EinHops: Einsum Notation for Expressive Homomorphic Operations on RNS-CKKS Tensors

Karthik Garimella
New York University
kg2383@nyu.edu

Austin Ebel
New York University
abe5240@nyu.edu

Brandon Reagen
New York University
bjr5@nyu.edu

ABSTRACT

Fully Homomorphic Encryption (FHE) is an encryption scheme that allows for computation to be performed directly on encrypted data. FHE effectively closes the loop on secure and outsourced computing; data is encrypted not only during rest and transit, but also during processing. Moreover, modern FHE schemes such as RNS-CKKS (with the canonical slot encoding) encrypt one-dimensional floating-point vectors, which makes such a scheme an ideal candidate for building private machine learning systems. However, RNS-CKKS provides a limited instruction set: SIMD addition, SIMD multiplication, and cyclic rotation of these encrypted vectors. This restriction makes performing multi-dimensional tensor operations (such as those used in machine learning) challenging. Practitioners must pack multi-dimensional tensors into 1-D vectors and map tensor operations onto this one-dimensional layout rather than their traditional nested structure. And while prior systems have made significant strides in automating this process, they often hide critical packing decisions behind layers of abstraction, making debugging, optimizing, and building on top of these systems difficult.

In this work we ask: can we build an FHE tensor system with a straightforward and transparent packing strategy regardless of the tensor operation? We answer affirmatively and develop a packing strategy based on Einstein summation (einsum) notation. We find einsum notation to be ideal for our approach since the notation itself explicitly encodes the dimensional structure and operation directly into its syntax, naturally exposing how tensors should be packed and manipulated in FHE. We make use of einsum's explicit language to decompose einsum expressions into a fixed set of FHE-friendly operations: dimension expanding and broadcasting, element-wise multiplication, and a reduction along the contraction dimensions.

We implement our design and present EinHops, which stands for Einsum Notation for Homomorphic Tensor Operations. EinHops is a minimalist system that factors einsum expressions into a fixed sequence of FHE operations, enabling developers to perform complex tensor operations using RNS-CKKS while maintaining full visibility into the underlying packing strategy. We evaluate EinHops on a range of tensor operations from a simple transpose to complex multi-dimensional contractions. We show that the explicit nature of einsum notation allows us to build an FHE tensor system that is simple, general, and interpretable. We open-source EinHops at the following repository: https://github.com/baahl-nyu/einhops.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; • **Software and its engineering** → *Compilers*;

```python
import torch
import einhops

# batched matrix multiplication (pytorch)
a = torch.randn(2, 3, 4)
b = torch.randn(2, 4, 5)
c = torch.einsum("bij,bjk->bik", a, b)

# encrypted batched matrix multiplication (ckks)
a_ctxt = einhops.encrypt(a)
b_ctxt = einhops.encrypt(b)
c_ctxt = einhops.einsum("bij,bjk->bik",
                        a_ctxt,
                        b_ctxt)

# verify correctness
assert c.shape == c_ctxt.shape == (2, 3, 5)
assert torch.allclose(c, einhops.decrypt(c_ctxt))
```

Listing 1: EinHops introduces einsum notation [40] to the RNS-CKKS FHE scheme [12]. This example performs batched matrix multiplication where both operands are encrypted.

KEYWORDS

einsum notation; fully homomorphic encryption; eager execution; tensor operations; privacy-preserving machine learning systems

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) is a powerful encryption scheme that enables computation to be performed directly on encrypted data without the need for decryption [27]. This property makes FHE a promising fit for outsourced computation, especially for applications that have strict data privacy rules (e.g., health care). Within FHE, there are several schemes that operate over different data types: TFHE/CGGI for encrypted boolean circuits [16], BFV/BGV for arithmetic circuits over encrypted integer vectors [9, 10, 25], and CKKS for arithmetic circuits over encrypted complex-valued (and therefore real-valued) vectors [13]. For this reason, the CKKS scheme (in particular, RNS-CKKS [12]) has been used as the backbone for developing outsourced privacy-preserving machine learning services [4, 20, 23, 33, 37]. These systems execute encrypted deep learning applications by performing tensor operations within the constraints of RNS-CKKS.

When implementing encrypted multi-dimensional tensor operations in RNS-CKKS, we must first map our multi-dimensional data onto 1-D vectors. This issue is also present in canonical tensor libraries such as PyTorch or Jax in which multi-dimensional tensors

are mapped to blocks of memory (i.e., the logical representation versus the physical representation) [5, 8, 45]. The key difference when applying this mapping problem to RNS-CKKS is the limited instruction set provided by FHE: we only have access to SIMD (Single Instruction, Multiple Data) addition, SIMD multiplication, and cyclic rotations [12]. And unlike PyTorch, which can freely index into memory, FHE cannot access individual elements without rotating the entire vector. This makes indexing or accessing encrypted sub-tensors fundamentally more challenging.

Prior systems like Fhelipe and CHET [20, 33] have made significant strides in bridging this gap through sophisticated compiler infrastructures that automatically handle packing, bootstrapping, and optimization. However, these approaches often obscure the crucial packing decisions behind layers of abstraction. For instance, when developers use a high-level function like matvec or matmul, they lose visibility into whether their vectors are packed row-wise or column-wise, whether padding is used, or how reductions are implemented. This abstraction is appropriate for modern deep learning libraries that implement such operations using highly optimized kernels (e.g., cuDNN) [15, 35, 39]. On the other hand, linear algebra subroutines in FHE are still being developed without a consensus on the optimal packing strategy, given that the underlying cost model is inherently different. In FHE, we must ask: How many multiplicative levels does your method consume? What encoding procedure is being used? How many unique rotation keys are required? Can you use hoisted rotations? How much memory is required?

In this work, rather than building a large-scale system, we choose to build a simple FHE tensor system guided by the principle that the packing strategy and FHE costs should be explicit to the end-user. We find einsum notation to be an ideal candidate for our approach since it naturally exposes the dimensional structure of tensor operations through its notation. We implement our design in EinHops, which brings the simple but powerful einsum notation to the RNS-CKKS FHE scheme . Listing 1 shows how einsum notation can be used to perform multi-dimensional tensor operations by explicitly labeling dimensions over which to perform contractions. EinHops runs the equivalent operations on encrypted tensors.

EinHops implements this design philosophy by recognizing a key structural equivalence between einsum notation and a sequence of FHE operations. For example, the reduction in `"ij,jk->ik"` maps naturally to a sequence of operations in the RNS-CKKS slot space: first, a linear transformation to align the data, a sequence of rotations and summations to replicate data, an element-wise multiplication, and finally another sequence of rotations and additions to perform the reduction over the contracted dimension.

Concretely, we make the following contributions:

- We develop a transparent packing strategy based on einsum notation that exposes data layout decisions and remains consistent across arbitrary tensor operations.
- We implement and open-source our system, EinHops, with support for both CPU and GPU through an existing FHE backend, plus a cleartext backend for slot-level debugging.
- We evaluate EinHops on more than 15 tensor operations, showing that explicit packing enables operations such as 5-D tensor contractions while remaining interpretable.

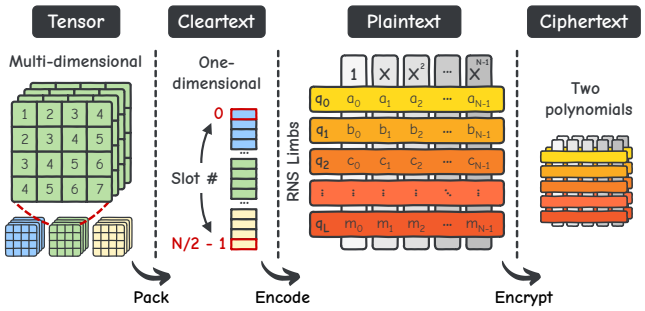

Figure 1: RNS-CKKS encrypts one-dimensional vectors of a fixed length (e.g., 2^{15} slots) and enables SIMD Addition, SIMD multiplication, and cyclic rotation upon these encrypted vectors. Multi-dimensional tensors must therefore be *packed* into one-dimensional cleartext vectors. These cleartext vectors are then *encoded* into plaintext polynomials that are decomposed using RNS [26]. Finally, plaintexts can be *encrypted* into ciphertexts, which consist of two RNS-polynomials.

The rest of the paper is organized as follows. Section 2 provides the relevant background information on the RNS-CKKS FHE scheme and einsum notation. In Section 3, we decompose einsum expressions into a series of explicit FHE-friendly operations, and in Section 4 we implement each of these steps using FHE primitives. We discuss the system design and limitations of EinHops in Section 5, and we report our results in Section 6.

2 BACKGROUND

2.1 RNS-CKKS

Here, we describe the RNS-CKKS homomorphic encryption scheme [12], its relevant datatypes, and the pipeline that consists of data packing, encoding, and encryption. A high level overview of this process is shown in Figure 1.

2.1.1 Datatypes. We interact with four primary datatypes in CKKS: tensors, cleartexts, plaintexts, and ciphertexts. Tensors are the familiar multi-dimensional arrays that represent our input data in its logical, high-level form and exist outside of the scope of FHE. A cleartext is a one-dimensional vector of a fixed power-of-two length (e.g., 2^{15} slots). We pack tensors into one or more cleartexts. A plaintext is then generated by encoding a cleartext vector into a single polynomial that resides in the ring $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N + 1)$, where N is a power-of-two degree (e.g., 2^{16}) and Q is a large integer modulus. Encryption then converts a plaintext into a ciphertext, which consists of a pair of polynomials in the product ring $\mathcal{R}_Q \times \mathcal{R}_Q$.

2.1.2 Packing. The process of flattening multi-dimensional tensors into one-dimensional cleartext vectors is called packing. Techniques for packing have been extensively explored in prior work [20, 23, 33, 36] and the best solutions are often problem-dependent. For example, packing strategies in convolutional neural networks [23] generally differ from those in language models [38] due to their distinct network architectures.

2.1.3 Encoding. The goal of encoding in CKKS is to find a polynomial in the ring $\mathcal{R}_Q$ whose evaluations at $N/2$ complex-valued roots of unity interpolate our cleartext vector. This process involves applying a variant of the inverse discrete Fourier transform. The conversion from point-value representation to coefficient representation guarantees that additions and multiplications between polynomials perform element-wise, SIMD operations on their underlying cleartexts. In this way, we can pack $N/2$ values into the *slots* of our cleartext vector. To satisfy RLWE constraints, a polynomial's coefficients must also be integers. Therefore before interpolating, we scale the evaluations by a large factor Δ (typically a power of two) to embed the desired precision into the integer parts. Then, rounding to the nearest integer incurs less error. Decoding performs the inverse transformation to recover scaled evaluations of any resultant cleartext. We note that other encoding schemes such as coefficient encoding exist [32], but do not preserve the SIMD properties we would like to utilize in this work.

2.1.4 Encryption. We can encrypt the scaled integer plaintext polynomial $m(X)$ into a ciphertext $ct = (b(X), a(X))$. The component $a(X)$ is typically sampled uniformly at random from the ring $\mathcal{R}_Q$. The second component, $b(X)$, is then constructed based on $a(X)$, the secret key $s(X)$ (a small polynomial in $\mathcal{R}_Q$), the plaintext $m(X)$, and a freshly sampled small error polynomial $e(X)$. Specifically, $b(X)$ is computed as $[-a(X) \cdot s(X) + m(X) + e(X)]_Q$, where the operations are performed with integer coefficients before the final reduction modulo Q. This structure securely masks $m(X)$. Decryption then uses the secret key $s(X)$ to reverse this masking. The decryption process computes $m'(X) = [b(X) + a(X) \cdot s(X)]_Q$. Substituting the definition of $b(X)$, we see the simplification:

$$\begin{aligned} m'(X) &\equiv b(X) + a(X) \cdot s(X) \pmod{Q} \\ &\equiv (-a(X) \cdot s(X) + m(X) + e(X)) + a(X) \cdot s(X) \pmod{Q} \\ &\equiv m(X) + e(X) \pmod{Q} \end{aligned}$$

As long as the coefficients of the combined $m(X) + e(X)$ are small enough, this final $m'(X)$ correctly recovers the original scaled plaintext plus the initial encryption error.

2.1.5 Homomorphic Operations. CKKS allows for computations directly on ciphertexts, corresponding to operations on the underlying cleartext vectors. The scheme supports element-wise addition and multiplication of these vectors. If ct_1 encrypts a vector corresponding to $m_1(X)$ and ct_2 encrypts a vector for $m_2(X)$, their homomorphic sum, $ct_{\text{add}} = ct_1 \oplus ct_2$, decrypts to approximately $m_1(X) + m_2(X)$, effectively adding the slot values. The same property holds for homomorphic multiplication. Additionally, CKKS supports homomorphic rotations (cyclic shifts) of the encrypted vector of slot values. This is achieved by applying a specific Galois automorphism to the ciphertext, which permutes the underlying data within the slots without decryption. All homomorphic operations are performed in the ring R_Q, where the ciphertext modulus Q must be very large to accommodate for noise growth.

2.1.6 Residual Number System (RNS). Working directly with the large integer modulus Q is compute intensive. To address this, CKKS implementations adopt the Residue Number System (RNS) [26]. The RNS approach decomposes the single large modulus Q into a product of several smaller, typically machine-word-sized, pairwise coprime moduli $q_0, q_1, \ldots, q_L$, so that $Q = Q_L = \prod_{i=0}^{L} q_i$. The integer L is known as the maximum multiplicative *level* of the ciphertext. By the Chinese Remainder Theorem, any integer coefficient c from a CKKS polynomial (which is an element of $\mathbb{Z}_{Q_L}$) can then be uniquely represented by a vector of its residues $(c \bmod q_0, \ldots, c \bmod q_L)$. Thus each polynomial in $\mathcal{R}_{Q_L}$ is transformed into a tuple of "limb" polynomials, where the i-th limb has coefficients modulo the small prime q_i. Operations originally performed on the large-coefficient polynomials are now efficiently computed through independent, parallel operations on their corresponding small-coefficient polynomial limbs using standard machine-word arithmetic.

2.1.7 Level Consumption. Homomorphic multiplication in CKKS consumes levels of the RNS modulus. This occurs because multiplication significantly increases both the magnitude of the scaled plaintext (from a scaling factor of Δ in the inputs to Δ^2 in the product before rescaling) and the accumulated noise. To manage these increases, *rescaling* is performed after multiplication. Rescaling effectively divides the ciphertext by the original scaling factor Δ (or an RNS prime q_j that approximates it) and, crucially, reduces the ciphertext modulus from the current $Q_l = \prod_{i=0}^{l} q_i$ to a smaller $Q_{l-1} = \prod_{i=0}^{l-1} q_i$ by dropping one of the RNS primes (e.g., q_l). Each rescaling operation therefore *consumes* one RNS modulus or one level. Since fewer limbs exist at lower levels, the latency of homomorphic operations increases with increasing level. When no further levels can be consumed, an expensive bootstrapping operation must be performed to refresh this multiplicative budget. As a result, the latency of bootstrapping often dominates the runtime of practical workloads such as private deep neural network inference.

2.2 Einsum Notation

2.2.1 Overview. Einsum notation is a language for performing tensor operations by explicitly stating over which dimensions to perform contractions [24]. This notation is a simple, but powerful abstraction that allows one to succinctly express a variety of tensor operations such as transposes, matrix-vector products, matrix-matrix multiplications, and higher-dimension tensor contractions.

While originally used in physics, einsum notation has since been adopted in modern tensor libraries such as PyTorch and Jax [5, 8], and each of these libraries is equipped with an `einsum` function. This function takes as input 1) an `equation` that specifies the tensor operation and 2) a series of operands. As an example, the expression `torch.einsum("ij->ji", x)` from Line 3 of Listing 2 tells us that we have a 2-D input matrix `x` with shape `"ij"`. Furthermore, the equation specifies that the resulting output tensor should have shape `"ji"`, informing us that we are transposing `x`.

By explicitly labeling both the input dimensions and the desired output dimensions, einsum notation facilitates tensor arithmetic while being *interpretable*. We make use of einsum's explicitness to develop EinHops, a system that performs encrypted tensor operations without obscuring the packing and implementation details.

2.2.2 Einsum Examples. Einsum notation is best understood by reading and running examples [41]. We begin with Listing 2 which instantiates a random tensor `x` of size $(3, 5)$. Each of the `einsum` calls

```
import torch
x = torch.randn(3,5)
torch.einsum("ij->ji", x) # transpose        (5,3)
torch.einsum("ij->", x)   # sum all elements
torch.einsum("ij->j", x)  # column-wise sum   (5,)
torch.einsum("ij->i", x)  # row-wise sum      (3,)
```

Listing 2: Einsum notation to perform a transpose and various summations for a 2-D tensor illustrating that einsum explicitly labels each dimension from the inputs.

specifies the input tensor as having shape `"ij"`, effectively labeling the axis `"i"`= 3 and `"j"`= 5. We could not have used `einsum` with the input dimensions being listed as`"ijk"` given that the input tensor `x` is two-dimensional.

The first `einsum` call (Line 3) performs the transpose operation by switching the order of the dimensions in the output (labeled `"ji"`). The subsequent calls to `einsum` perform a reduction over one or more dimensions by omitting these dimensions from the output equation. For example, Line 4 reduces over both dimensions by having no output dimensions specified. On the other hand, Line 5 reduces over the `"i"` dimension by stating that the output dimension should only be `"j"`= 5.

```
import torch
x = torch.randn(3,5)
y = torch.randn(5)

# matrix-vector product: (3,5) x (5,) -> (3,)
torch.einsum("ij,j->i", x, y)
```

Listing 3: A matrix-vector product using einsum notation. In this case, the dimension `"j"` is multiplied and then reduced.

Listing 3 has an example of an einsum expression with two inputs, `x` and `y`. The `equation` argument illustrates this by labeling each dimension of the inputs, separated by a comma. In this instance, we are performing a matrix-vector product. The `"j"` dimension is shared between both inputs and is omitted from the output dimensions. This tells us that we are performing a contraction over dimension `"j"`: the inputs are multiplied and then reduced along the `"j"` dimension, effectively computing dot products over `"j"`.

2.2.3 Einsum in Practice. We now examine a real-world application of the einsum notation from a recent ICLR 2024 paper [21]. In Listing 4, einsum notation appears in key steps when performing the multi-headed attention mechanism. By explicitly labeling each dimension, the einsum expressions are self-documenting, making them translatable to natural language. For example, Line 5 of Listing 4 can be read as: "for every sample in the batch (`b`) and for every head (`h`), construct the `t` by `T` attention matrix by contracting the inputs over their shared hidden dimension (`d`)".

3 DECOMPOSING EINSUM FOR FHE (*concept*)

In this section, we *examine* einsum notation from the perspective of FHE. In particular, we develop a mapping from einsum expressions to three FHE-friendly steps: 1) expanding and broadcasting inputs

```
import jax.nn as jnn
import jax.numpy as jnp

# batch_size, seq_len, n_heads, h_dim_per_head
attn = jnp.einsum("bthd,bThd->bhtT", q, k)
n_attn = jnn.softmax(attn)
output = jnp.einsum("bhtT,bThd->bthd", n_attn, v)
```

Listing 4: Since input and output dimensions are explicitly labeled, einsum notation is self-documenting as shown in this implementation of multi-headed attention.

to match dimensions, 2) performing an element-wise multiplication between the broadcasted inputs, and 3) reducing the resulting product across the contraction dimensions.

To clarify, standard einsum expressions are ***not*** implemented using these three steps; in reality, an `einsum` call will dispatch to a highly optimized backend implementation that depends upon the size, sparsity, and device associated with each input [2, 7]. Rather, our conceptual view of einsum lets us carry over the explicit dimensionality analysis into the realm of FHE by building a system that explicitly realizes this decomposition. We use Listing 5 and Figure 2 as our reference for this section, which performs a matrix-matrix multiplication between two 2-D tensors.

3.1 Matching Dimensions (`torch`)

First, we analyze the shapes of each input as well as the total number of input dimensions. In our running example, the einsum expression tells us that the input A has shape `"i"`= 4 and `"j"`= 5, whereas the input B has shape `"j"`= 5 and `"k"`= 2. The total number of input dimensions is three; we have `{ "i", "j", "k"}`.

Our first FHE-friendly step is to expand and match the shapes of the inputs. This means we must inflate A to include the missing input dimension `"k"` and inflate B to include the missing input dimension `"i"`. We label this Step ① and Step ② in Figure 2.

3.1.1 Expanding. We first match the *number* of input dimensions by simply adding singleton dimensions in an ordering that satisfies broadcasting rules [17]. In Figure 2, we insert these new axes using `A[:, :, None]` and `B[None, :, :]` although we could have also used `view`, `reshape`, or `unsqueeze`.

The shapes of A and B are now (4, 5, 1) and (1,5,2) as shown in Step ① in Figure 2. Note that we comply with broadcasting rules and that the contraction dimension (`"j"`= 5) is aligned at the `dim=1` position. This step performs no data duplication. Rather, it simply *views* the tensor within a higher dimension. As we will see in the following section, we can perform this operation in FHE either as a linear transformation or a `nop`, depending upon where the singleton dimensions must be added.

3.1.2 Broadcasting. Now that their shapes are aligned, we can broadcast both inputs A and B to match their sizes along each dimension. In this step, we duplicate the data across the singleton dimensions; we visualize this duplication in Step ② of Figure 2 where the tensors A and B now have shapes (4, 5, 2). In the following section, we will show how to perform this broadcasting across any dimension in FHE using a logarithmic number of operations.

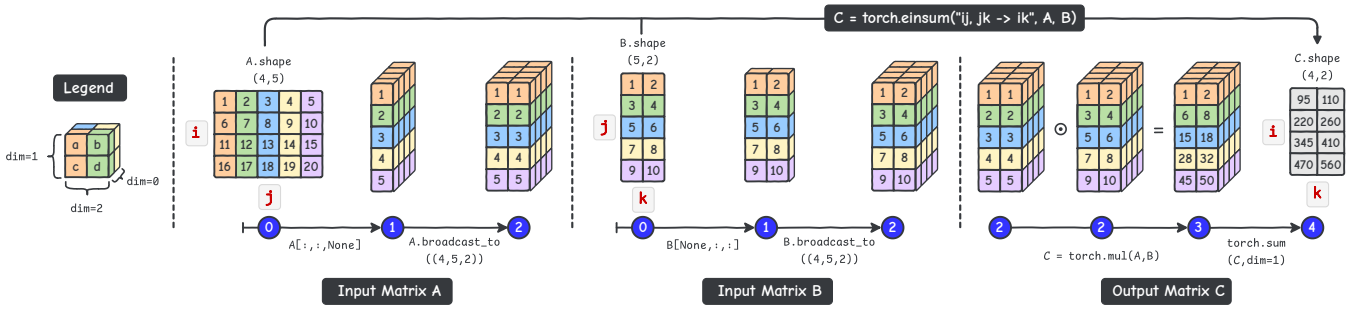

Figure 2: Einsum notation for a matrix-matrix multiplication in PyTorch of size $(4,5) \times (5,2) \rightarrow (4,2)$. The einsum equation labels these input dimensions as $(i,j) \times (j,k)$ and explicitly states the output shape as (i,k). The dataflow decomposes this einsum equation in an FHE-friendly manner. First, all operands are expanded and broadcasted to match tensor shapes. Then, operands are element-wise multiplied. Finally, the output is reduced by summing across the contraction dimensions.

```
import torch
A = torch.arange(start=1,end=21).reshape(4,5)
B = torch.arange(start=1,end=11).reshape(5,2)
C = torch.einsum("ij,jk->ik", A, B)
```

Listing 5: Matrix-matrix multiplication using `torch.einsum`.

3.2 Multiplication (torch)

Now that the input tensor shapes precisely match up, we may perform an element-wise Hadamard product between the inputs A and B. Here, we are effectively performing all partial products in the matrix-matrix multiplication in parallel, and this results in an output tensor C of shape $(4,5,2)$. This resulting tensor C is shown in Step ③ of Figure 2. We note that this multiplication naturally extends to the scenario when we have more than two inputs since all input tensors will have the same shape.

3.3 Reduction (torch)

Finally, we must reduce all partial products by summing over the contraction dimension, which in this case is `"j"`$= 5$. This step can be seen in Step ④ of Figure 2 where we sum out `dim=1` which corresponds to the labeled dimension `"j"`, resulting in a tensor of shape $(4,2)$. This resulting tensor precisely corresponds to the desired matrix-matrix multiplication performed in Listing 5. Similar to broadcasting, we will show how to perform this reduction over the contraction dimensions in a logarithmic number of FHE operations in the following section.

4 DECOMPOSING EINSUM FOR FHE (*impl.*)

We now *implement* our decomposed view of einsum from the previous section using RNS-CKKS primitives. As a consequence, we follow the same outline from the previous section and show the explicit set of FHE instructions that carry out 1) expanding and broadcasting, 2) element-wise multiplication, and 3) summation along the contraction dimensions. We assume that the product of all input dimensions fit within a single ciphertext, and that all dimensions are padded up to powers of two.

For this section, we demonstrate our low-level implementation details in Figures 3, 4, and 5. At the end of this section, we use Figure 6 to illustrate the end-to-end EinHops implementation of the same matrix-matrix multiplication from Listing 5 and Figure 2.

4.1 Matching Dimensions (FHE)

As we saw in the previous section, we must first expand our tensors by inserting singleton dimensions in order to match the *number* of inputs dimensions for each operand. In libraries such as PyTorch or Jax, such an expansion is trivial and for tensors allocated in contiguous memory blocks, these expansions only change tensor metadata without modifying the actual data itself [45].

However for encrypted tensors, there is a possibility that we must actually manipulate the ciphertext slot ordering given that indexing-based tensor operations are invalid (i.e. we only have SIMD Add, SIMD Mult, and cyclic rotation). There are two possible scenarios for dimension expansion which we detail below. To reiterate, we are not duplicating data during expansion; we are merely re-ordering data within the slots of a CKKS ciphertext.

4.1.1 Expanding inner dimensions. When inserting an inner dimension into an encrypted tensor, we must permute the underlying elements within the slots of the ciphertext. This is because inserting an inner dimension will change the *stride* of existing dimensions when the tensor is flattened [1]. To see why, let us examine the input tensor in the bottom of Figure 4 (`input.shape == (4,1)`). In this case, we must insert an inner dimension: $(4,1) \rightarrow (4,2)$. Before any expansion, the data within the input ciphertext is contiguously located in the top four slots of the underlying vector. However, adding an inner dimension will naturally stride the original elements, in this case by a stride of 2. This means we must insert placeholder slots for this new dimension between the original elements, and we do this by permuting the slots of the CKKS ciphertext using a linear transformation.

In more detail, we choose to perform these permutations as a linear transformation via a permutation matrix. Figure 3 (left) shows the Baby-Step Giant-Step (BSGS) algorithm that we use to perform this linear transformation. The BSGS algorithm is a derivative of the more straightforward Halevi-Shoup matrix-vector product algorithm. Halevi-Shoup multiplies the plaintext diagonals of a matrix

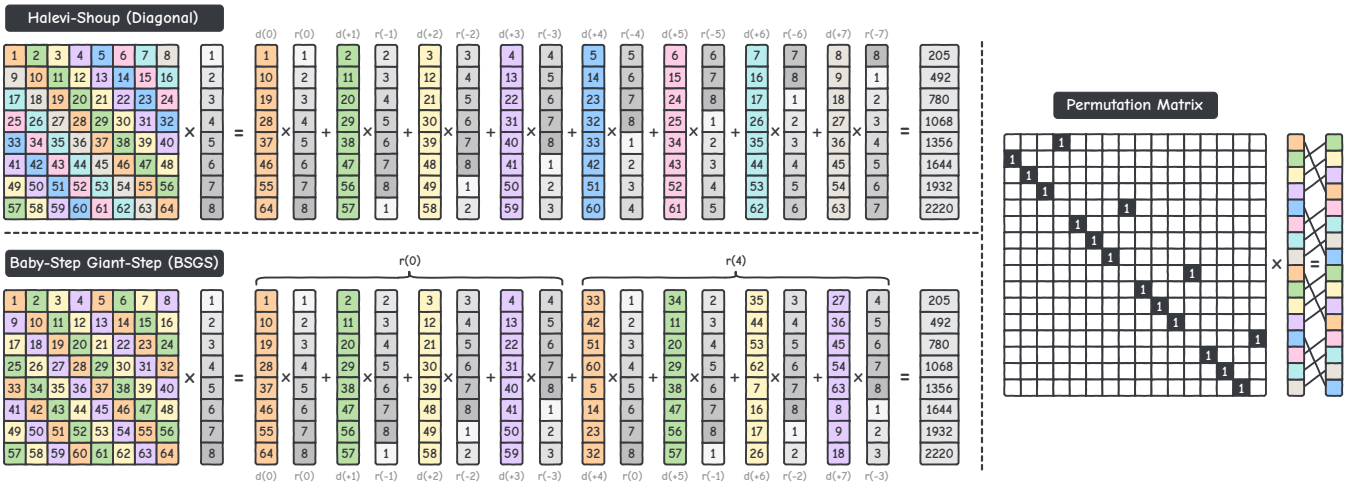

Figure 3: (*Left*) Matrix-vector products between an unencrypted matrix and an encrypted vector with n slots. The canonical Halevi-Shoup method multiplies the diagonals of the matrix with the (homomorphically) aligned ciphertext [28]. For a full $n \times n$ matrix, Halevi-Shoup requires n homomorphic rotations. The Baby-Step Giant-Step (BSGS) algorithm pre-rotates the cleartext diagonals, therefore reducing the homomorphic rotation count to $O(\sqrt{n})$ [11]. (*Right*) EinHops utilizes BSGS to re-arrange the slots of an encrypted vector by performing a matrix-vector product between a permutation matrix and a ciphertext.

with a homomorphically rotated ciphertext, effectively sharding each dot product across the aligned slots of many ciphertexts as shown. Concretely, the first row of the matrix performs a dot product with the ciphertext elements across slot index 0, the second row with slot index 1, and so forth. BSGS provides precisely the same interface as Halevi-Shoup but reduces the homomorphic rotation count from $O(n)$ to $O\sqrt{n}$ by instead rotating (i.e. `torch.roll`) the cleartext diagonals before encoding them into plaintexts. Figure 3 (right) shows that the diagonal structure of a permutation matrix fits naturally with BSGS.

4.1.2 Expanding outer dimensions. Expanding outer dimensions is a much simpler scenario since the original input data remains in contiguous slots. An example of this setup is shown in the top of Figure 4 (`input.shape == (1,4)`) where we must add an outer dimension: $(1, 4) \rightarrow (2, 4)$. Here, the original input data retains its stride of 1, so no permutation matrix is required to prepare this tensor for broadcasting. This effectively becomes a nop in FHE since the data is already in the correct order.

4.1.3 Broadcasting. At this stage, the input tensors have been expanded by inserting placeholder slots via BSGS in the case of inner dimension expansion or performing a nop in the case of outer dimension expansion. We are now ready to broadcast our original data across the new dimensions as shown in Figure 4.

Because we have chosen to pad all dimensions to a power of 2, we can perform broadcasting across any set of dimensions using the rotation-and-summation algorithm down the slots of the ciphertext while only requiring a logarithmic number of homomorphic operations with respect to the dimension size [43]. The *number* of homomorphic rotations is logarithmic in the *size* of the new dimension. The number of *places* by which we rotate the vector depends upon the *stride* of the new dimension. In both the top and

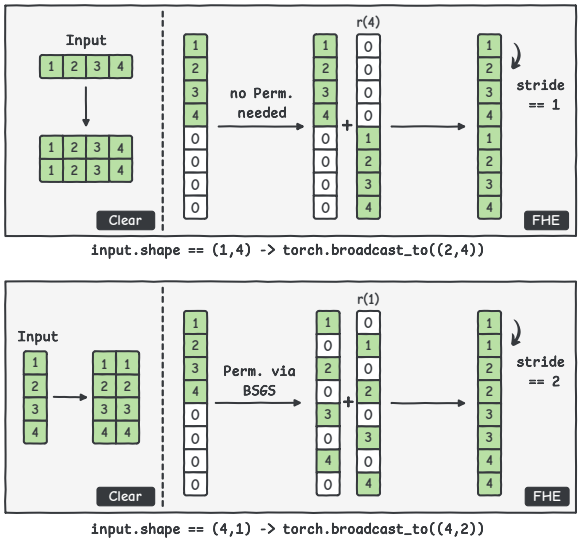

Figure 4: Broadcasting in FHE using $O(\log n)$ operations. (*Top*) Expanding outer dimensions ($(1, 4) \rightarrow (2, 4)$) does not permute the ordering of existing elements. (*Bottom*) Expanding inner dimensions ($(4, 1) \rightarrow (4, 2)$) strides existing elements and requires a permutation before broadcasting.

bottom of Figure 4, we only perform a single homomorphic rotation because the size of the new dimension is 2 and so $\log(2) = 1$. In the top of Figure 4, we rotate down by four since the stride of the new dimension is 4. In the bottom, we rotate down by one since the stride of the new dimension is 1.

4.2 Multiplication (FHE)

Now that all ciphertext tensors have been broadcasted using FHE primitives and have matching dimensions, it is straightforward to perform the multiplication using homomorphic SIMD Multiplication. For k operands, we can perform this step in $O(\log k)$ multiplicative levels by performing a tree-based multiplication.

4.3 Reduction (FHE)

Now, we must reduce the output ciphertext across the contraction dimensions. Similar to broadcasting, we can perform reductions across any arbitrary set of dimensions by using rotation-and-summation up the slots of the ciphertext. We illustrate how to perform this reduction in Figure 5 for a 2-D tensor. Again, we need a logarithmic number of rotations with respect to the sizes of the contraction dimensions, and the number of places by which we rotate is dictated by the strides of the contraction dimensions.

Here, we make a critical observation that facilitates a simpler FHE implementation. When performing a reduction over an inner dimension (e.g., `torch.einsum("ij->i", input)`) as shown in the bottom right of Figure 5, the desired sums are strided by the size of this inner dimension. This introduces gaps in the output ciphertext. On the other hand, reducing over an outer dimension (e.g., `torch.einsum("ij->j", input)`) as shown in the bottom left of Figure 5 naturally places the output sums in contiguous slots at the beginning of the ciphertext.

We use this observation to make sure that the resulting reduction places the computation in contiguous slots at the top of the ciphertext. In more detail, when we parse our einsum equation in EinHops to build our expanded and broadasted dimensions, we always place the contraction dimensions as the outer dimensions and the desired output dimensions as the inner dimensions in the correct order. This design choice enables us to forgo any permutation to re-align the output into contiguous slots in the output ciphertext.

4.4 Putting it all Together

We now use Figure 6 to illustrate this entire process of 1) expanding and broadcasting input dimensions, 2) performing the element-wise multiplication, and 3) reducing along the contraction dimensions for the same matrix-matrix multiplication example from Listing 5.

4.4.1 Inputs. The inputs A and B have shapes $(4, 5)$ and $(5, 2)$, respectively. These are padded up to the nearest power of two to $(4, 8)$ and $(8, 2)$, but we keep the original dimensions as an attribute for both tensors. We can now encrypt these two padded tensors. The equation `"ij,jk->ik"` in Listing 5 informs us that we are reducing over dimension `"j"` and desire an output dimension of `"ik"`. We choose our broadcasted dimension to be `"jik"` in order to place the contraction dimension as the outer dimension and the resulting dimensions as inner dimensions in the correct order.

4.4.2 Matching. The inputs A and B have dimensions `"ij"` and `"jk"`, respectively. Expanding both to have a dimension of `"jik"` requires inserting an inner dimension and so we need permutations for both inputs. Furthermore, the permutation for A includes the transpose from `"ij"` to `"ji"`. The permutations (Step ② in Figure 6) result in ciphertexts with placeholder slots for broadcasting. We now perform broadcasting using the rotation-and-summation algorithm to ensure that we fill up the first `"jik"` slots in each ciphertext with the correct and aligned data. This process is shown as Step ③ of Figure 6 and results in tightly packed ciphertexts.

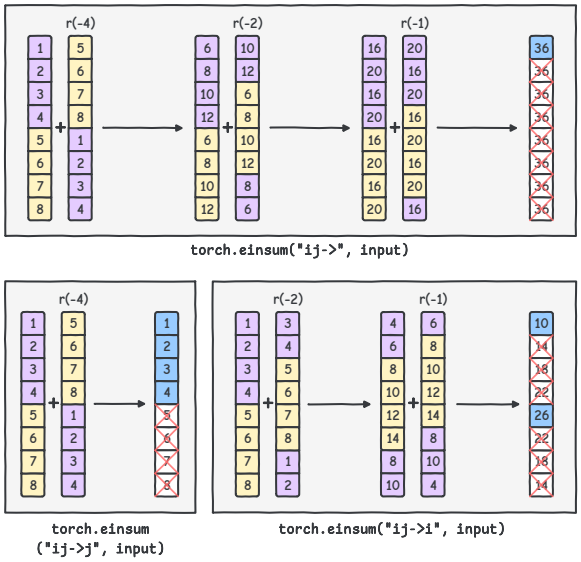

Figure 5: Reductions in FHE using $O(\log n)$ operations. Summing an inner dimension (`"ij->i"`) introduces gaps between the desired output, whereas summing the outer dimension (`"ij->j"`) produces the desired sums in contiguous slots.

4.4.3 Multiplying. We can now perform a simple SIMD multiplication using our backend FHE library. This will perform all the partial products required by the matrix-multiplication and result in a ciphertext tensor with the same dimension (`"jik"`) as shown in Step ④ of Figure 6.

4.4.4 Reducing. We now reduce over the contraction dimension `"j"`. We know that the (FHE) size of `"j"` is 8 and the stride of `"j"` is also 8. So, we perform the logarithmic rotation-and-summation algorithm up the ciphertext. This requires $\log(8) = 3$ steps and the places by which we rotate starts at 8 (the stride) and doubles each time. Since we placed the contraction dimension as the outer dimension, we've calculated our desired matrix-matrix multiplication in the top contiguous slots of the ciphertext. The output after this reduction is shown in Step ⑤ of Figure 6. We will perform a final masking of all other slots to maintain correctness for any further computations (i.e. multiplication by a binary vector).

5 EINHOPS SYSTEM

In this section, we discuss the overall design philosophy, implementation details, and limitations of EinHops. Our code is open-sourced at: https://github.com/baahl-nyu/einhops.

5.1 Design Philosophy

At a high level, the theme of EinHops is *minimalism* and *simplicity*. This means that we would like to be explicit about our packing strategy and we prefer eager execution versus graph execution which is the de-facto standard in FHE tensor systems [20, 23, 33].

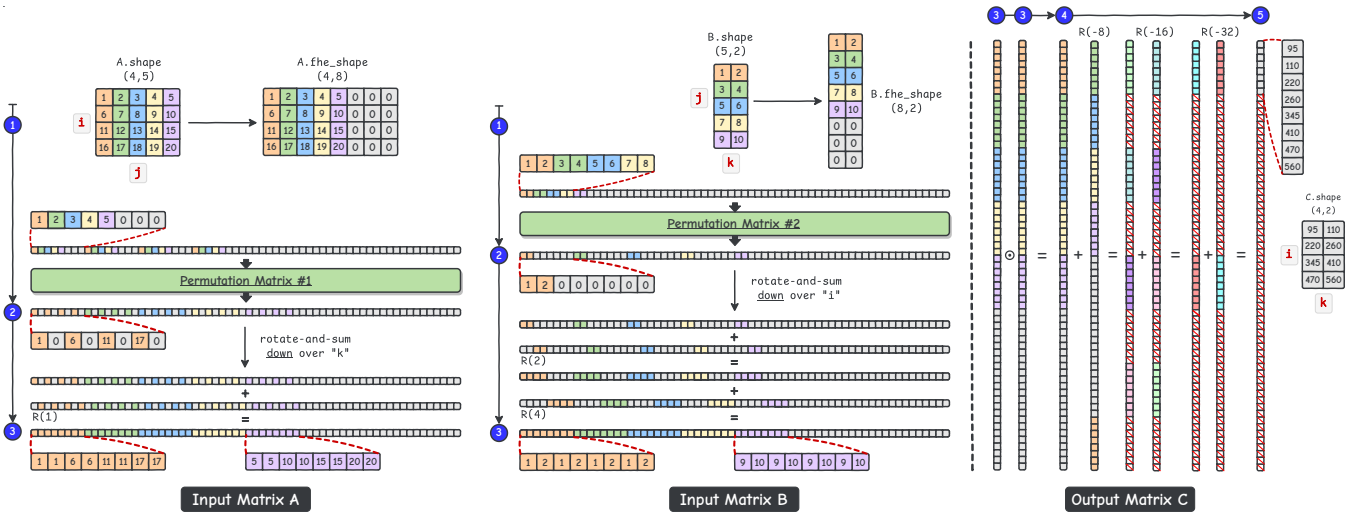

Figure 6: An overview of EinHops using the same example from Listing 5 and Figure 2. Step ① the einsum equation is parsed in order to determine the broadcast shape. Step ② if required, we permute the encrypted inputs using a BSGS permutation. Step ③ each input is broadcasted to match tensor shapes. Step ④ all inputs are multiplied using SIMD multiplication. Step ⑤ we perform a reduction to collapse the contraction dimensions. Step ⑥ a final mask ensures that all unused slots are zeroed out.

This design choice has ramifications in terms of raw performance but greatly simplifies the EinHops architecture.

5.1.1 Goals. Similar to PyTorch or Jax, we would like to provide an `einsum` interface for expressing tensor reductions and contractions in FHE. Inputs can either be tensors or encrypted tensors. We target small FHE programs such as tensor contractions, matrix-vector products, and matrix-matrix multiplication. Finally, we would like to provide a simple and readable code-base to provide a scaffolding for future research and engineering.

5.1.2 Non-goals. We do not support running large FHE programs (e.g., encrypted ResNet-20 or LLM inference). Running such programs requires a robust system that handles the sharper bits of FHE (e.g., memory management, bootstrap placement, scale management, data normalization, and non-linear approximations) [23].

5.2 Implementation

Figure 7 shows the overall system architecture for EinHops, which we implement in roughly 1,000 lines of pure `Python` code.

5.2.1 Backend. We currently support the `Liberate.FHE` library by using their `desilo-fhe` `Python` frontend [22]. This library lowers FHE operations to both CPUs and GPUs. Additionally, we support an artificial backend of PyTorch 1-D cleartext tensors. This means that we can simulate homomorphic operations using `torch.add`, `torch.mul`, and `torch.roll`, which brings debugging FHE programs to the slot level with minimal overhead. We use the `opt_einsum` package to parse and validate einsum expressions.

5.2.2 Hoisted Baby-Step Giant-Step. Since the `desilo-fhe` backend only provides an interface for homomorphic SIMD addition, SIMD multiplication, and cyclic rotation on `desilo` objects, we implement the baby-step giant-step algorithm directly in `Python` with the single hoisting optimization that amortizes the cost of key-switching within each baby step [29].

5.3 Limitations

Here, we briefly describe the current limitations of EinHops.

5.3.1 Single Ciphertext. Currently, EinHops only supports operations between single ciphertexts and as a consequence, the product of all input dimensions to an einsum expression must fit within a single ciphertext, effectively limiting our total dimension size to $N/2$. Adding support for multi-ciphertext tensors unlocks a new scaling dimension for EinHops. Given that EinHops builds wide computation graphs, task-based optimization can be used to increase throughput [6]. At the same time, the multi-ciphertext case raises several crucial questions over the meaning of applying homomorphic operations across ciphertexts. For example: what does it mean to rotate a multi-ciphertext by some value k? Is this a global rotation by k or a per-ciphertext rotation by k?

5.3.2 Bootstrap Placement. We do not currently implement any bootstrapping placement policy. While it is straightforward to implement a lazy bootstrapping strategy, this method has shown to scale poorly, especially with arithmetic circuits that contain residual connections [23].

5.3.3 Permutations. We implement permutations right now as BSGS linear transformations. However, it would beneficial to employ a different permutation strategy such as the Vos-Vos-Erkin method [34, 44] and analyze the change in terms of required memory, latency, and level consumption.

5.3.4 jit support. Compiling a series of einsum calls in EinHops may reveal better packing and permutations strategies rather than employing eager execution mode.

Operation	Syntax	Input Dimensions	FHE CPU (s)	FHE CPU + Keys (s)	FHE GPU (s)	FHE GPU + Keys (s)
Matrix transpose	`ij → ji`	(128, 128)	60.55	19.58	11.52	5.91
Matrix sum	`ij →`	(128, 128)	2.80	2.09	0.44	0.40
Column sum	`ij → j`	(128, 128)	0.77	0.49	0.12	0.11
Row sum	`ij → i`	(128, 128)	62.59	19.83	10.16	6.26
Matrix × vector	`ik, k → i`	(128, 128), (128)	124.41	34.84	17.69	11.58
Matrix × matrix	`ik, kj → ij`	(16, 32), (32, 32)	27.16	10.17	5.91	4.97
Dot product	`i, i →`	(16384,)	2.50	1.63	0.27	0.33
Inner product	`ij, ij →`	(128, 128), (128, 128)	2.26	1.84	0.27	0.31
Hadamard product	`ij, ij → ij`	(128, 128), (128, 128)	0.05	0.03	0.01	0.01
Outer product	`i, j → ij`	(128,), (128,)	61.16	16.21	8.55	5.00
Batched matrix × matrix	`ijk, ikl → ijl`	(16, 8, 8), (16, 8, 8)	11.54	3.70	4.12	4.01
3-way Hadamard	`ij, ij, ij → ij`	(128, 128), (128, 128), (128, 128)	0.08	0.05	0.01	0.01
Chained matrix × matrix	`ij, jk, kl → il`	(16, 8), (8, 8), (8, 16)	85.34	21.31	14.22	9.30
Bilinear transform	`ik, jkl, il → ij`	(8, 16), (8, 16, 16), (8, 16)	167.89	69.16	29.2	27.07
Tensor contraction	`pqrs, tuqvr → pstuv`	(2, 4, 8, 8), (1, 4, 4, 2, 8)	104.28	28.13	16.12	12.04

Table 1: EinHops on a CPU and GPU FHE backend. The "+ Keys" columns indicate the we generate the specific rotations keys needed for the BSGS matrix-vector product apriori. BSGS rotations keys increase working memory from ≈ 3 GB to ≈ 32GB, but reduce computation by enabling a single-step rotation rather than a series of power-of-two-step rotations for the BSGS stage.

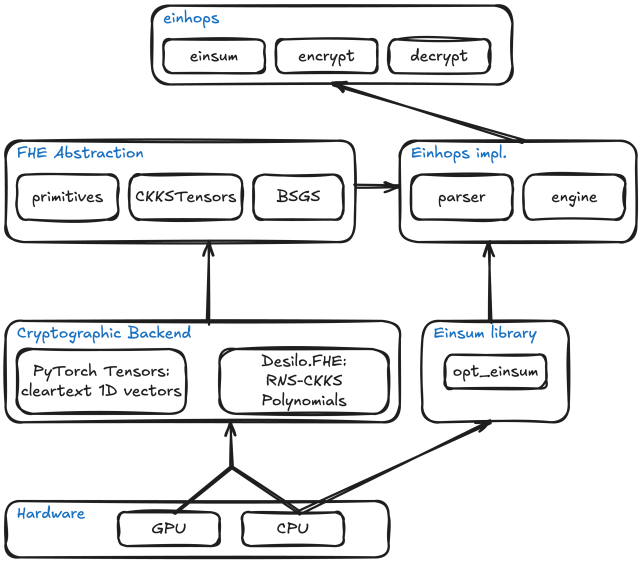

Figure 7: EinHops systems overview from the hardware to user interface. We implement this system in pure `Python`.

6 EVALUATION

In this section, we report our results for evaluating EinHops on a series of 15 einsum expressions [41] as well as calculating the multi-headed self-attention score (Line 5 of Listing 4). All experimental results are averaged over 10 runs and we observe a small standard deviation in each case. We also calculate the ℓ_2 norm of the difference between the PyTorch and EinHops `einsum` expressions, and observe a discrepancy on the order of roughly 10^{-5} in each case. For every experiment, we choose to encrypt all inputs to the einsum expression, although this does not have to be the case. Indeed, the EinHops implementation of `einsum` accepts both torch tensors and ciphertexts.

6.1 Experimental Setup

All experiments are performed on an Intel(R) Xeon(R) Platinum 8480+ processor (26 threads, 13 cores, 2 threads per core) that is equipped with a single NVIDIA H100 PCIe GPU (80GB memory). We run the FHE backend on both the CPU (using all 26 threads) and the GPU. We use the medium engine provided by the `desilo-fhe` backend which has a polynomial degree of 2^{15} and a corresponding slot count of 16384. We run each experiment at the minimum starting level: this is level 4 for `Chained matrix × matrix` and `Bilinear transform` and level 3 for all other einsum expressions.

For both the CPU and GPU backend, we also run two different configurations of EinHops. The first configuration is memory-efficient but requires more compute by generating only the power-of-two rotation keys (14 keys in our setup). This means, for example that rotating a ciphertext by 3 slots requires two distinct rotations: a rotation by 1 slot, followed by a rotation by 2 slots. The second variant (labeled `+ Keys` in Table 1) is compute-efficient but requires more working memory by generating all rotation keys needed for a general BSGS matrix transformation (14 + 256 keys in our setup).

6.2 Results

Table 1 lists our results for running 15 different einsum expressions using EinHops [41] for both the CPU and GPU backend as well as

our two key-generation configurations. We list the tensor operation along with the corresponding einsum syntax on the left side of the Table. For example, the `Chained matrix × matrix` operation performs two back-to-back matrix-matrix multiplications where all matrices are ciphertexts. For each experiment, we specify the input dimension so as to use a majority of the available slots in the CKKS ciphertext (16384 slots in our setup).

6.2.1 Power-of-Two Keys Only. For the memory-efficient setup, which generates only the power-of-two rotation keys, we find that in most cases the GPU backend achieves roughly a 6× speedup compared to the CPU backend. For example, transposing a 128×128 matrix takes 60.55 seconds on the CPU backend while taking only 11.52 seconds on the GPU. In this particular instance, no reduction is occurring and all time is spent permuting the ciphertext slots using BSGS. Still, this memory-efficient configuration requires only 3 GB of working memory, which lowers the barrier to entry for running and understanding FHE programs. While we choose to run our experiments on server-grade hardware, this modest memory requirement means EinHops can run on consumer-grade equipment such as a laptop or an RTX 3090 GPU.

6.2.2 Power-of-Two + BSGS Keys. The compute-efficient setup, which generates both the power-of-two rotation keys and the BSGS rotation keys immediately lowers runtime in both the CPU and GPU setting. This is because we are able to perform the rotations with the pre-computed BSGS rotation keys rather than using a series of power-of-two steps [30]. Furthermore in this setup, we are able to leverage single-hoisting to amortize the Decompose component of rotations for the baby-step rotations [29]. As an example, the compute-efficient setup on the GPU results in a 10.7× speedup to compute a *ciphertext-ciphertext* matrix-vector product on inputs of size (128×128) and $(128, 1)$ when compared to the CPU backend which generates only the power-of-two rotation keys. Still, this compute-efficient configuration requires roughly 32 GB of working memory to hold all power-of-two and BSGS rotation keys.

6.2.3 Case Study: Multi-Headed Self Attention Scores. Besides the tensor operations from Table 1, we also use EinHops to compute the multi-head attention score from Listing 4 (Line 5). We set the batch size, sequence length, number of heads, and hidden dimension to be `b=2`, `t=T=5`, `h=8`, `d=16`, respectively. Again, both inputs to the attention mechanism are encrypted. We are able to compute the attention matrix for each sample and each head in 28.57 seconds with roughly 23 seconds being spent aligning data using the BSGS transformation. This experiment illustrates the importance of researching strategies for re-aligning data within CKKS slots. Overall, our results demonstrate that einsum notation provides a clean abstraction for FHE tensor operations without sacrificing the transparency of underlying slot manipulations.

7 RELATED WORKS

The development of EinHops is informed by two primary areas of research: the evolution of compilers for FHE and the use of einsum notation in tensor libraries.

FHE Compilers: The field of FHE compilation has rapidly matured to address the significant performance and programmability challenges of computing on encrypted data. Early works focused on circuit-level optimizations, and more recent domain-specific compilers such as CHET [20], EVA [19], Porcupine [18], and Hecate [37] targeted shallow machine learning workloads. These systems introduce abstractions to automate complex tasks such as encryption parameter selection, noise management through sophisticated rescaling strategies (e.g., EVA's "waterline"), and optimizing data layouts for SIMD execution. More advanced compilers such as Orion [23], Dacapo [14], and Fhelipe [33] tackle the challenge of *deep* learning by automating the placement of bootstrapping operations.

A common thread among prior FHE compilers is their reliance on generating static computational graphs. These systems typically parse a program into an intermediate representation and apply a series of optimization passes to manage noise, latency, and scheduling. We instead adopt an eager execution model akin to PyTorch [5]. Instead of building and optimizing a graph, the einsum expression itself serves as an explicit plan. This design prioritizes simplicity and transparency, making the packing strategy explicit rather than abstracting it away behind an optimization framework.

Einsum Notation: Einsum notation has become a staple in modern tensor libraries such as PyTorch [5], JAX [8], and NumPy [31] for its ability to express complex tensor operations in a single, concise line of code. The notation's power has led to the development of specialized optimization packages like `opt_einsum` [2], which focuses on finding the most computationally efficient contraction order for a given expression. The notation has proven particularly valuable in implementing attention mechanisms in transformers [42], where complex batched matrix operations can be expressed more clearly than with traditional BLAS calls. While einsum implementations typically compile down to optimized GEMM routines, the notation itself serves as a powerful intermediate representation. Libraries such as TensorFlow [3] use einsum expressions internally to represent and optimize operations before lowering to platform-specific kernels. This dual nature of einsum as both a user-facing API and an optimization target makes it uniquely suited for bridging high-level tensor expressions with low-level execution strategies. EinHops leverages this same expressive power to provide a clean abstraction for homomorphic tensor operations.

8 CONCLUSION

In this work, we decomposed the expressive einsum notation into a series of FHE-friendly operations and built EinHops, a system for performing high-dimensional tensor operations in FHE. By mapping the semantics of einsum to a sequence of permutations (via BSGS), broadcasts, and reductions (via rotate-and-sum), EinHops brings a familiar and highly expressive programming model to the constrained environment of RNS-CKKS.

Unlike graph-based compilers that abstract away implementation details behind complex optimization passes, EinHops makes the data layout and operational plan explicit through the einsum string itself. This transparency empowers researchers to reason clearly about the underlying homomorphic operations, how slots are being used, and apply optimizations at each stage of the pipeline. We evaluate EinHops on a variety of tensor operations, from simple transposes to multi-head attention scores used in transformers, demonstrating that einsum notation provides an intuitive yet powerful abstraction for FHE tensor operations.

We embraced minimalism and simplicity throughout this project. We hope our design philosophy makes learning about FHE more accessible and helps practitioners better explore system-level nuances of running FHE programs.

ACKNOWLEDGEMENTS

This work was supported in part by Graduate Assistance in Areas of National Need (GAANN). The research was developed with funding from the NSF CAREER award #2340137 and DARPA, under the Data Protection in Virtual Environments (DPRIVE) program, contract HR0011-21-9-0003. Reagen and Ebel received a gift award from Google. The views, opinions, and/or findings expressed are those of the authors and do not necessarily reflect the views of sponsors.

REFERENCES

[1] 2025. torch.Tensor.stride — PyTorch Documentation. https://docs.pytorch.org/docs/stable/generated/torch.Tensor.stride.html. (2025). Accessed: 2025-06-26.

[2] Daniel G. a. Smith and Johnnie Gray. 2018. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753. https://doi.org/10.21105/joss.00753

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[4] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Proceedings on Privacy Enhancing Technologies* 2023, 1 (Jan. 2023), 325–342. https://doi.org/10.56553/popets-2023-0020

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christiao Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. https://doi.org/10.1145/3620665.3640366

[6] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the Gap Between CUDA and Task Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '24)*. IEEE Press, Article 43, 17 pages. https://doi.org/10.1109/SC41406.2024.00049

[7] Mark Blacher, Christoph Staudt, Julien Klaus, Maurice Wenig, Niklas Merk, Alexander Breuer, Max Engel, Sören Laue, and Joachim Giesen. 2024. Einsum Benchmark: Enabling the Development of Next-Generation Tensor Execution Engines. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 98033–98048. https://proceedings.neurips.cc/paper_files/paper/2024/file/b1bbfdb9197bfc819a52c34dce493f85-Paper-Datasets_and_Benchmarks_Track.pdf

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. (2018). http://github.com/jax-ml/jax

[9] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. Cryptology ePrint Archive, Paper 2012/078. (2012). https://eprint.iacr.org/2012/078

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. Cryptology ePrint Archive, Paper 2011/277. (2011). https://eprint.iacr.org/2011/277

[11] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2018/153. (2018). https://eprint.iacr.org/2018/153

[12] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A Full RNS Variant of Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2018/931. (2018). https://eprint.iacr.org/2018/931

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Paper 2016/421. (2016). https://eprint.iacr.org/2016/421

[14] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. 2024. DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 6993–7010. https://www.usenix.org/conference/usenixsecurity24/presentation/cheon

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. (2014). arXiv:cs.NE/1410.0759 https://arxiv.org/abs/1410.0759

[16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. (2018). https://eprint.iacr.org/2018/421

[17] PyTorch Contributors. 2024. Broadcasting semantics. https://docs.pytorch.org/docs/stable/notes/broadcasting.html. (2024). Accessed: 2025-06-26.

[18] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: a synthesizing compiler for vectorized homomorphic encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 375–389. https://doi.org/10.1145/3453483.3454050

[19] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3385412.3386023

[20] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 142–156. https://doi.org/10.1145/3314221.3314628

[21] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. 2024. Language Modeling Is Compression. In *ICLR*.

[22] DESILO. 2023. Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy. (2023). https://github.com/Desilo/liberate-fhe.

[23] Austin Ebel, Karthik Garimella, and Brandon Reagen. 2025. Orion: A Fully Homomorphic Encryption Framework for Deep Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 734–749. https://doi.org/10.1145/3676641.3716008

[24] Albert Einstein. 1916. The foundation of the general theory of relativity. *Annalen Phys.* 49, 7 (1916), 769–822. https://doi.org/10.1002/andp.19163540702

[25] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. (2012). https://eprint.iacr.org/2012/144

[26] Harvey L. Garner. 1959. The residue number system. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western))*. Association for Computing Machinery, New York, NY, USA, 146–153. https://doi.org/10.1145/1457838.1457864

[27] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[28] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. Cryptology ePrint Archive, Paper 2014/106. (2014). https://eprint.iacr.org/2014/106

[29] Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 93–120. https://doi.org/10.1007/978-3-319-96884-1_4

[30] Kyoohyung Han and Dohyeong Ki. 2019. Better Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Paper 2019/688. (2019). https://eprint.iacr.org/2019/688

[31] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[32] Dongwoo Kim and Cyril Guyot. 2023. Optimized Privacy-Preserving CNN Inference With Fully Homomorphic Encryption. *IEEE Transactions on Information Forensics and Security* 18 (2023), 2175–2187. https://doi.org/10.1109/TIFS.2023.3263631

[33] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. *Proc. ACM Program. Lang.* 8, PLDI, Article 152 (June 2024), 25 pages. https://doi.org/10.1145/3656382

[34] Jeremy Kun. 2024. Shift Networks. (Sept. 2 2024). https://www.jeremykun.com/2024/09/02/shift-networks/ Math and Programming blog post.

[35] C. Lawson, Richard Hanson, David Kincaid, and Fred Krogh. 1979. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Softw.* 5 (09 1979), 308–323. https://doi.org/10.1145/355841.355847

[36] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.), Vol. 162. PMLR, 12403–12422. https://proceedings.mlr.press/v162/lee22e.html

[37] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. 2022. HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–204. https://doi.org/10.1109/CGO53902.2022.9741265

[38] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. 2024. THOR: Secure Transformer Inference with Homomorphic Encryption. Cryptology ePrint Archive, Paper 2024/1881. (2024). https://eprint.iacr.org/2024/1881

[39] NVIDIA Corporation. 2025. *cuBLAS Library*. NVIDIA Corporation. Version 12.9. Documentation: https://docs.nvidia.com/cuda/cublas/.

[40] PyTorch Team. 2025. torch.einsum. (2025). https://pytorch.org/docs/stable/generated/torch.einsum.html Accessed: 2025-06-21.

[41] Tim Rocktäschel. 2018. Einsum is All You Need – Einstein Summation in Deep Learning. (2018). https://web.archive.org/web/20250514130020/https://rockt.ai/2018/04/30/einsum Accessed: 2025-06-25.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. (2023). arXiv:cs.CL/1706.03762 https://arxiv.org/abs/1706.03762

[43] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. HECO: fully homomorphic encryption compiler. In *Proceedings of the 32nd USENIX Conference on Security Symposium (SEC '23)*. USENIX Association, USA, Article 264, 18 pages.

[44] Jelle Vos, Daniël Vos, and Zekeriya Erkin. 2022. Efficient Circuits for Permuting and Mapping Packed Values Across Leveled Homomorphic Ciphertexts. In *Computer Security – ESORICS 2022*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng (Eds.). Springer International Publishing, Cham, 408–423.

[45] Edward Z. Yang. 2019. PyTorch Internals. https://blog.ezyang.com/2019/05/pytorch-internals/. (May 2019). https://blog.ezyang.com/2019/05/pytorch-internals/ Long-form essay based on a talk at the PyTorch NYC meetup (May 14, 2019).