \alpha, \quad \text{and} \quad q_{ij} = c_{ij} - \varepsilon_{ij} = c_{ij}\left(1 + O\left(\frac{1}{\alpha}\right)\right).$$

Furthermore,

$$q_{ij} = c_{ij} - \varepsilon_{ij} = \alpha z_{ij} - \delta_{z,ij} - \varepsilon_{ij} = \alpha z_{ij} + O(1) = \alpha z_{ij}\left(1 + O\left(\frac{1}{\alpha}\right)\right).$$

This implies that

$$z_{ij} = \frac{q_{ij}}{\alpha}\left(1 + O\left(\frac{1}{\alpha}\right)\right).$$

In some cases, cancellation effects may be such that the sum

$$z_{ij} = \sum_{k=1}^{m} x_{ik} y_{kj}$$

is significantly smaller than 1—in extreme cases, it could be as small as $O(1/\alpha)$. (For instance, if row i of X is orthogonal to column j of Y, then $z_{ij} = 0$.) However, such occurrences are expected to be rare. In practical applications such as quantization in neural networks, the network is typically robust enough to maintain performance even when some entries of the product matrix are unusually small. ■

Proof of Proposition 3.5. For each fixed pair of indices $i \in \{1, \ldots, \ell\}$ and $j \in \{1, \ldots, n\}$, the (i, j)-entry of the true matrix product is given by

$$z_{ij} = \sum_{k=1}^{m} x_{ik} y_{kj}.$$

By the definition of C, we have

$$c_{ij} = \alpha z_{ij} - \delta_{z,ij} \quad \Longrightarrow \quad \alpha^2 z_{ij} = \alpha(c_{ij} + \delta_{z,ij}), \tag{3.4}$$

where $\delta_{z,ij} \in [-1, 1]$. Meanwhile, in fixed-point arithmetic, we compute

$$(AB)_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}.$$

Since

$$a_{ik} = \alpha x_{ik} - \delta_{x,ik} \quad \text{and} \quad b_{kj} = \alpha y_{kj} - \delta_{y,kj},$$

it follows that

$$a_{ik} b_{kj} = (\alpha x_{ik} - \delta_{x,ik})(\alpha y_{kj} - \delta_{y,kj}) = \alpha^2 x_{ik} y_{kj} - \alpha\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right) + \delta_{x,ik}\delta_{y,kj}.$$

Summing over all k, we obtain

$$\begin{aligned}
(AB)_{ij} &= \sum_{k=1}^{m} a_{ik} b_{kj} \\
&= \sum_{k=1}^{m} \left\{\alpha^2 x_{ik} y_{kj} - \alpha\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right) + \delta_{x,ik}\delta_{y,kj}\right\} \\
&= \alpha^2 \sum_{k=1}^{m} x_{ik} y_{kj} - \alpha \sum_{k=1}^{m}\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right) + \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj} \\
&= \alpha^2 z_{ij} - \alpha \sum_{k=1}^{m}\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right) + \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj} \\
&\overset{(3.4)}{=} \alpha\left(c_{ij} + \delta_{z,ij}\right) - \alpha \sum_{k=1}^{m}\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right) + \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj} \\
&= \alpha c_{ij} + \alpha\left\{\delta_{z,ij} - \sum_{k=1}^{m}\left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik}\right)\right\} + \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj}.
\end{aligned} \tag{3.5}$$

[1] The notation $f \asymp g$ denotes that $f \ll g$ and $g \ll f$, i.e., there exist absolute positive constants c_1 and c_2 such that $c_1|f| < |g| < c_2|f|$.

On the other hand, we have integers q_{ij} and r_{ij}, $0 \leqslant r_{ij} < \alpha$, such that

$$(AB)_{ij} = \alpha q_{ij} + r_{ij}.$$

Substituting this into (3.5), we obtain

$$\alpha q_{ij} + r_{ij} = \alpha c_{ij} + \alpha \left\{ \delta_{z,ij} - \sum_{k=1}^{m} \left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik} \right) \right\} + \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj}.$$

Rearranging and dividing by α gives

$$c_{ij} = q_{ij} + \frac{r_{ij}}{\alpha} - \delta_{z,ij} + \sum_{k=1}^{m} \left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik} \right) - \frac{1}{\alpha} \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj}.$$

Since $\delta_{x,ik}, \delta_{y,kj}, \delta_{z,ij} \in [-1,1]$ and $0 \leqslant r_{ij} < \alpha$, we bound the error terms:

$$\left| \frac{r_{ij}}{\alpha} - \delta_{z,ij} - \frac{1}{\alpha} \sum_{k=1}^{m} \delta_{x,ik}\delta_{y,kj} \right| \leqslant \frac{\alpha - 1}{\alpha} + 1 + \frac{m}{\alpha} = 2 + \frac{m-1}{\alpha}$$

and

$$\left| \sum_{k=1}^{m} \left(x_{ik}\delta_{y,kj} + y_{kj}\delta_{x,ik} \right) \right| \leqslant \sum_{k=1}^{m} \left(|x_{ik}| + |y_{kj}| \right).$$

Combining these, we conclude that

$$c_{ij} = q_{ij} + \varepsilon_{ij},$$

where

$$|\varepsilon_{ij}| \leqslant 2 + \frac{m-1}{\alpha} + \sum_{k=1}^{m} \left(|x_{ik}| + |y_{kj}| \right).$$

□

Example 3.7. Let

$$X = \begin{bmatrix} 1.11 & -3.23 \\ 4.0 & 2.56 \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} -2.12 & 0.9 \\ 3.324 & -1.15 \end{bmatrix}.$$

We simulate the computation of the product $Z = XY$ in fixed-point arithmetic using a scaling factor of $\alpha = 10$.

Each entry of X and Y is converted to fixed-point form by scaling it by $\alpha = 10$ and taking the integer part (floor). This yields the fixed-point matrices A and B:

$$A = \lfloor \alpha X \rfloor = \begin{bmatrix} 11 & -33 \\ 40 & 25 \end{bmatrix} \quad \text{and} \quad B = \lfloor \alpha Y \rfloor = \begin{bmatrix} -22 & 9 \\ 33 & -12 \end{bmatrix}.$$

Our goal is to approximate

$$C = \lfloor \alpha Z \rfloor, \quad \text{where} \quad Z = XY.$$

However, computing Z exactly and then applying the fixed-point transformation would defeat the purpose of fixed-point arithmetic. Instead, we must work exclusively with A and B.

The product of the fixed-point matrices is computed as

$$AB = \begin{bmatrix} 11 & -32 \\ 40 & 25 \end{bmatrix} \begin{bmatrix} -22 & 9 \\ 33 & -11 \end{bmatrix} = \begin{bmatrix} -1331 & 495 \\ -55 & 60 \end{bmatrix}.$$

We now apply the division algorithm with divisor $\alpha = 10$ to each entry of AB, yielding a matrix of quotients Q and a matrix of remainders R:

$$AB = \alpha Q + R = 10 \begin{bmatrix} -134 & 49 \\ -6 & 6 \end{bmatrix} + \begin{bmatrix} 9 & 5 \\ 5 & 0 \end{bmatrix}.$$

Thus,

$$\alpha^{-1} AB = Q + \alpha^{-1} R = \begin{bmatrix} -134 & 49 \\ -6 & 6 \end{bmatrix} + \alpha^{-1} R,$$

where the entries of $\alpha^{-1}R$ all lie in $[0,1)$. We approximate the fixed-point representation C of Z as

$$C \approx Q = \begin{bmatrix} -134 & 49 \\ -6 & 6 \end{bmatrix}.$$

For comparison, we compute the exact product $Z = XY$ using standard arithmetic:

$$Z = \begin{bmatrix} -13.08972 & 4.7135 \\ 0.02944 & 0.656 \end{bmatrix}.$$

Applying the fixed-point transformation,

$$C = \begin{bmatrix} \lfloor -130.8972 \rfloor & \lfloor 47.135 \rfloor \\ \lfloor 0.2944 \rfloor & \lfloor 6.56 \rfloor \end{bmatrix} = \begin{bmatrix} -131 & 47 \\ 0 & 7 \end{bmatrix}.$$

The error in approximating C by Q is

$$E = C - Q = \begin{bmatrix} 3 & -2 \\ 6 & 1 \end{bmatrix}.$$

Finally, the error in approximating Z by $\alpha^{-1}Q$ is

$$Z - \alpha^{-1}Q = \begin{bmatrix} 0.31028 & -0.1865 \\ 0.62944 & 0.056 \end{bmatrix}.$$

■

4. The Code

The pseudocode and Rust code that follow are designed for implementation within the ExpanderCompilerCollection (ECC) framework [5, 6]. This library provides a specialized interface for constructing and verifying arithmetic circuits.

We begin with a simple unconstrained division function that can be used to compute the quotient and remainder when dividing an integer by α. This function is used during witness generation and does not impose any constraints within the circuit.

```rust
fn div_unconstrained<C: Config, Builder: RootAPI<C>>(
    api: &mut Builder,
    c: Variable,
    alpha: u32,
) -> (Variable, Variable) {
    // Since c is nonnegative (a least residue mod p), we can directly compute
    // the quotient and remainder using the unconstrained division and modulo APIs.
    let q = api.unconstrained_int_div(c, alpha);
    let r = api.unconstrained_mod(c, alpha);
    (q, r)
}
```

Listing 1: Simplified unconstrained division algorithm using the ECC Rust API

Next, we verify deterministic matrix multiplication by checking that each entry of $\bar{C}$ matches the corresponding entry of $\bar{A}\bar{B}$. This is implemented both as pseudocode and using ECC's circuit API.

Algorithm 4.1 `verify_mat_mul`: verify that $\bar{A}\bar{B} = \bar{C}$

Require: Matrices $\bar{A} : \ell \times m$, $\bar{B} : m \times n$, $\bar{C} : \ell \times n$
Require: All entries are circuit variables representing least residues modulo p

function VERIFY_MAT_MUL($\bar{A}, \bar{B}, \bar{C}$)
  for $i \leftarrow 0$ to $\ell - 1$ **do**
    for $j \leftarrow 0$ to $n - 1$ **do**
      $acc \leftarrow 0$
      for $k \leftarrow 0$ to $m - 1$ **do**
        $acc \leftarrow acc + (\bar{a}_{ik} \cdot \bar{b}_{kj})$
      end for
      `assert_is_equal`($\bar{c}_{ij}, acc$)
    end for
  end for
end function

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

declare_circuit!(MatMulCircuit {
    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), claimed output
});

impl<C: Config> Define<C> for MatMulCircuit<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);
                }

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

Listing 2: ECC Rust API: verify $\bar{A}\bar{B} = \bar{C}$

For a more efficient probabilistic check, we use Freivalds' algorithm. The idea is to compress the matrix identity $AB = C$ using a random vector x, and check that $A(Bx) = Cx$. This verification has one-sided soundness error at most $1/p$, and can be repeated to reduce the soundness error.

Algorithm 4.2 `freivalds_check`: One-sided Monte Carlo verification of $AB = C$. Fails with high probability if $AB \neq C$.

Require: Matrices $A \in \mathbb{F}^{l \times m}$, $B \in \mathbb{F}^{m \times n}$, $C \in \mathbb{F}^{l \times n}$
Ensure: Reject with high probability if $AB \neq C$

Sample random vector $x \in \mathbb{F}^n$
Initialize $Bx \in \mathbb{F}^m$ as zero vector
for $i = 0$ to $m-1$ **do**
    for $j = 0$ to $n-1$ **do**
        $Bx[i] \leftarrow Bx[i] + B[i][j] \cdot x[j]$
    end for
end for
Initialize $ABx \in \mathbb{F}^l$ as zero vector
for $i = 0$ to $l-1$ **do**
    for $j = 0$ to $m-1$ **do**
        $ABx[i] \leftarrow ABx[i] + A[i][j] \cdot Bx[j]$
    end for
end for
Initialize $Cx \in \mathbb{F}^l$ as zero vector
for $i = 0$ to $l-1$ **do**
    for $j = 0$ to $n-1$ **do**
        $Cx[i] \leftarrow Cx[i] + C[i][j] \cdot x[j]$
    end for
end for
for $i = 0$ to $l-1$ **do**
    Assert $ABx[i] = Cx[i]$
end for

Below is the ECC implementation of Freivalds' technique. The circuit samples a random vector x, computes $u = Bx$, $ABx = A(u)$, and Cx, and asserts that $ABx = Cx$ componentwise.

We now describe the base-β range check technique used to enforce boundedness of quantities such as $\bar{r}_{ij} \in [0, \alpha)$. The next algorithm computes the κ least significant base-β digits of a given integer.

```rust
use expander_compiler::frontend::*;

const N_ROWS_A: usize = 4; // l
const N_COLS_A: usize = 4; // m
const N_ROWS_B: usize = 4; // m
const N_COLS_B: usize = 4; // n

declare_circuit!(MatMulCircuit {
    matrix_a: [[Variable; N_COLS_A]; N_ROWS_A],         // (l x m)
    matrix_b: [[Variable; N_COLS_B]; N_ROWS_B],         // (m x n)
    matrix_product_ab: [[Variable; N_COLS_B]; N_ROWS_A] // (l x n), provided by prover
});

impl<C: Config> Define<C> for MatMulCircuit<Variable> {
    fn define<Builder: RootAPI<C>>(&self, api: &mut Builder) {
        let a: Vec<Vec<Variable>> = self.matrix_a.iter().map(|row| row.to_vec()).collect();
        let b: Vec<Vec<Variable>> = self.matrix_b.iter().map(|row| row.to_vec()).collect();
        let c: Vec<Vec<Variable>> = self.matrix_product_ab.iter().map(|row| row.to_vec()).collect
    ();

        // 1. Sample random vector x in F^n
        let x: Vec<Variable> = (0..N_COLS_B).map(|_| api.get_random_value()).collect();
        // let mut x = vec![];
        // for _ in 0..N_COLS_B {
        //     x.push(api.get_random_value());
        // }
        // println!("Random vector x: {:?}", x);

        // 2. Compute u = Bx, which lies in F^m
        let mut bx = vec![api.constant(C::CircuitField::zero()); N_ROWS_B];
        for i in 0..N_ROWS_B {
            for j in 0..N_COLS_B {
                let prod = api.mul(b[i][j], x[j]);
                bx[i] = api.add(bx[i], prod);
            }
        }

        // 3. Compute A(Bx), which lies in F^l
        let mut abx = vec![api.constant(C::CircuitField::zero()); N_ROWS_A];
        for i in 0..N_ROWS_A {
            for j in 0..N_COLS_A {
                let prod = api.mul(a[i][j], bx[j]);
                abx[i] = api.add(abx[i], prod);
            }
        }

        // 4. Compute v = Cx, which lies in F^l
        let mut cx = vec![api.constant(C::CircuitField::zero()); N_ROWS_A];
        for i in 0..N_ROWS_A {
            for j in 0..N_COLS_B {
                let prod = api.mul(c[i][j], x[j]);
                cx[i] = api.add(cx[i], prod);
            }
        }

        // 5. Freivalds check: abx == cx
        for i in 0..N_ROWS_A {
            api.assert_is_equal(abx[i], cx[i]);
        }
    }
}
```

Listing 3: ECC Rust API: One-sided Monte Carlo verification of $AB = C$. Fails with high probability if $AB \neq C$

Algorithm 4.3 `to_base_beta`: compute the κ least significant digits of base-β representation of a nonnegative integer

Require: nonnegative integers q_0 and κ
Ensure: a list d representing the κ least significant base-β digits of q_0

$d \leftarrow []$ ▷ Initialize an empty list
for $i \leftarrow 0$ to $\kappa - 1$ **do**
    append $q_0 \bmod \beta$ **to** d
    $q_0 \leftarrow \lfloor q_0 / \beta \rfloor$ ▷ Shift q_0 to the right by one digit
end for
return d

The following Rust function performs the same task: it extracts the κ least significant base-β digits of a variable using ECC's unconstrained APIs.

```rust
fn to_base_beta<C: Config, Builder: RootAPI<C>>(
    api: &mut Builder,
    q_0: Variable,
    beta: u32,
    kappa: u32,
) -> Vec<Variable> {
    let mut d = Vec::with_capacity(kappa as usize); // Preallocate vector
    let mut q = q_0; // Copy q_0 to modify iteratively

    for _ in 0..kappa {
        d.push(api.unconstrained_mod(q, beta)); // Extract least significant base-beta digit
        q = api.unconstrained_int_div(q, beta); // Shift q to remove the extracted digit
    }

    d
}
```

Listing 4: ECC Rust API: compute the κ least significant base-β digits of a nonnegative integer

To verify that a value is indeed represented by a valid base-β decomposition, we reconstruct the original integer from its digits and enforce range constraints using a lookup table. The next algorithm shows the reconstruction logic and final consistency check.

Algorithm 4.4 `from_base_beta`: reconstruct and verify a nonnegative integer from at most κ least significant base-β digits

Require: list of base-β digits `d`, nonnegative integer κ, and original value $\bar{a}$
Ensure: `reconstructed_integer`: the integer represented by the first κ base-β digits of `d`

`reconstructed_integer` $\leftarrow 0$
`LOOKUP_TABLE` $\leftarrow \{0, 1, \ldots, \beta - 1\}$ ▷ Predefined valid digit set
for $j \leftarrow 0$ to $\max\{\kappa - 1, \text{len}(d) - 1\}$ **do**
    `digit` $\leftarrow d[j]$
    Enforce `digit` $\in$ `LOOKUP_TABLE` as a circuit constraint
    `digit_by_beta_to_the_j` $\leftarrow$ `digit` $\times \beta^j$
    `reconstructed_integer` $\leftarrow$ `reconstructed_integer` + `digit_by_beta_to_the_j`
end for
assert `reconstructed_integer` $= \bar{a}$ ▷ Final constraint: ensure correctness of base-β representation
return `reconstructed_integer`

The following Rust implementation reconstructs the original variable from its base-β digits and enforces validity using a lookup table and a final equality constraint. This is a standard technique for implementing range checks in arithmetic circuits.

```rust
fn lookup_digit<C: Config, API: RootAPI<C>>(
    api: &mut API,
    digit: Variable,
    lookup_table: &mut LogUpSingleKeyTable
) {
    // Use the lookup table (populated with valid digit constants) to constrain `digit`.
    // The second argument here is the associated value vector, which in this case we assume to be empty.
    lookup_table.query(digit, vec![]);
}

// Check that every digit in the slice `d` is a valid base-beta digit using the lookup table.
fn base_beta_digit_check<C: Config, API: RootAPI<C>>(
    api: &mut API,
    d: &[Variable],
    beta: u32,
    lookup_table: &mut LogUpSingleKeyTable
) {
    for &digit in d.iter() {
        lookup_digit(api, digit, lookup_table);
    }
}

// Reconstruct an integer from the first `kappa` digits in `d` (assumed little-endian)
// and enforce that each digit is a valid base-beta digit via the lookup table.
// Finally, assert equality with a given input `a_bar`.
fn from_base_beta<C: Config, API: RootAPI<C>>(
    api: &mut API,
    d: &[Variable],
    kappa: usize,
    beta: u32,
    lookup_table: &mut LogUpSingleKeyTable,
    a_bar: Variable
) -> Variable {
    base_beta_digit_check(api, d, beta, lookup_table);

    let mut reconstructed_integer = api.constant(0);
    for (j, &digit) in d.iter().take(kappa).enumerate() {
        let factor = api.constant(beta.pow(j as u32));
        let term = api.mul(factor, digit);
        reconstructed_integer = api.add(reconstructed_integer, term);
    }

    api.assert_is_equal(reconstructed_integer, a_bar);

    reconstructed_integer
}
```

Listing 5: ECC Rust API: reconstruct a nonnegative integer from at most κ least significant base-β digits and impose constraints

5. Examples

Example 5.1. Let $p = 521$, which is prime, and $\alpha = 2^3$. Consider

$$A = \begin{bmatrix} 2 & -3 \\ -1 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} -1 & 2 \\ 3 & -2 \end{bmatrix}, \quad C = \begin{bmatrix} -11 & 10 \\ 13 & -10 \end{bmatrix}, \quad Q = \begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix}.$$

Note that if $h = (p+1)/2 = 261$, then $h - p = -260$. Each entry A, B, C lies in $[h-p, h) = [-260, 261)$. Also, each entry x of A and B satisfies

$$0 \leqslant x + T_1 \leqslant T_1 + T_2$$

with $T_1 = T_2 = 4$. A range check can verify this. (See [3] for worked example of a range check. We could have taken $T_1 = T_2 = 3$ here, but our simplest range check works only for intervals of the form 2^κ.) We could choose any T_1, T_2 such that (2.7) holds in this case: $-260 \leqslant -2T_1T_2$ and $2 \cdot \max\{T_1^2, T_2^2\} \leqslant 260$. It follows that each entry of AB lies in $[h-p, h) = [-260, 261)$.

Similarly, a range check may be performed to ensure that each entry q of Q satisfies

$$0 \leqslant q + U_1 \leqslant U_1 + U_2,$$

with $U_1 = U_2 = 2$. We could choose any U_1, U_2 such that (2.10) holds: $-260 \leqslant -\alpha U_1$ and $\alpha(U_2 + 1) \leqslant 261$. It follows that each entry of Q satisfies $h - p \leqslant \alpha q \leqslant h - \alpha$.

Taking least residues modulo $p = 521$ of each entry yields

$$\bar{A} = \begin{bmatrix} 2 & 518 \\ 520 & 4 \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} 520 & 2 \\ 3 & 519 \end{bmatrix} \quad \bar{C} = \begin{bmatrix} 510 & 10 \\ 13 & 511 \end{bmatrix}, \quad \bar{Q} = \begin{bmatrix} 519 & 1 \\ 1 & 519 \end{bmatrix}.$$

We verify that

$$\bar{A}\bar{B} \equiv \bar{C} \bmod 521$$

either directly or by using Freivalds' technique (see [1, 2] for details).

We compute

$$\bar{C} - \alpha\bar{Q} \equiv \begin{bmatrix} 5 & 2 \\ 5 & 6 \end{bmatrix} \bmod 521,$$

and perform a range check on each entry of the remainder matrix

$$R = \begin{bmatrix} 5 & 2 \\ 5 & 6 \end{bmatrix}$$

to confirm that it lies in $[0, \alpha) = [0, 8)$ (see [3] for details).

We can deduce from this, and the bounds for the entries of A, B, C, Q, that

$$AB = \alpha Q + R.$$

■

References

[1] Inference Labs. *Matrix Addition, Hadamard Products, and Matrix Multiplication: Arithmetic Circuit Blueprint*. `https://github.com/inference-labs-inc/zkml-blueprints/blob/main/matmul/matrix_addition_hadamard_product_matrix_multiplication.pdf`. Accessed May 16, 2025.

[2] Inference Labs. *Matrix Multiplication via Freivalds' Algorithm: Arithmetic Circuit Blueprint*. `https://github.com/inference-labs-inc/zkml-blueprints/blob/main/matmul/matrix_multiplication_freivalds_algorithm.pdf`. Accessed May 16, 2025.

[3] 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 May 16, 2025.

[4] 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 May 16, 2025.

[5] Polyhedra Network. *ExpanderCompilerCollection: High-Level Circuit Compiler for the Expander Proof System*. `https://github.com/PolyhedraZK/ExpanderCompilerCollection`. Accessed May 16, 2025.

[6] Polyhedra Network. *ExpanderCompilerCollection: Rust Frontend Introduction*. `https://docs.polyhedra.network/expander/rust/intro`. Accessed May 16, 2025.