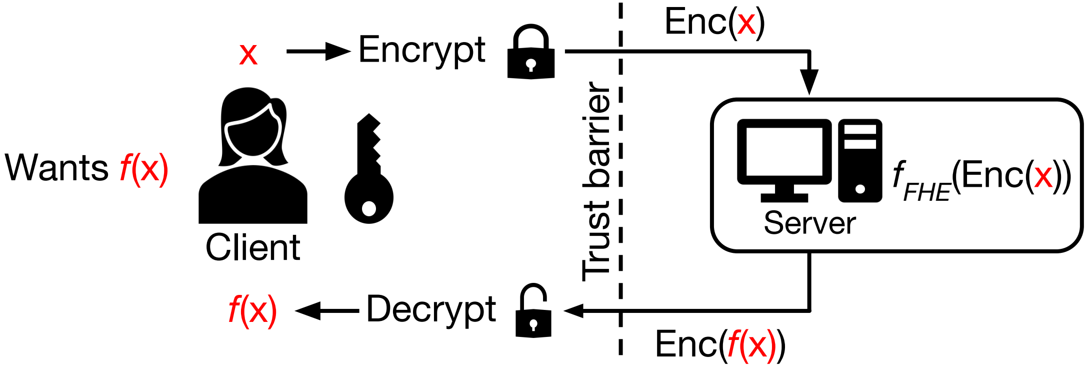
A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption

Aleksandar Krastev*, Nikola Samardzic*, Simon Langowski, Srinivas Devadas, Daniel Sanchez

*Authors contributed equally

Why use Fully Homomorphic Encryption (FHE)?

- Compute on encrypted data
- Private cloud computing



- The server never decrypts anything!
- 10,000× slower on CPU
  - GPU[1], FPGA [FAB[2], Poseidon[3]]: 100× speedup
  - ASICs [SHARP[4], ARK[5], BTS[6], CraterLake[7]]: 10,000× speedup

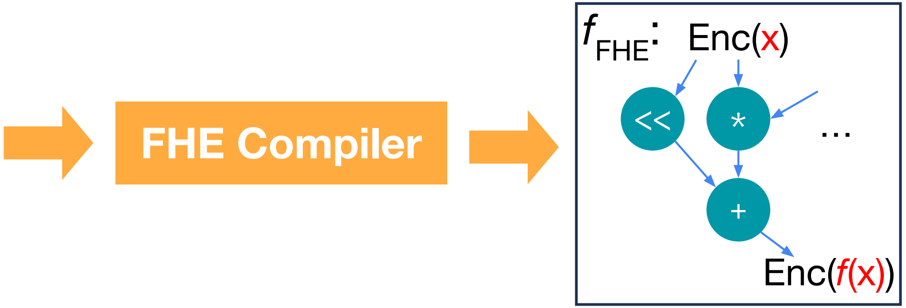
1. Jung et al. '21
2. Agrawal et al. HPCA '23
3. Yang et al. HPCA '23
4. Kim et al. ISCA '23
5. Kim et al. MICRO '22
6. Kim et al. ISCA '22
7. Samardzic et al. ISCA '22

Why an FHE Compiler?

- $f_{FHE} \neq f$; writing f_{FHE} is hard
  - ResNet[1], RNN[2], Logistic Regression[3], CryptoNets[4]

```python
def f(x: Tensor):
  for i in range(10):
    x = convLayer(x, w)
  return x
```

High-level description



Low-level FHE ops

- Felipe bridges this gap!
- vs. manual:
  - 10–48× less code
  - 1.0–12.5× faster

1. Lee et al. ICML '22
2. Podschwadt and Takabi '20
3. Han et al. '19
4. Brutzkus et al. ICML '19

Fhelipe's Contributions and Prior Compilers

Data Layouts

- FHE: Huge vectors with expensive reorder
- Manual: avoid reordering ⇒ many layouts
  - Fast, but hard to write
- Prior compilers:
  - Few layouts: slow
    [CHET[1], HECO[2], HeLayers[3]]
  - Superoptimizers: tiny programs
    [Coyote[4]]
- Fhelipe: novel layout representation
  - Large programs; 2.2–322.4× faster

Focus of this talk

Noise Management

- FHE: Random noise for security
  - Aux ops: **rescale** and **bootstrap**
- Rescale: prior compilers do well
  [EVA[5], HECATE[6], ELASM[7]]
- Bootstrap: limited prior work
  - Appears only in large programs
- Fhelipe: novel bootstrap placement
  - First to match manual

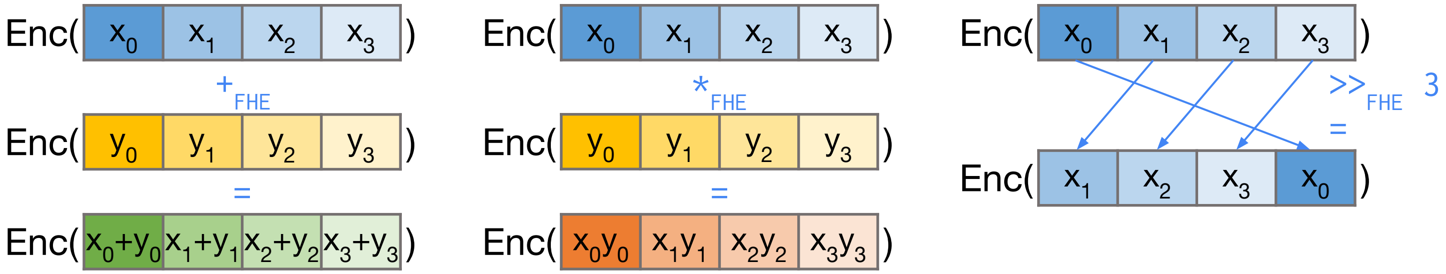
In the paper

1. Dathathri et al. PLDI '19
2. Viand et al. USENIX Sec '23
3. Aharoni et al. PoPETs '23
4. Malik et al. PLDI '23
5. Dathathri et al. PLDI '20
6. Lee et al. CGO '22
7. Lee et al. USENIX Sec '23

Challenges of FHE Layouts

FHE Programing Interface

- CKKS (state-of-the-art FHE scheme)
- Ciphertexts: vectors of fixed-point numbers



- No random access or reorder
- Large vectors: 32K elements
  - Unused slots are wasted
- Good fit: linear algebra & machine learning

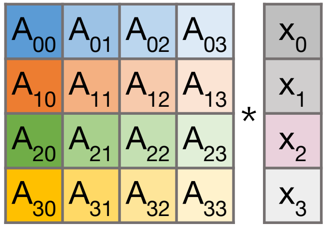
Example: Sequence of Matrix-Vector Multiplies

- E.g., fully-connected or recurrent NNs
- A: [128 × 128]; x: [128]
  - Fills a 16K-element ciphertext
- Diagrams: [128 × 128] ⟶ [4 × 4]
  16K ⟶ 16 elements/ciphertext

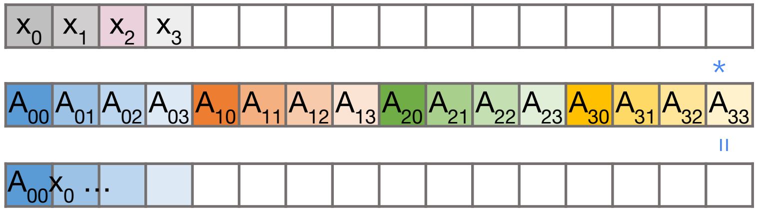
```
x: Vector
A: List[Matrix]

for A_i in A:
  x = mv_mul(A_i, x)
return x
```

Tensors

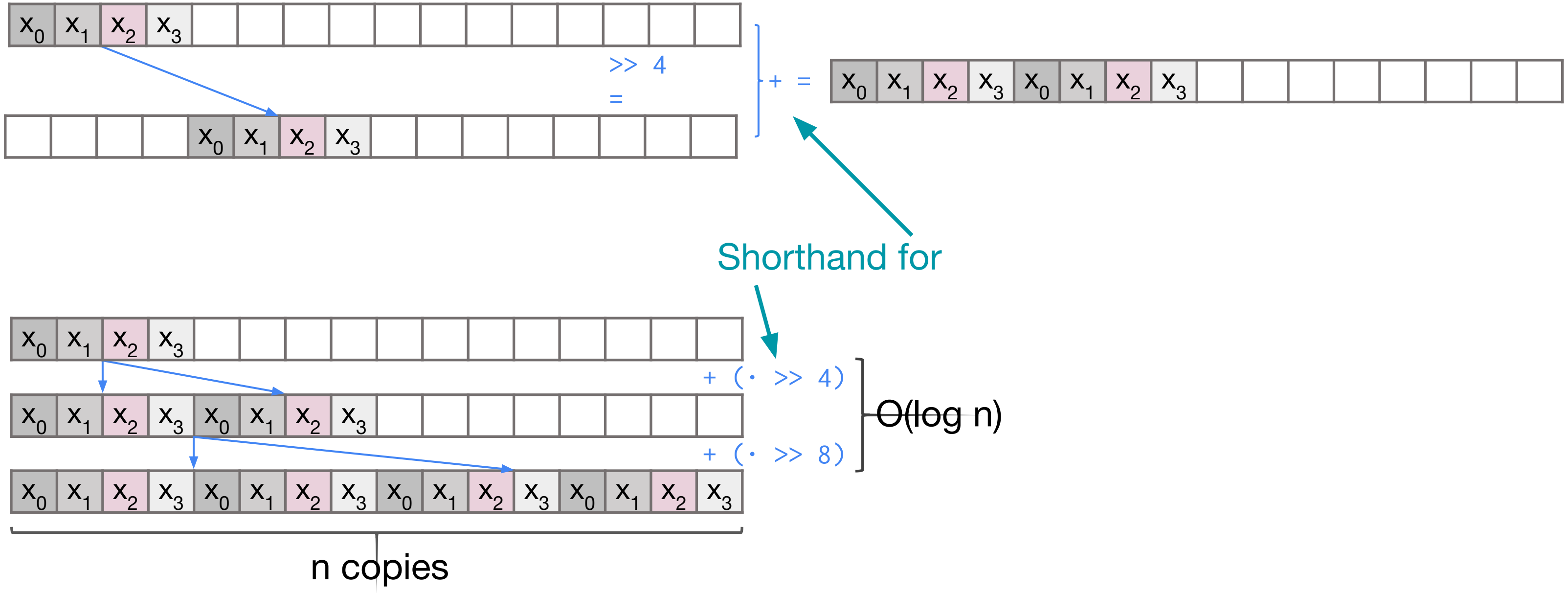


Ciphertexts

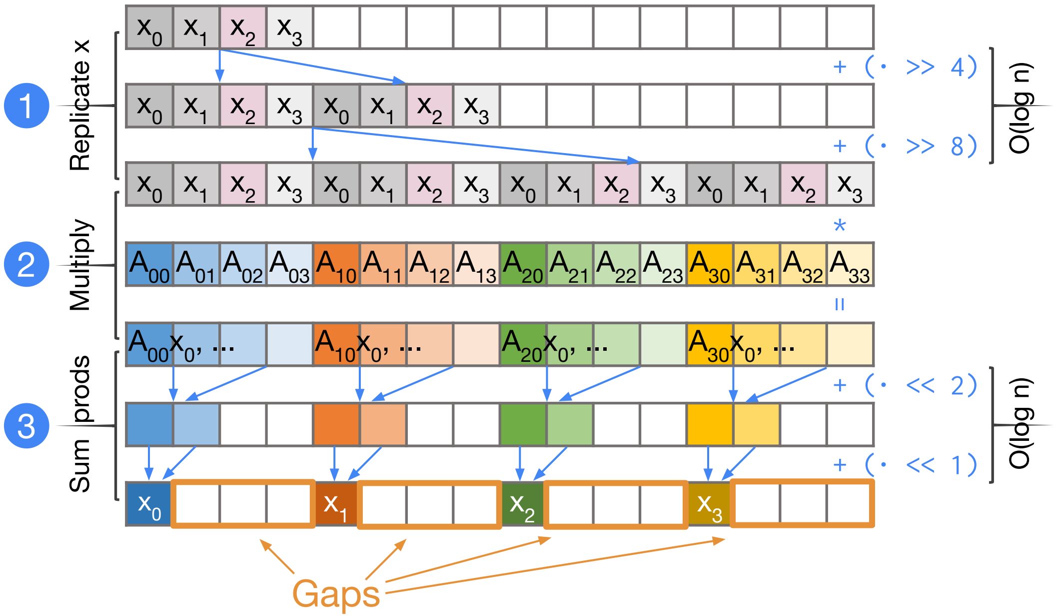


Uses only 1/128 of slots!

Replicate to Enable Data Parallelism

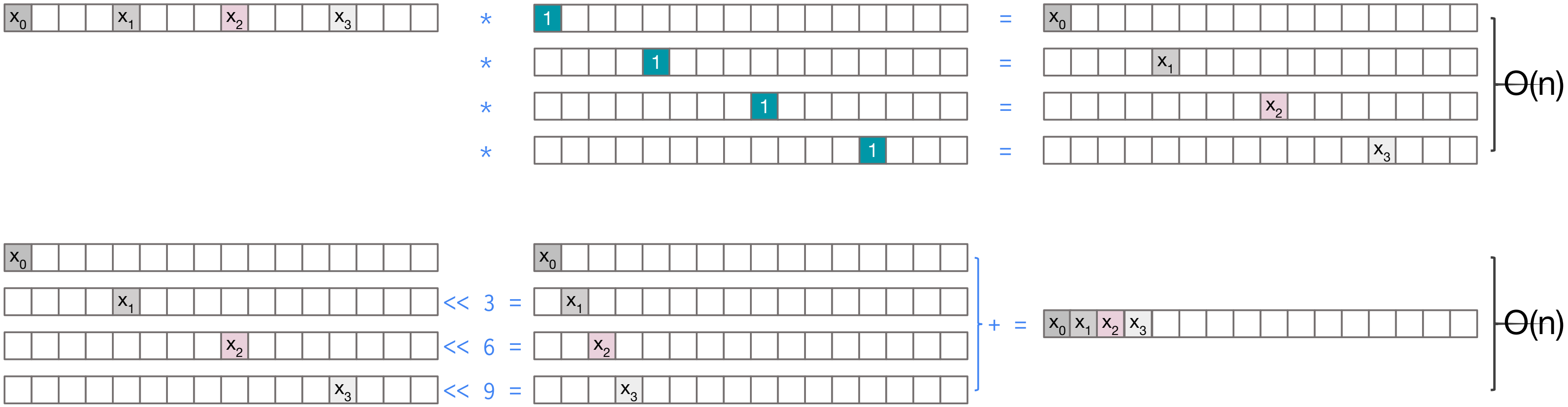


Efficient Matrix-Vector Multiply



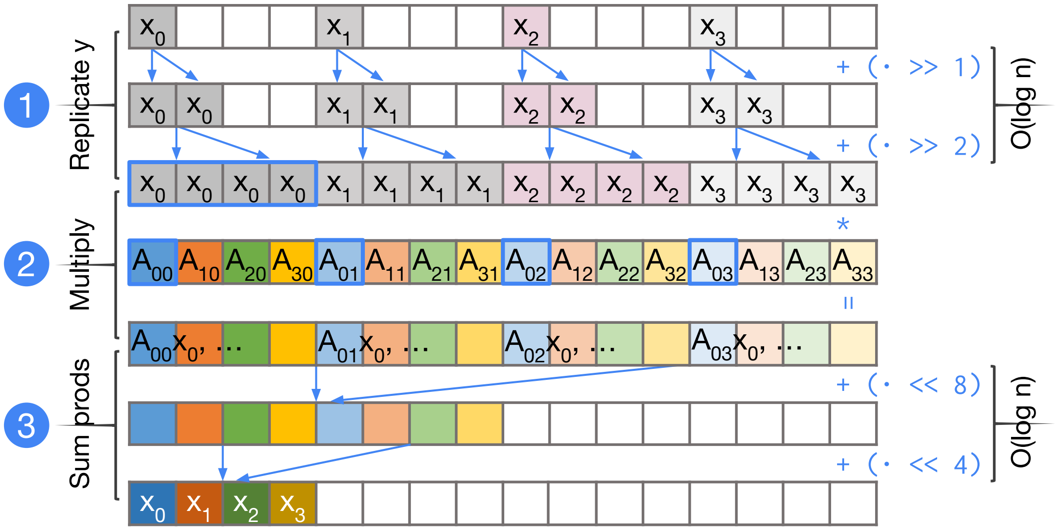
- Efficient: O(log n) ops for n dot products
- But, different output *layout*
- What can we do?

Convert Back to Original Layout?



O(n) ops!

Work with the Layout As-Is?



- Efficient!
- Manually stitching layouts: tedious and error-prone
  - Many more layouts for different matrix sizes
- We want a compiler

Fhelipe:
Tensor FHE Compiler

Fhelipe Language

- Python DSL
- Datatype: **`Tensor`**
  - For FHE data parallelism

Tensor Operation	Description
`t + u`	Elementwise add
`t * u`	Elementwise multiply
`t.rotate(dim: int, by: int)`	Cyclic shift along `dim`
`t.replicate(dim: int, n: int)`	Copy tensor `n` times, forming a new dimension `dim`
`t.sum(dim: int)`	Sum along dimension `dim`, discarding it
`t.stride(dim: int, by: int)`	Discard indices `i ≡ 0` (mod `by`); `by` must be a power of 2
`t.extend(dim: int, size: int)`	Zero-pad `dim` up to `size`
`t.shrink(dim: int, size: int)`	Shrink `dim` down to `size`
`t.drop_dim(dim: int)`	Discard a dimension of size 1
`t.insert_dim(dim: int)`	Insert a dimension of size 1
`t.reorder_dim(p: List[int])`	Permute dimensions (e.g., transpose)

Fhelipe Operations Compose!

- Functions: write once and reuse
- Enabled by automatic layouts and noise management

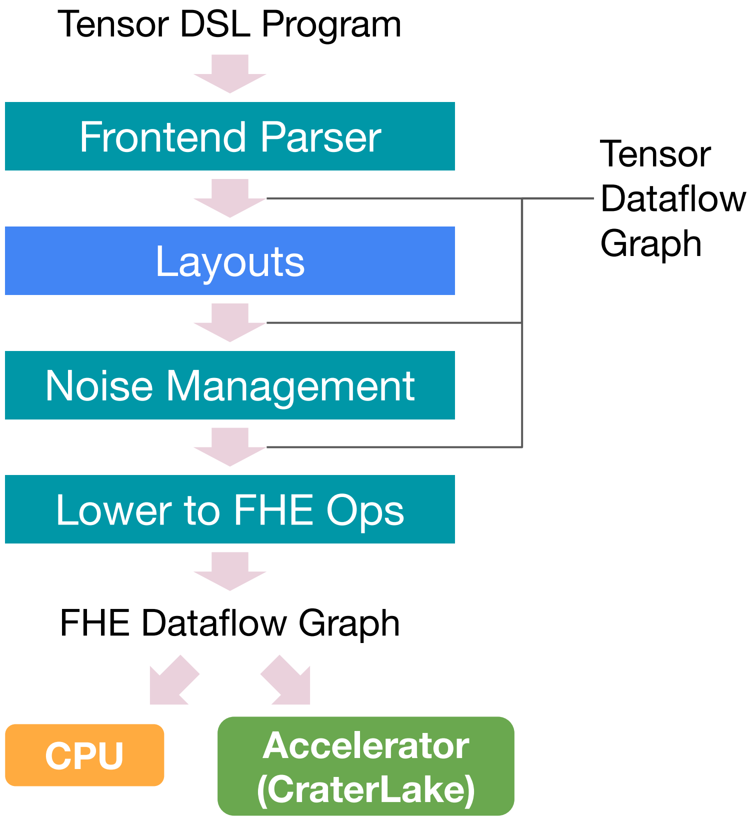
```python
def mv_mul(m: Tensor, v: Tensor) -> Tensor:
```

```python
def approx_tanh(x: Tensor) -> Tensor:
    c_1 = 0.249476365628036
    c_3 = -0.00163574303018748
    return c_1 * x + (c_3 * x) * (x * x)
```

Operations Used:

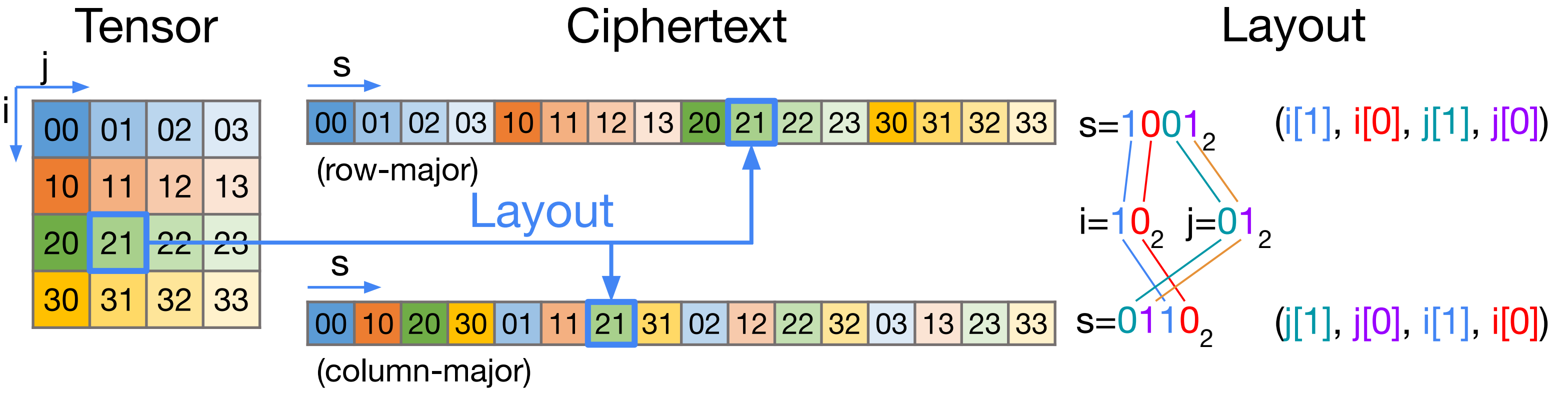
Tensor Operation	Description
`t.replicate(dim: int, n: int)`	Copy tensor `n` times, forming a new dimension `dim`
`t * u`	Elementwise multiply
`t.sum(dim: int)`	Sum along dimension `dim`, discarding it

Compilation Flow

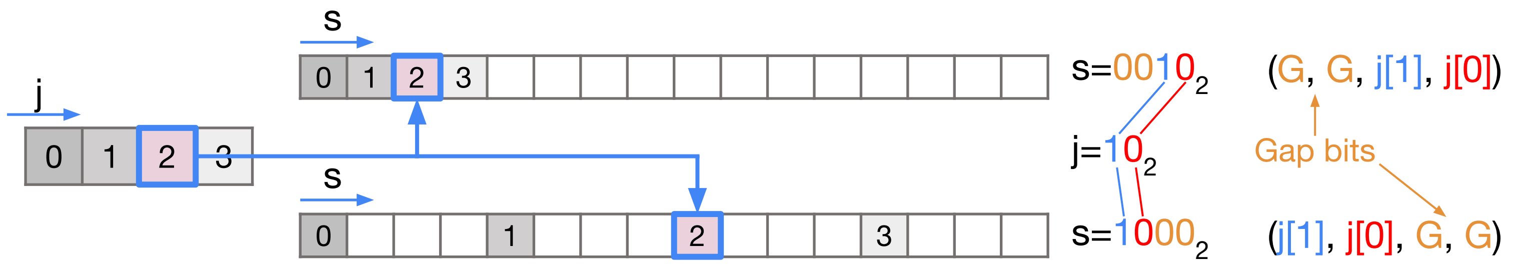


Fhelipe Layouts

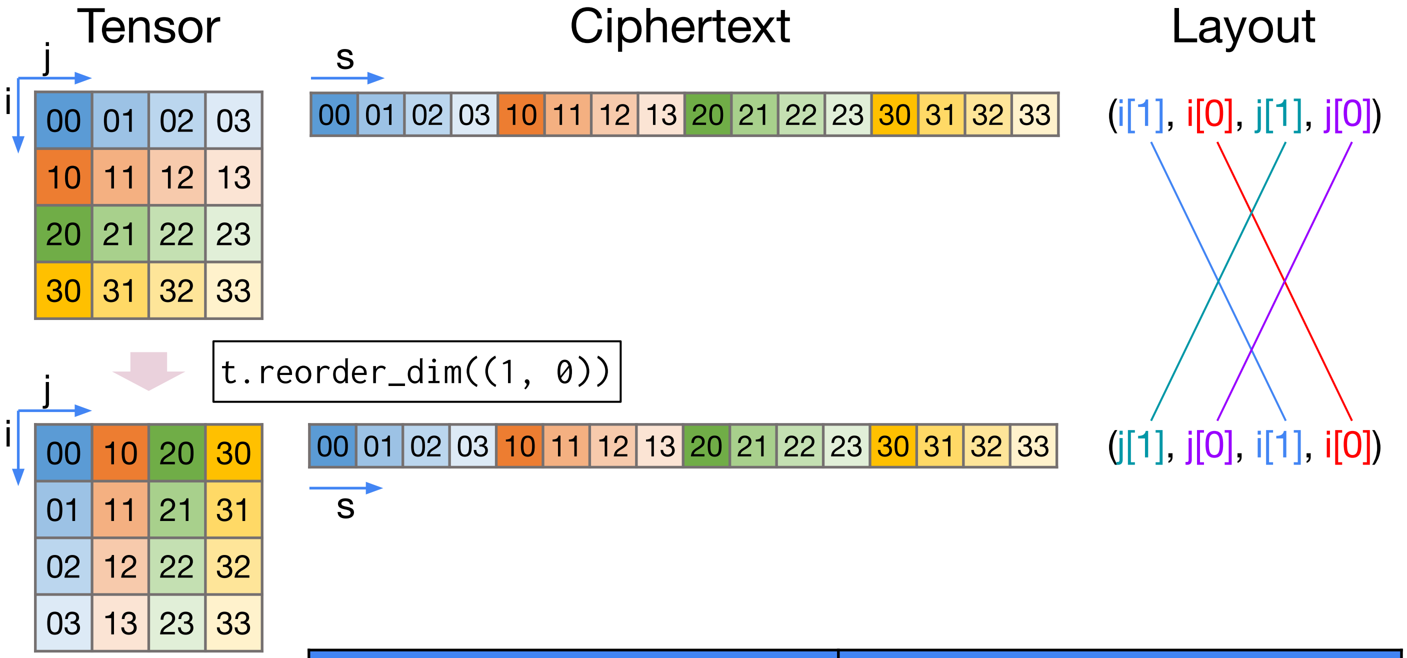
Fhelipe Layouts



Fhelipe Layout: permutation of dimension *index bits* with interleaved *gap bits*

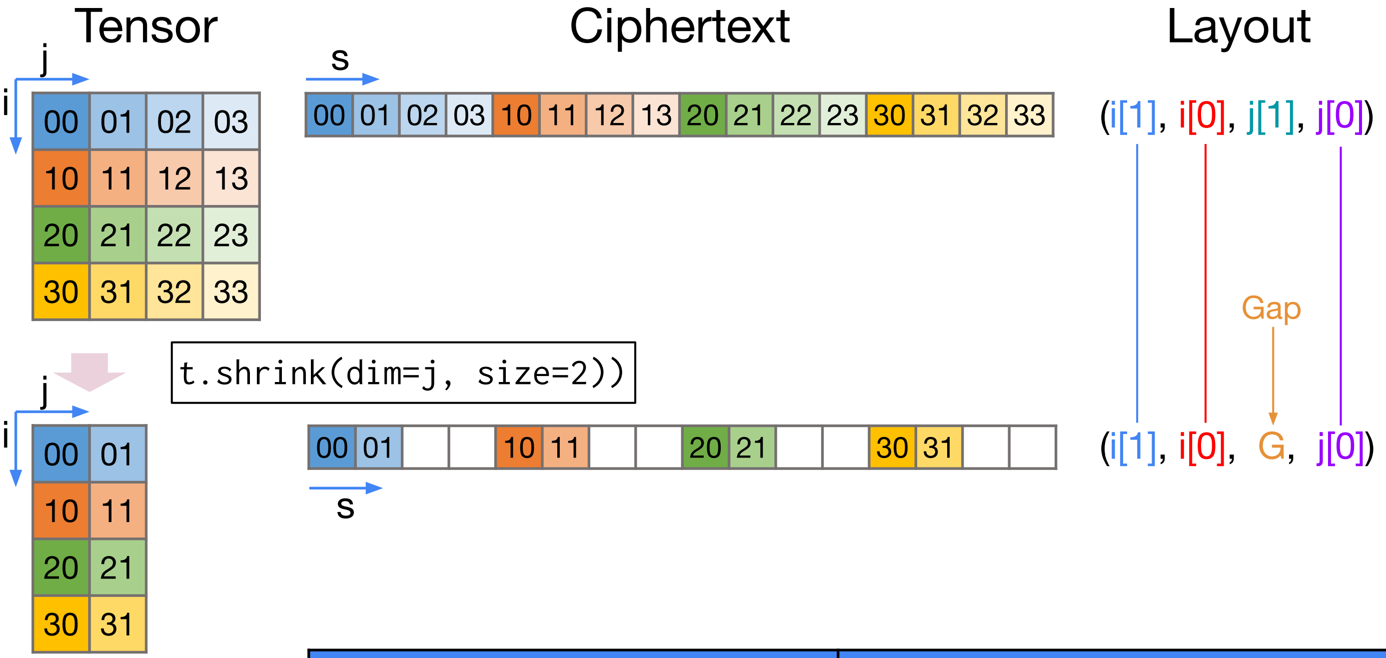


Layouts: Transpose



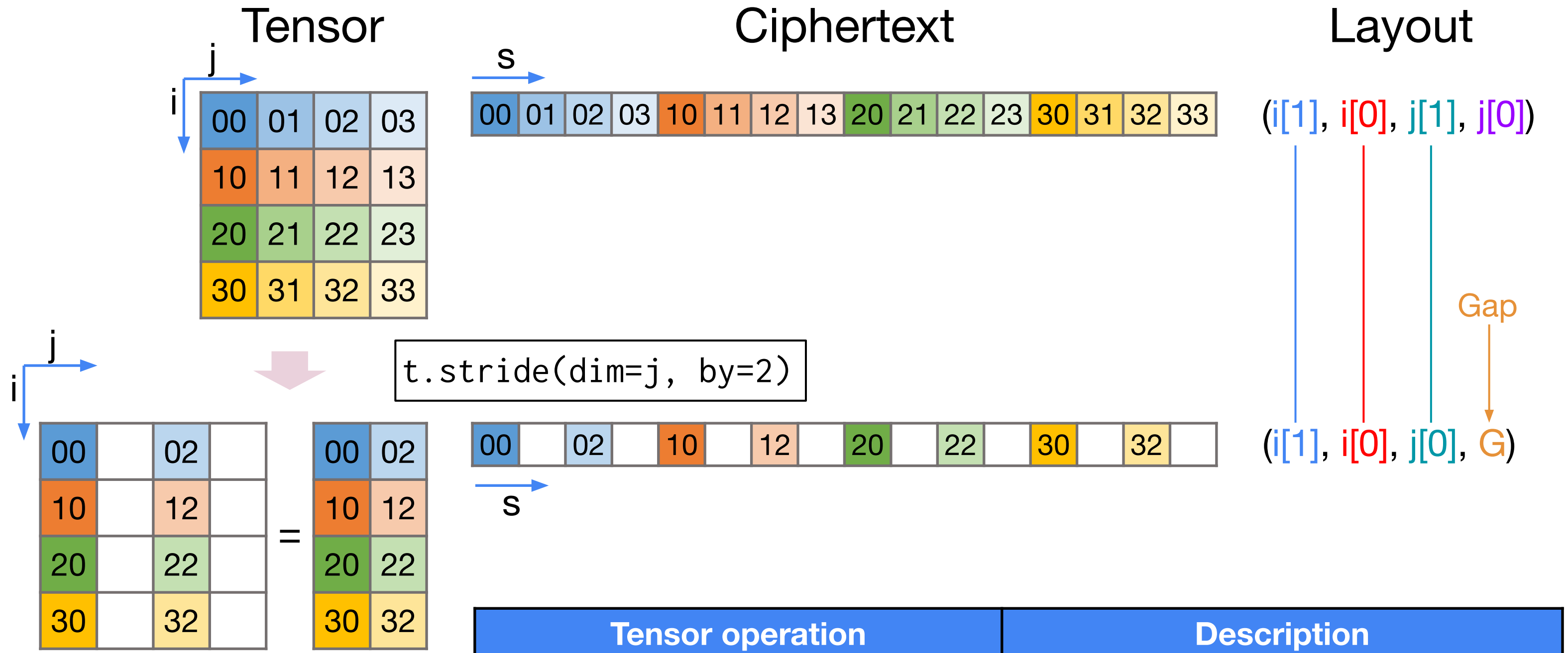
Tensor operation	Description
`t.reorder_dim(p: List[int])`	Permute dimensions (e.g., transpose)

Layouts: Shrink



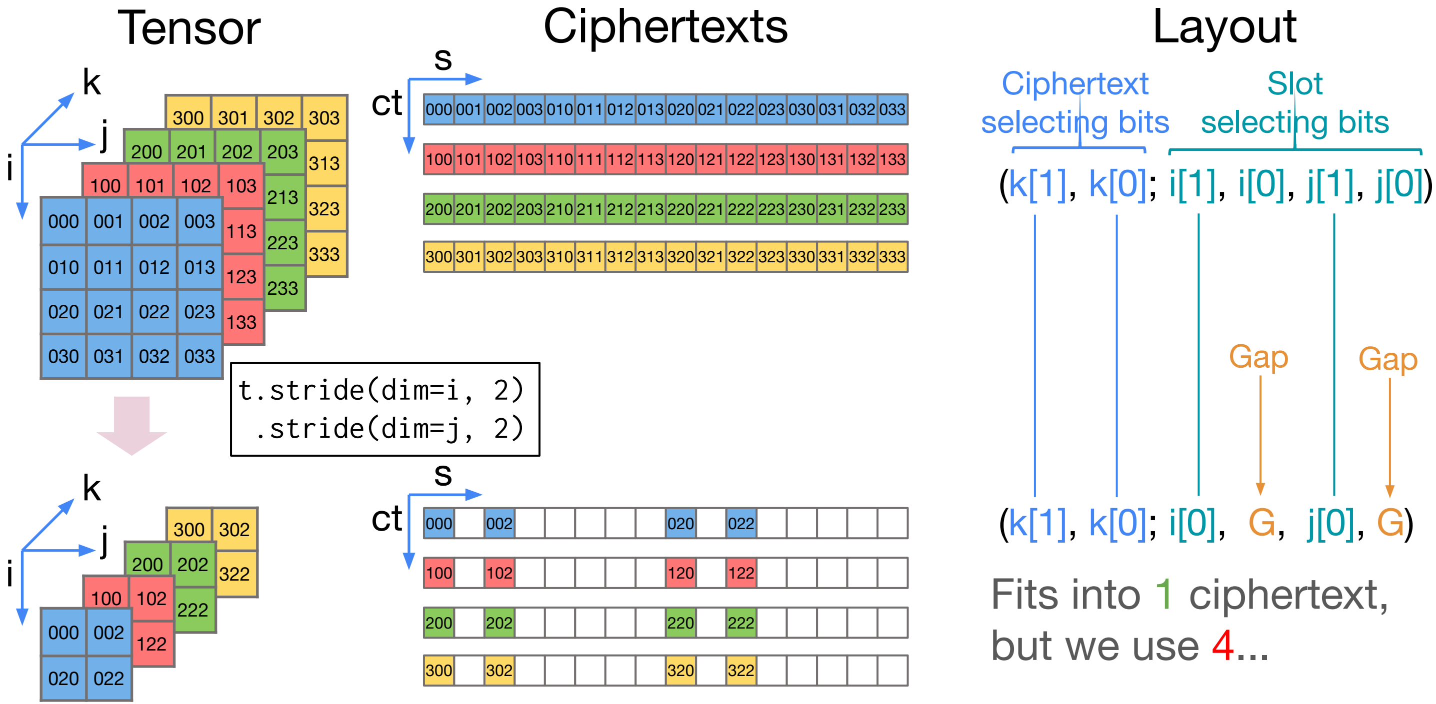
Tensor operation	Description
`t.shrink(dim: int, size: int)`	Shrink `dim` down to `size`

Layouts: Stride



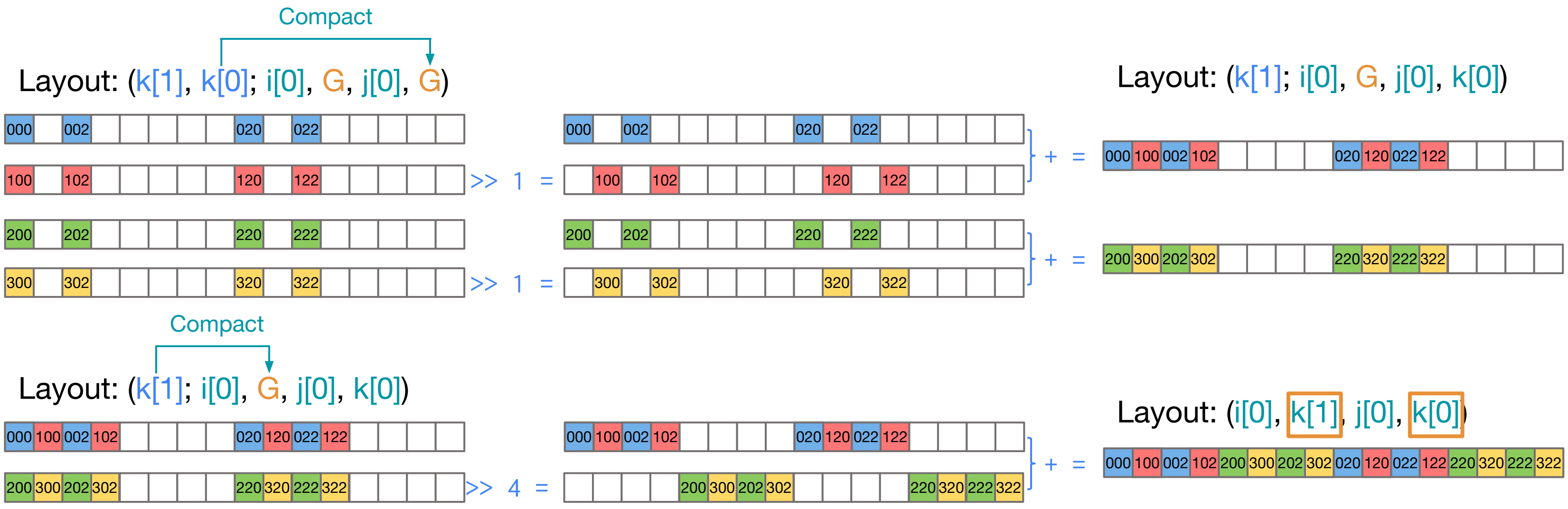
Tensor operation	Description
`t.stride(dim: int, by: int)`	Discard indices `i ≡ 0` (mod `by`); `by` must be a power of 2

Layouts: Large Tensors



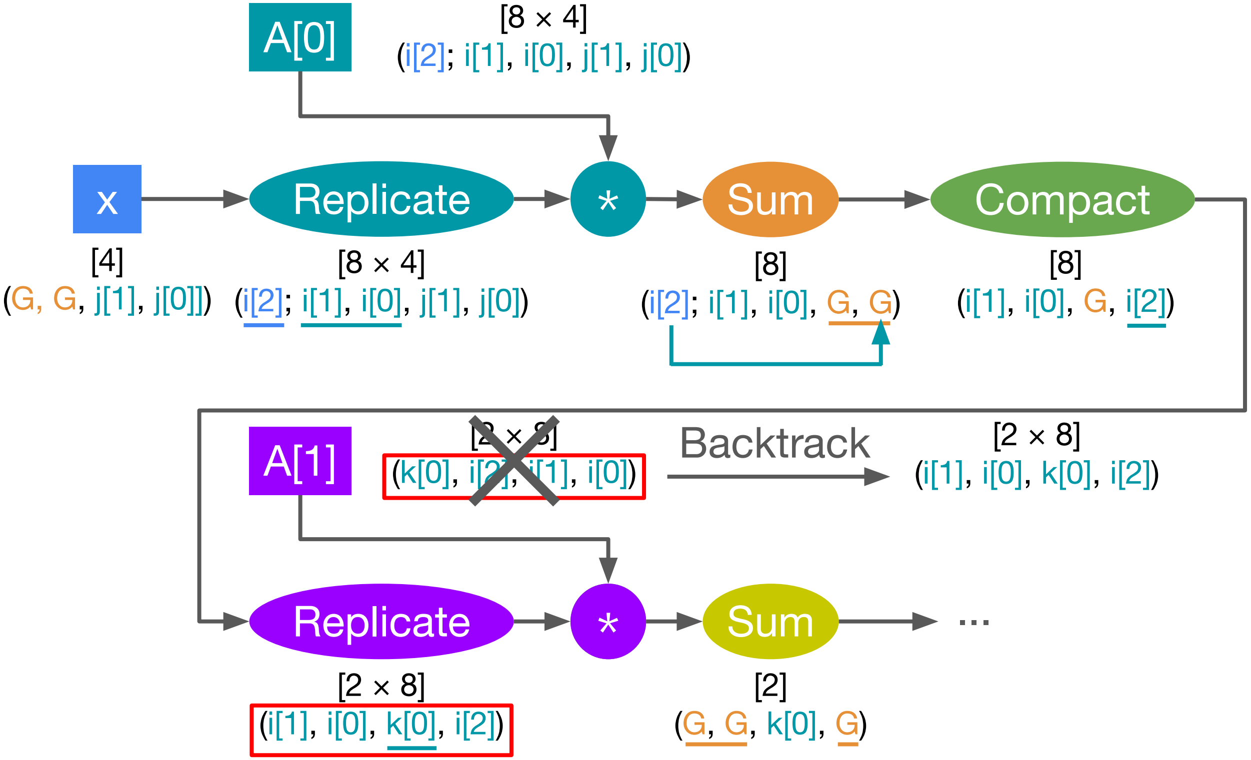
Fits into 1 ciphertext, but we use 4...

Layouts: Compaction



- Flexible layouts ⇒ compaction:
  - Cheap: 1 rotate-add / ciphertext
  - Utilizes all slots
- Prior compilers ⇒ no compaction
  - Remove gaps ⇒ expensive [CHET, HECO]
  - Leave gaps ⇒ unused slots [HeLayers]

Layout Assignment




```
for A_i in A:
  x = mv_mul(A_i, x)
return x
```

- Inputs: row-major
- Greedily fill gaps
- Match layouts on binary operations
- Compact eagerly

Mismatch!

- Backtrack
- Add conversion (if necessary)

Evaluation

				Runtime CraterLake [ms]			Speedup vs	
Application	Ops	LoC	Compile	Fhelipe	Manual	CHET+	Manual	CHET+
ResNet-20	120M	100	14.7 s	235.8	236.1	526.4	1.0×	2.2×
RNN	13M	80	1.6 s	434.7	452.4	2,223	1.0×	5.1×
LogReg	77M	60	27.3 s	141.5	1,741	4,592	12.3×	32.5×
LoLa-MNIST	1M	50	0.8 s	0.3	0.9	90.1	3.2×	322.4×
						gmean	**2.5×**	**18.5×**

Summary

- Easy-to-use tensor FHE language
- Automates layouts and noise management
  - Enables reuse and composition
- Great performance
  - First to match state-of-the-art manual implementations