EPIC EIC Architecture

JAX Blocks

Overview

Rationale behind the design

- The approach is to map a known architecture that gives reasonably good accuracy (> 95%) on MNIST on to the EIC architecture constraints. Which currently is a dense network: 784 - 2048 - 1024 - 10. We need to truncate it to 784 - 2048 - 256 - 10 for given the EIC constraints!

- The idea is to distribute dense layers across multiple cores. Three challenges with this are:

    - [`EICDense`] Every neuron in a core has a fixed in-degree of 256.

    - [`Accumulator`] Every core applies the non linearity so accumulation over contracting dimension isn't straightforward.

    - [`ShuffleBlock`] There are only positive weights and no biases. Furthermore, we have some mathematical constraints on how inputs are presented to cores ("balanced inputs").

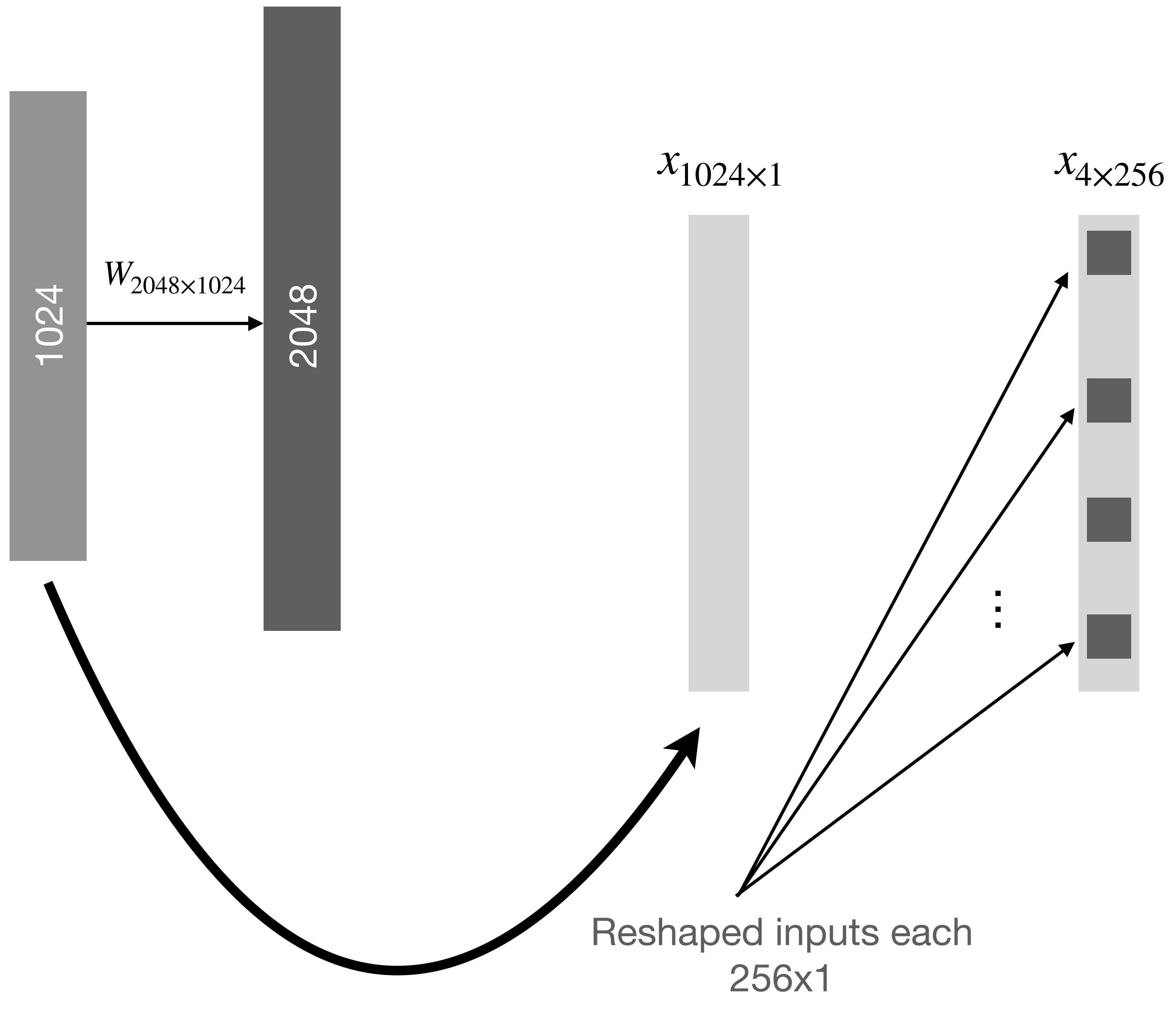
- To tackle the first issue `EICDense` block implements tensor reshaping for the positive weights of the layer such that at a time a 256-element wide vector can only multiply with a 256x256 core. This multiplication is followed by non-linearity. This block does NOT accumulate the results as a straightforward accumulation along the contracting dimension (i.e. the dimension of the input layer in a dense network) is incorrect due to the non-linearity.

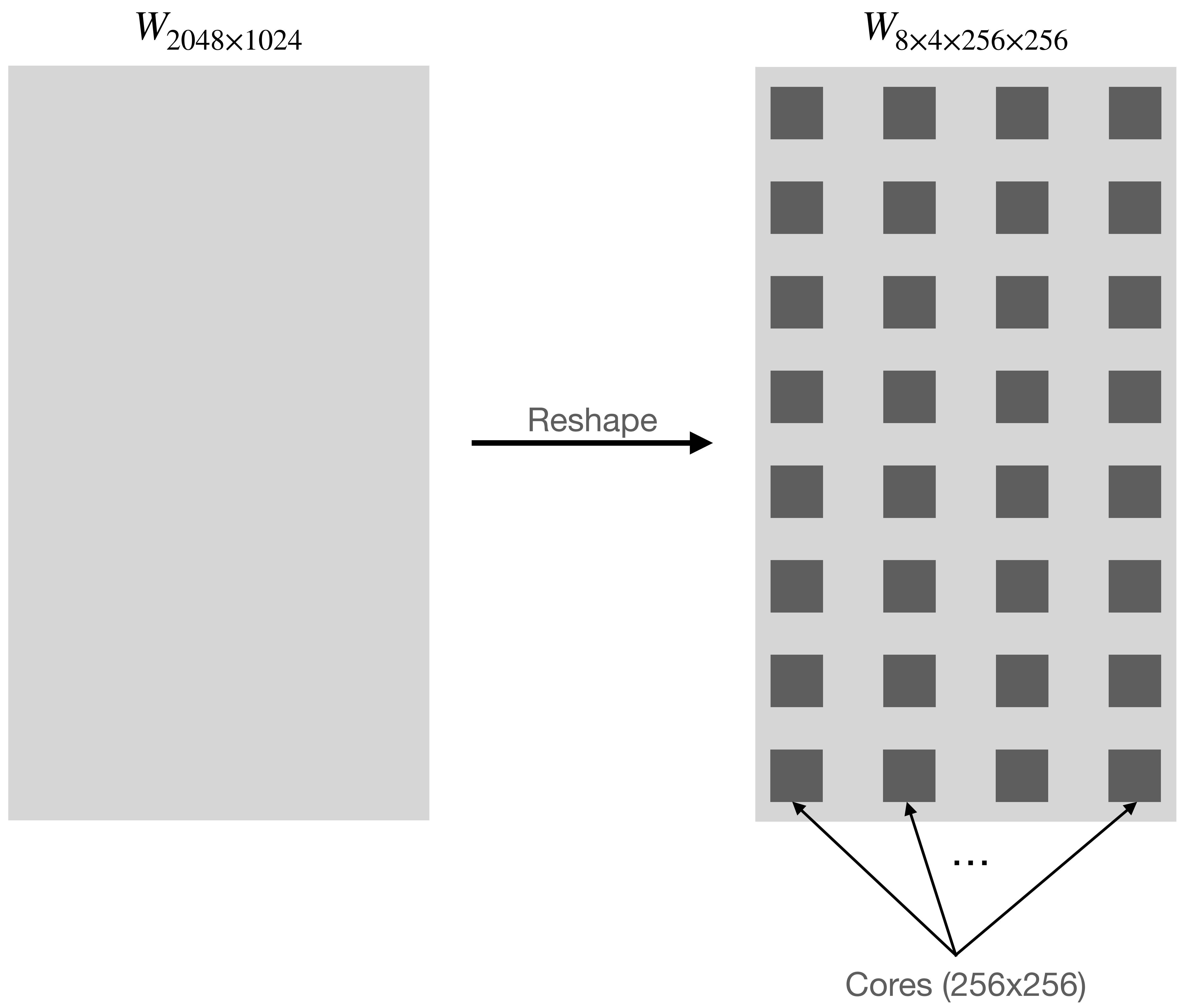
- The `Accumulator` block then defines additional cores, typically this number is `out_size`//256, that contain learnable positive weights. These cores accumulate the results of `EICDense` block into a tensor of `out_shape`. Since the weights are learnable, the idea is that gradient descent would provide us with the best way to accumulate the outputs of `EICDense` block (this problem is exclusively because the non-linearity is applied after every core-multiplication operation). In terms of hardware, this accumulation can be done in a time multiplexed fashion.

- We do not have negative weights, nor biases in EIC circuit, so a curious question arises as to where can we obtain the negative signs from. One solution for this is to split the input registers at the cores into negative and positive subgroups and accumulate the results separately followed by differencing the two accumulations. Mathematically, this operation is $y = f(W(x^+ - x^-))$. The `ShuffleBlock` ensures this at a very abstract level by defining a permutation matrix. There is also the constraint of "balanced" inputs i.e. a given input in the positive slot register at a core must also be present in some negative slot of the same core. This isn't **straightforward to implement in Jax**! For this reason we use permutation matrix to shuffle the `Accumulator` output before feeding it into the subsequent cores. This block is still work in progress and will likely be modified once we get clear directives from the simulator folks about the details of LUTs. However, in a broad sense this block will allow us to also suggest how inter-core connectivity should look like since the permutation matrix is also set up as learnable!

EICDense

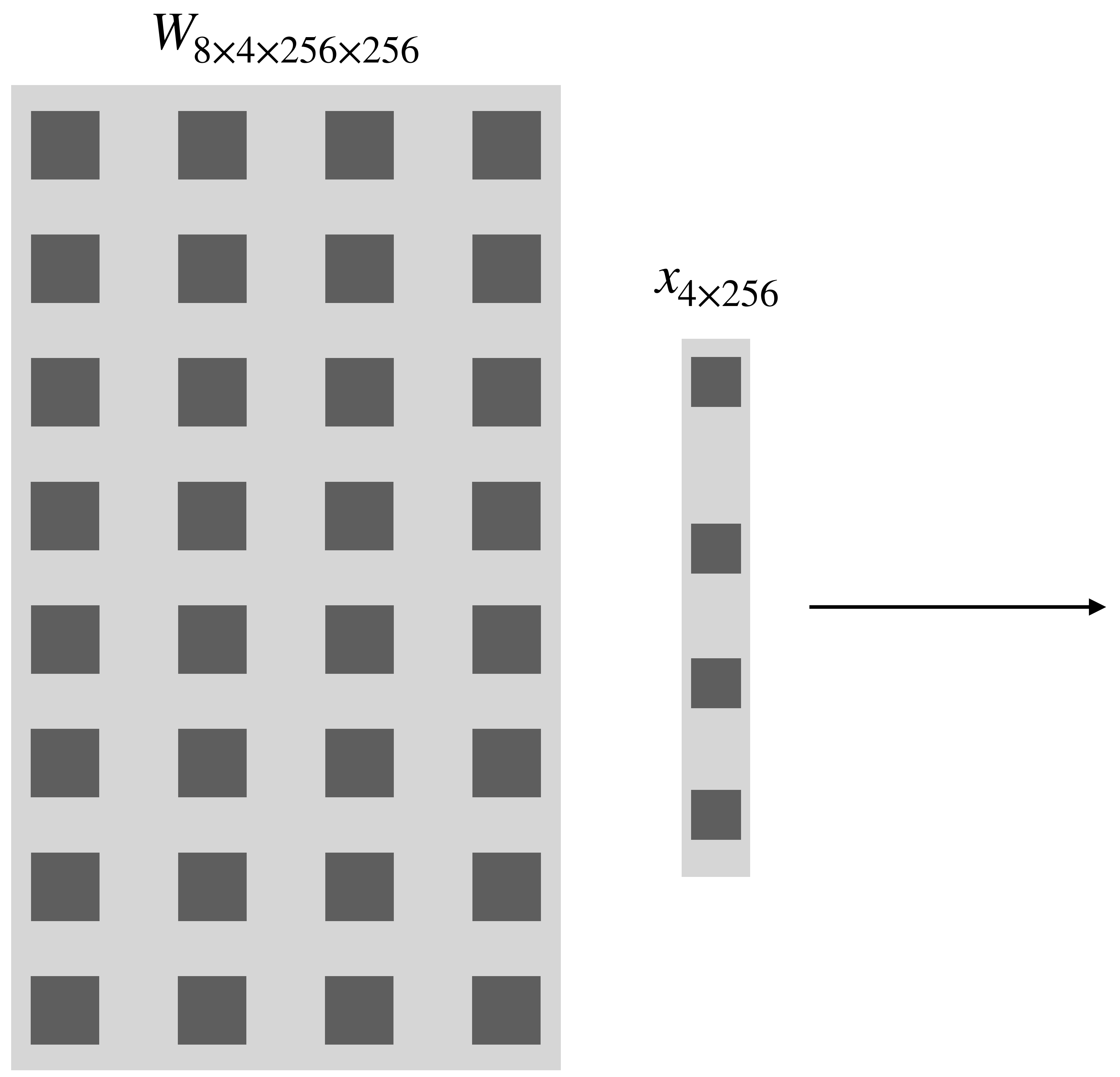
EIC hardware-constrained matrix-vector product





EICDense

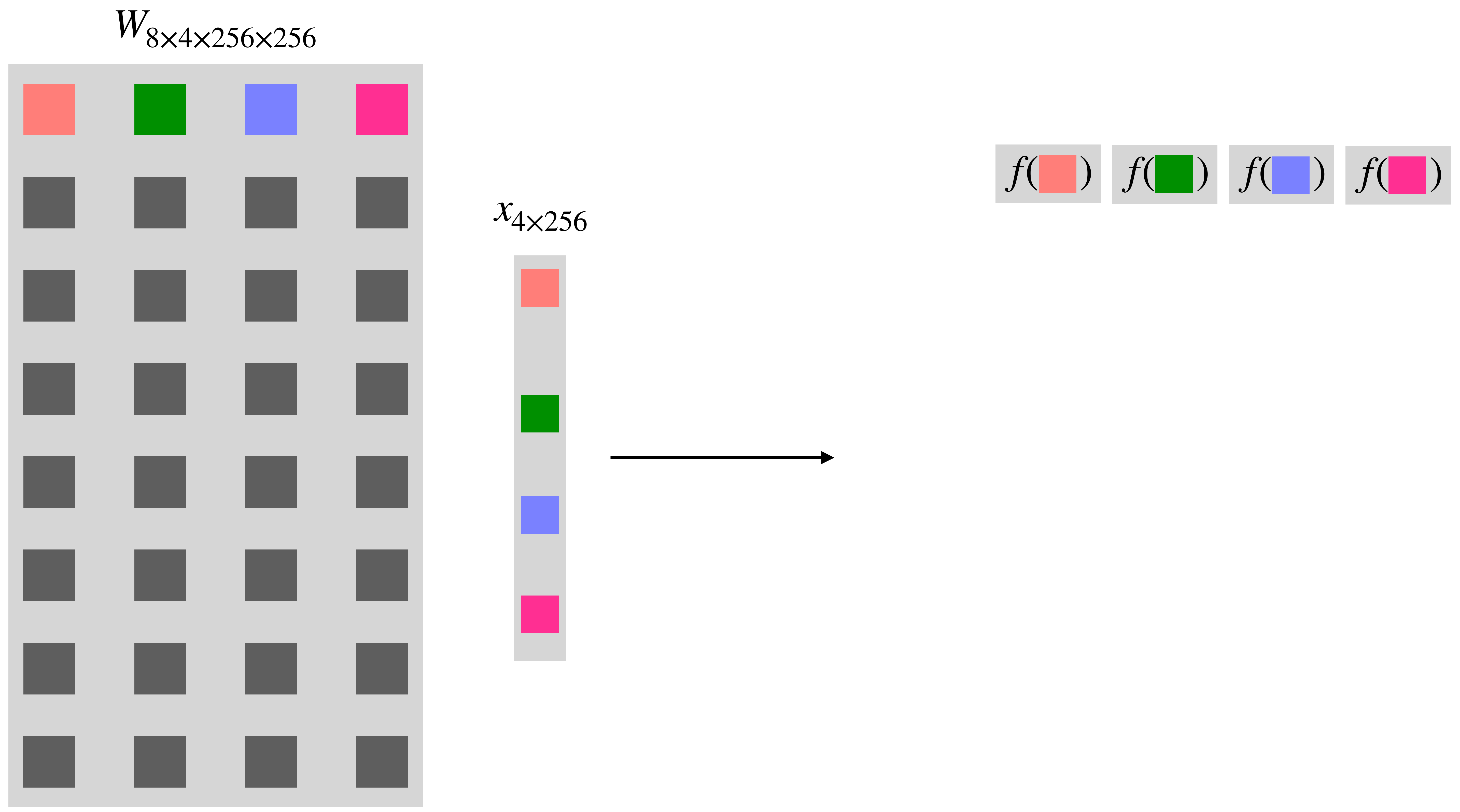
EIC hardware-constrained matrix-vector product



$f(.)$ Non-linear function

EICDense

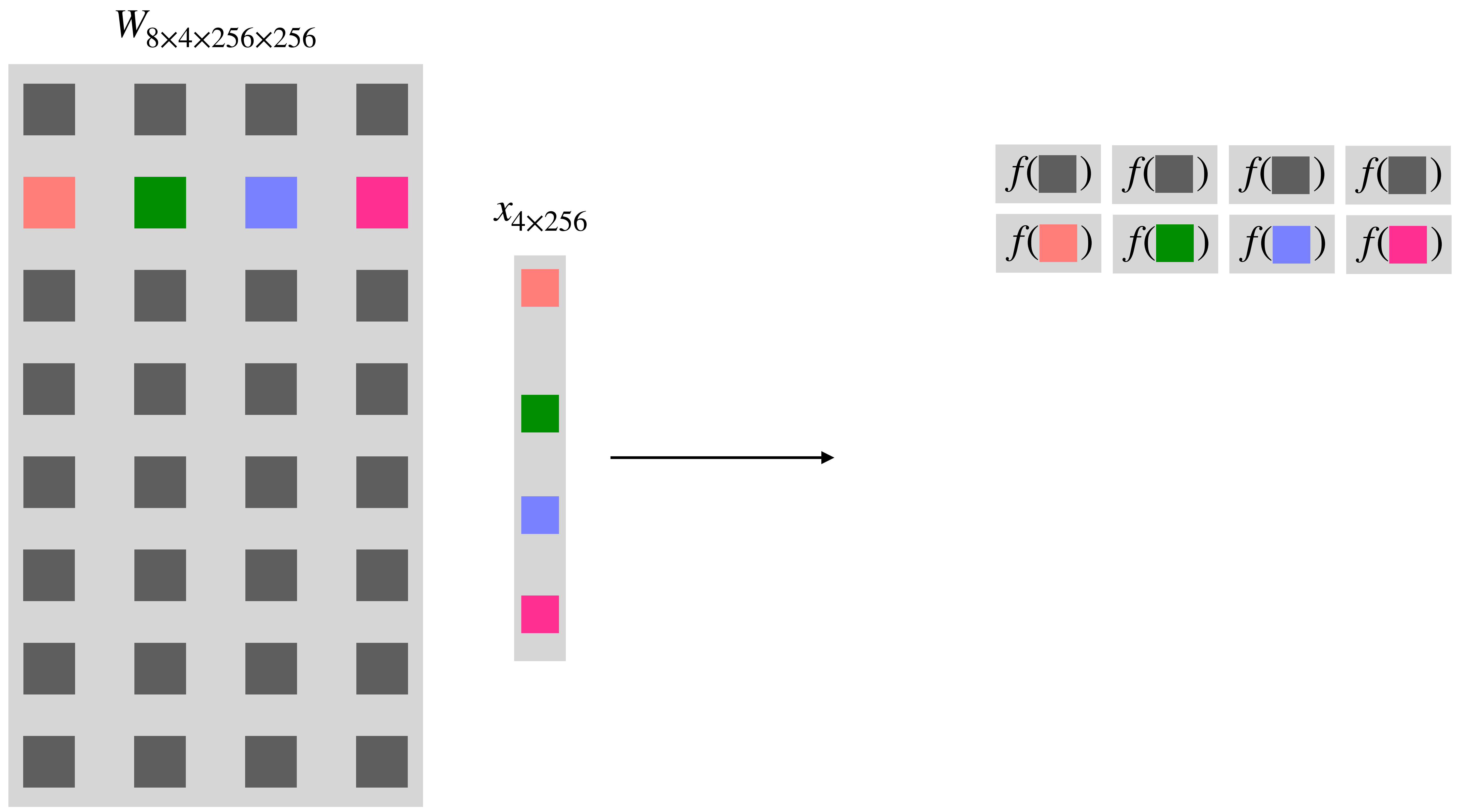
EIC hardware-constrained matrix-vector product


$W_{8\times4\times256\times256}$
$x_{4\times256}$
$f($ $)$ $f($ $)$ $f($ $)$ $f($ $)$

$f(.)$ Non-linear function

EICDense

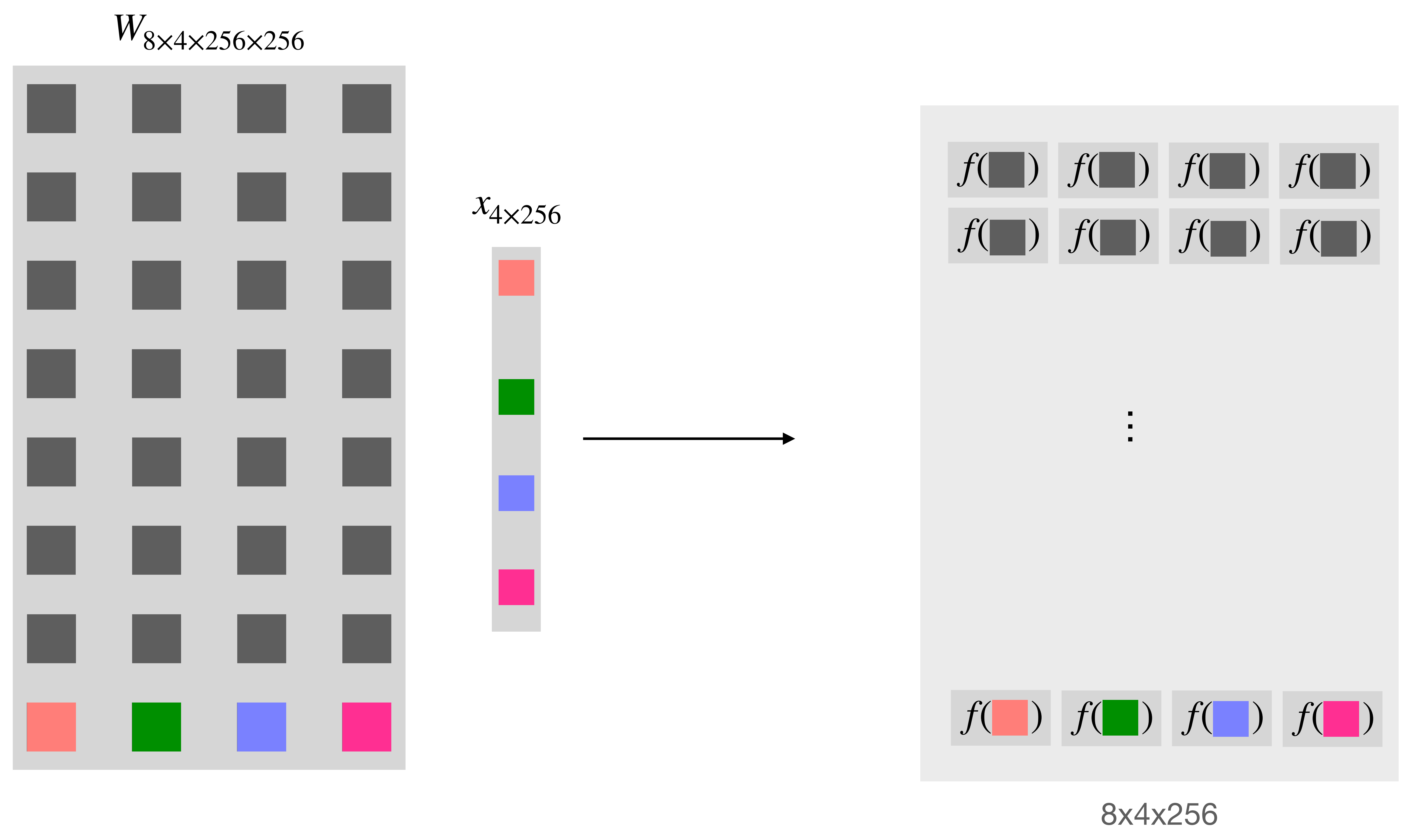
EIC hardware-constrained matrix-vector product



$f(.)$ Non-linear function

EICDense

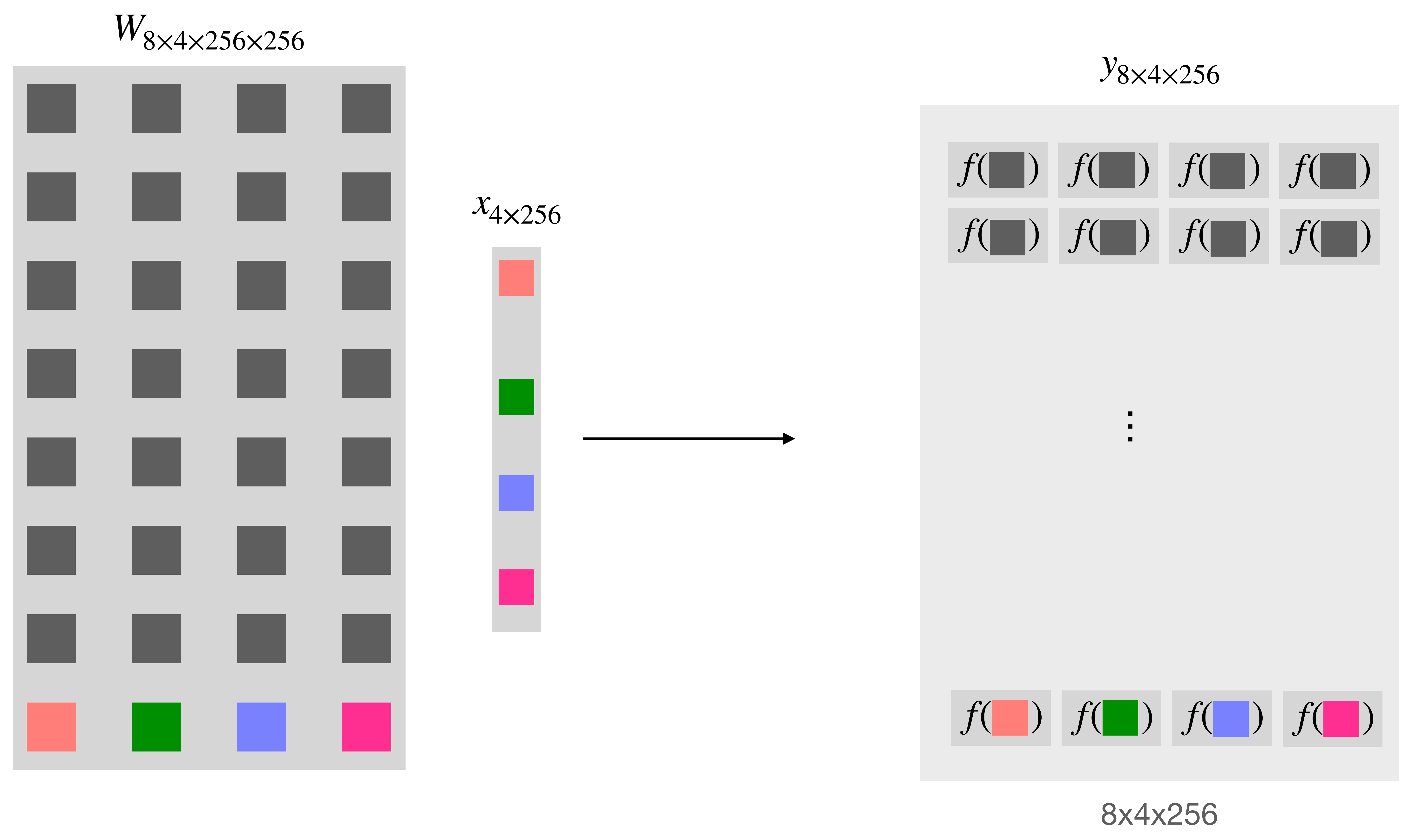
EIC hardware-constrained matrix-vector product



$f(\,.\,)$ Non-linear function

EICDense

EIC hardware-constrained matrix-vector product



Using `einsum`

```
y = einsum('ijkl,jl->ijk', W, x)

y = f(y)
```

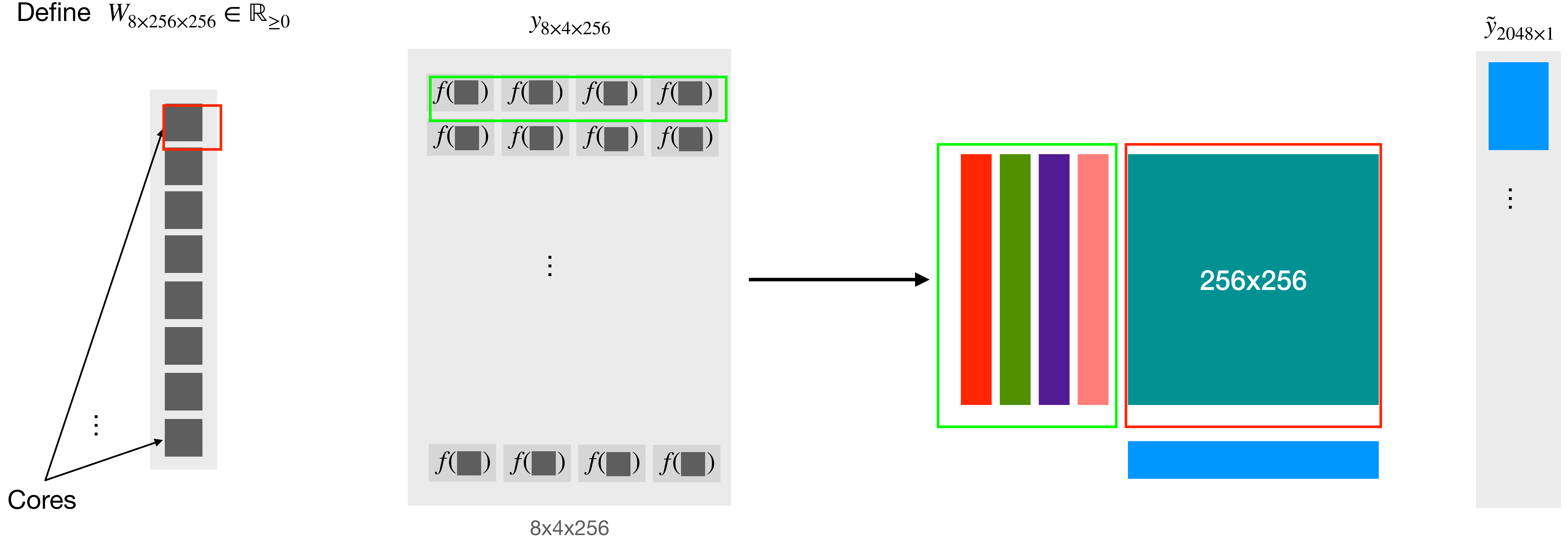
$f(\cdot)$ Non-linear function

But how to put these blocks together, as we can't implement exact dense layers?

Accumulator

Learnable accumulation of outputs

Define $W_{8\times256\times256} \in \mathbb{R}_{\geq 0}$



```
y_tilde = einsum('ijk,imk -> ik', W, x)

y_tilde = flatten(y_tilde)
```

ShuffleBlock

Proxy for LUTs

There are two functional aspects to consider:

1. How to keep “balanced” inputs in the input registers?

2. What’s the best connectivity for the input slots?

Current Solution

Define a permutation matrix P for every layer, such that

$$x^{(n+1)} = P\tilde{y}^{(n)}$$

Where,

$$P = \text{softmax}(Z/\tau)$$

$$Z \sim \mathcal{N}^{N\times N}(0,1)$$

$$\tau \in (0,1]$$ (Temperature parameter)

Note: This is still work in progress, any improvements/better ideas are welcome!

Nikolentzos et al. aRxiv (2021), *Permute Me Softly: Learning Soft Permutations for Graph Representations*