Matrix Multiplication I

Naive Method

1 The Process 1
2 Parameter Roles: Public, Private, and Computed 2
3 The Maths 2
4 The Code 3
References 4

1. The Process

(1) The circuit operates over the finite field $\mathbb{Z}/p\mathbb{Z}$, where p is an n-bit prime (typically, $n \approx 256$ in practice):

$$2^{n-1} < p < 2^n.$$

(2) Let $A = [a_{ik}]$ and $B = [b_{kj}]$ be integer matrices of dimensions $\ell \times m$ and $m \times n$, respectively. Let U be an integer such that, for all i, k, j,

$$-U \leqslant a_{ik}, b_{kj} \leqslant U.$$

(3) Assume[1] that

$$mU^2 < \frac{p}{2}.$$

In practice, typical values of m include 28, 256, and 1568, while typical values of U include 2^{16}, 2^{21}, and 2^{32}.

(4) Let $C = [c_{ij}]$ be an $\ell \times n$ integer matrix and assume[2] that, for all i, j,

$$-\frac{p}{2} < c_{ij} \leqslant \frac{p}{2}.$$

(5) Convert each a_{ik}, b_{kj}, and c_{ij} to its least-residue modulo p, denoted a'_{ik}, b'_{kj}, and c'_{ij}, respectively. This conversion is necessary because frameworks typically do not support signed integers even in unconstrained environments.

(6) For all i, j, impose the constraint [Algorithm 4.1, Listing 1]

$$c'_{ij} \equiv \sum_{k=1}^{m} a'_{ik} b'_{kj} \bmod p. \tag{1.1}$$

This is of course equivalent to the same congruence with c_{ij}, a_{ik}, b_{kj}.

(7) Assuming $|a_{ik}|, |b_{kj}| \leqslant U$, $mU^2 < p/2$, and $-p/2 < c_{ij} \leqslant p/2$, the constraint (1.1) ensures [Proposition 3.1] that

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

Therefore, if such a constraint is imposed for all i, j, we are assured that

$$AB = C.$$

Remark 1.1 (Detection of Negative-to-Positive Substitution Attack)**.** A dishonest prover might attempt to mask a negative matrix entry (e.g., $-a$ for some $a > 0$) by providing its least-residue modulo p, namely $p - a$, in order to claim it is positive. However, by design the circuit enforces that all entries satisfy

$$-\frac{p}{2} < x \leqslant \frac{p}{2}.$$

Since $p - a > p/2$ (for any $0 < a \leqslant U$, given that $p > 2U$), such a substitution would violate the range constraint. Hence, the circuit will detect any attempt to disguise a negative value in this manner. ■

[1] Assuming that each a_{ik} and b_{kj} is stored as a 64-bit signed integer (i.e., `i64`), we set $U = 2^{63}$. Provided $m \leqslant 2^{64}$, it follows that $mU^2 \leqslant 2^{190}$, which is less than $p/2$ for any prime $p > 2^{191}$. If no such off-circuit reasoning applies, or if tighter bounds are desired, a range check is required.

[2] A similar remark to Footnote 1 applies.

2. Parameter Roles: Public, Private, and Computed

For the circuit verifying matrix multiplication, we distinguish between public parameters, private inputs (the prover's witness), and values computed within the circuit. These roles are summarized as follows:

Public Parameters

- **Field Modulus p:** The prime p defining the finite field $\mathbb{Z}/p\mathbb{Z}$ is public and fixed during circuit setup.
- **Matrix Dimensions ℓ, m, n:** The dimensions of the matrices, with A of size $\ell \times m$, B of size $m \times n$, and C of size $\ell \times n$, are public. Hence, no in-circuit verification of these bounds is necessary.
- **Bound U:** The bound U on the absolute values of matrix entries (for example, $U = 2^{63}$ when using 64-bit signed integers) is public. This is used to ensure that $mU^2 < p/2$.

Private Inputs (Prover's Witness)

- **Matrices A and B:** These matrices are provided by the prover as private inputs. Their entries must be preprocessed into their least-residue representations modulo p (since the API does not allow signed integers).
- **Matrix C:** Typically, the prover also supplies the matrix C as the claimed product AB. Alternatively, if C is derived from other computations, it may be computed off-circuit and provided as input.

Computed Within the Circuit

- **Matrix Product AB:** The circuit computes the product $A \times B$ using the provided matrices A and B. This computed product is then compared against the supplied matrix C to verify correctness.
- **Derived Matrices:** In some applications (e.g., a neural network), one of the matrices (such as a weight matrix) may be computed within the circuit from other inputs rather than being provided directly.

In summary, public parameters such as p, the dimensions ℓ, m, n, and the bound U are fixed and known to all parties, while the matrices A, B, and C form the prover's private witness. The circuit uses these inputs to compute and verify the relation $AB = C$.

3. The Maths

Proposition 3.1. *Let p be a positive integer (not necessarily a prime), and let a_{ik}, b_{kj}, and c_{ij} be integers satisfying*

$$c_{ij} \equiv \sum_{k=1}^{m} a_{ik} b_{kj} \bmod p.$$

Assume there is a constant U such that $|a_{ik}| \leqslant U$ and $|b_{kj}| \leqslant U$ for all i,k,j, and also assume $|c_{ij}| \leqslant p/2$. If $mU^2 < p/2$, then

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

The congruence guarantees that the computed sum and c_{ij} coincide as integers.

Consequently, interpreting the entries a_{ik}, b_{kj}, c_{ij} as the components of matrices A, B, C, we conclude that $AB = C$ as integer matrices, not merely modulo p.

Proof. By the given congruence condition, there is an integer t such that

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

We claim $t = 0$. Suppose not; then $|t| \geqslant 1$. We split into two cases:

(1) Case $t \geqslant 1$. Then

$$c_{ij} \geqslant p + \sum_{k=1}^{m} a_{ik} b_{kj} \geqslant p - mU^2 > p - \frac{p}{2} = \frac{p}{2},$$

hence $c_{ij} > p/2$. But this contradicts the bound $c_{ij} \leqslant p/2$ (we assume $|c_{ij}| \leqslant p/2$).

(2) Case $t \leqslant -1$. Then
$$c_{ij} \leqslant -p + \sum_{k=1}^{m} a_{ik} b_{kj} \leqslant -p + mU^2 < -p + \frac{p}{2} = -\frac{p}{2},$$
hence $c_{ij} < -p/2$. But this contradicts the bound $c_{ij} \geqslant -p/2$ (we assume $|c_{ij}| \leqslant p/2$).

Since in either case we reach a contradiction, the only possibility is $t = 0$. This concludes the proof. ☐

4. The Code

We assume that the bounds of Proposition 3.1 hold and that all matrix elements have been converted to their least-residue equivalents in preprocessing.

Algorithm 4.1 `mat_mul`: verify matrix multiplication

Require: Matrices A of size $(\ell \times m)$, B of size $(m \times n)$, and C of size $(\ell \times n)$. Each entry is represented by a circuit variable.

function MATRIX_PRODUCT(A, B) ▷ Compute $A \times B$ in the circuit.
    $P \leftarrow$ new $(\ell \times n)$ matrix of circuit variables
    for $i \leftarrow 0$ to $\ell - 1$ **do**
        for $j \leftarrow 0$ to $n - 1$ **do**
            $P[i][j] \leftarrow \sum_{k=0}^{m-1} \big(A[i][k] \times B[k][j]\big)$
        end for
    end for
    return P
end function

function VERIFY_MAT_MUL(A, B, C)
    $P \leftarrow$ matrix_product(A, B)
    for $i \leftarrow 0$ to $\ell - 1$ **do**
        for $j \leftarrow 0$ to $n - 1$ **do**
            `assert_is_equal`$(C[i][j], P[i][j])$
        end for
    end for
end function

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

declare_circuit!(Circuit {
    // A has shape (\ell x m)
    matrix_a: [[Variable; M]; L],

    // B has shape (m x n)
    matrix_b: [[Variable; N]; M],

    // AB has shape (\ell x n)
    matrix_product_ab: [[Variable; N]; L],
});

impl<C: Config> GenericDefine<C> for Circuit<Variable> {
    fn define<Builder: RootAPI<C>>(&self, api: &mut Builder) {
        // ‘matrix_product‘ is a helper function that multiplies two
        // matrices within the circuit, returning a new (\ell x n) matrix.
        let computed_product = matrix_product(api, self.matrix_a, self.matrix_b);

        // For each entry (i, j) in the expected product, assert equality
        // with the corresponding computed entry in 'computed_product'.
        for i in 0..L {
            for j in 0..N {
                api.assert_is_equal(
                    self.matrix_product_ab[i][j],
                    computed_product[i][j],
```

```
                        );
                    }
                }
            }
}
```

Listing 1: ECC Rust API: verify matrix multiplication

References

[1] Polyhedra Network. *ExpanderCompilerCollection*. GitHub repository. Accessed January 28, 2025.