A APPENDIX

A.1 Array Index Notation Grammar

The full syntax of array index notation can be found in Figure 32.

$$
\begin{aligned}
\langle \mathit{array\_stmt} \rangle &::= \langle \mathit{access} \rangle \text{ `='' } \langle \mathit{expr} \rangle \\
\langle \mathit{access} \rangle &::= \langle \mathit{tensor} \rangle_{\{\langle \mathit{index} \rangle\}} \\
\langle \mathit{index} \rangle &::= \langle \mathit{index\_var} \rangle \; [\; \langle \mathit{index\_slice} \rangle \;] \\
\langle \mathit{index\_slice} \rangle &::= \text{`('} \; \langle \mathit{lo} \rangle \; \text{`:'} \; \langle \mathit{hi} \rangle \; [\; \text{`:'} \; \langle \mathit{st} \rangle \;] \; \text{`)'} \\
\langle \mathit{expr} \rangle &::= \langle \mathit{literal} \rangle \mid \langle \mathit{access} \rangle \mid \langle \mathit{call\_expr} \rangle \mid \langle \mathit{reduce\_expr} \rangle \\
&\quad\mid \; \langle \mathit{binary\_expr} \rangle \mid \langle \mathit{unary\_expr} \rangle \mid \text{`('} \; \langle \mathit{expr} \rangle \; \text{`)'} \\
\langle \mathit{call\_expr} \rangle &::= \langle \mathit{func} \rangle \; \text{`('} \; \langle \mathit{expr} \rangle \; \{ \text{`,'} \; \langle \mathit{expr} \rangle \} \; \text{`)'} \\
\langle \mathit{reduce\_expr} \rangle &::= \underset{\langle \mathit{index\_var} \rangle}{\langle \mathit{func} \rangle} \; \langle \mathit{expr} \rangle \\
\langle \mathit{binary\_expr} \rangle &::= \langle \mathit{expr} \rangle \; \langle \mathit{op} \rangle \; \langle \mathit{expr} \rangle
\end{aligned}
$$

Fig. 32. The syntax of array index notation. Expressions within braces may be repeated any number of times. $\langle \mathit{func} \rangle$ and $\langle \mathit{op} \rangle$ both represent arbitrary (user-defined or predefined) functions and are implemented in the same way; they differ only in how they are invoked.

A.2 PyData/Sparse API

An example of performing the xor operation on two sparse tensors using PyData/Sparse is found below.

```
import numpy
import sparse

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = numpy.logical_xor(A, B)
```

An example performing the GCD operation can be found below:

```
import numpy
import

def gcd(x, y):
  return ... # Compute the GCD between x and y.
# Register the gcd function as a ufunc.
gcd = np.frompyfunc(gcd, 2, 1)

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = gcd(A, B)
```

While this code is simpler than the code to use our sparse array compiler, users do not have control over many factors, such as the formats of the tensors, and are restricted to the predefined set of NumPy functions.

A.3 Iteration Lattice Construction Algorithm

As described in Section 6, the presented iteration lattice construction algorithm (Algorithm 1) supports only array index notation expressions that do not contain repeat tensors. Fig. 18 illustrates an example of when iteration sub-spaces do not overlap when the index notation contains a repeated tensor. This example motivates our implementation of a filtered Cartesian Product.

We include the full algorithm that does support repeated tensors in Algorithm 2.

Algorithm 2 Full iteration lattice construction algorithm

procedure ConstructLattice (FunctionAlgebra A, FunctionArguments args)
  // Preprocessing steps
  Algebra A = DeMorgan(A) ▷ Apply De Morgan's Law
  Algebra A = Augment(A, args) ▷ Augmentation pass
  return BuildLattice(A)
end procedure

// **let** $\mathcal{L}$ represent an iteration lattice and p represent an iteration lattice point
procedure BuildLattice (Algebra A)
  if A is Tensor(t) **then** ▷ Segment Rule
    return $\mathcal{L}(p(\{ t \}$, producer=true))
  else if A is ~Tensor(t) **then** ▷ Complement Rule
    $p_o = p(\{ t, \mathbb{U} \}$, producer=false)
    $p_p = p(\{ \mathbb{U} \}$, producer=true)
    return $\mathcal{L}(\{ p_o, p_p \})$
  else if A is (left ∩ right) **then** ▷ Intersection Rule
    $\mathcal{L}_l$, $\mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
    cp = FilteredCartesianProduct($\mathcal{L}_l$.points(), $\mathcal{L}_r$.points())
    mergedPoints = { $p(\{ p_l + p_r \}$, producer=p_l.producer ∧ p_l.producer) : $\forall(p_l, p_r) \in$ cp }
    mergedPoints = RemoveDuplicates(mergedPoints, ommitterPrecedence)
    return $\mathcal{L}$(mergedPoints)
  else if A is (left ∪ right) **then** ▷ Union Rule
    $\mathcal{L}_l$, $\mathcal{L}_r$ = BuildLattice(left), BuildLattice(right)
    cp = FilteredCartesianProduct($\mathcal{L}_l$.points(), $\mathcal{L}_r$.points())
    mergedPoints = { $p(\{ p_l + p_r \}$, producer=p_l.producer ∨ p_l.producer) : $\forall(p_l, p_r) \in$ cp }
    mergedPoints = mergedPoints + $\mathcal{L}_l$.points() + $\mathcal{L}_r$.points()
    mergedPoints = RemoveDuplicates(mergedPoints, producerPrecedence)
    return $\mathcal{L}$(mergedPoints)
end procedure

procedure FilteredCartesianProduct (LatticePoints left, LatticePoints right)
  $p_{l,\text{root}}$, $p_{r,\text{root}}$ = left.root, right.root
  for (p_l in left) × (p_r in right) **do** overlap = true
    for tensor in p_l **do**
      if (tensor in $p_{r,\text{root}}$) ∧ (tensor not in p_l) **then** overlap = false
    end for
    for tensor in p_r **do**
      if (tensor in $p_{l,\text{root}}$) ∧ (tensor not in p_l) **then** overlap = false
    end for
    if overlap **then** cp += {(p_l, p_r)}
  end for
  return cp
end procedure

A.4 Medical Imaging Edge Detection

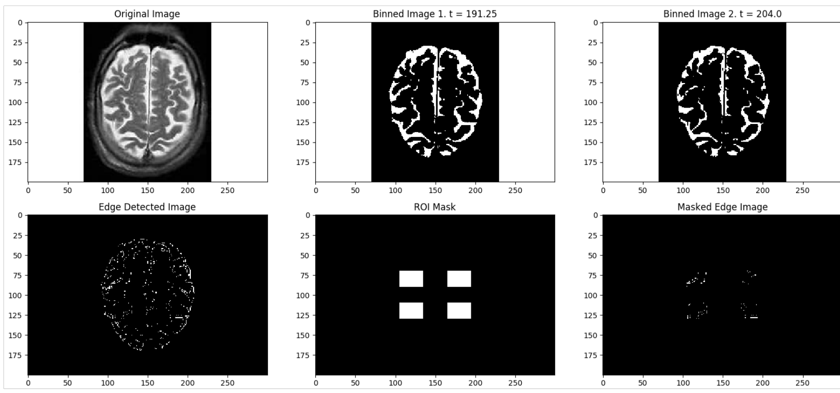

Fig. 33. Example MRI image, thresholding, ROI mask, and output