Recursive Storage Format of Sparse Tensors and Cache Efficient Tensor Compiler

Authors

October 7, 2017

1 Background - Common Tensor Storage Format

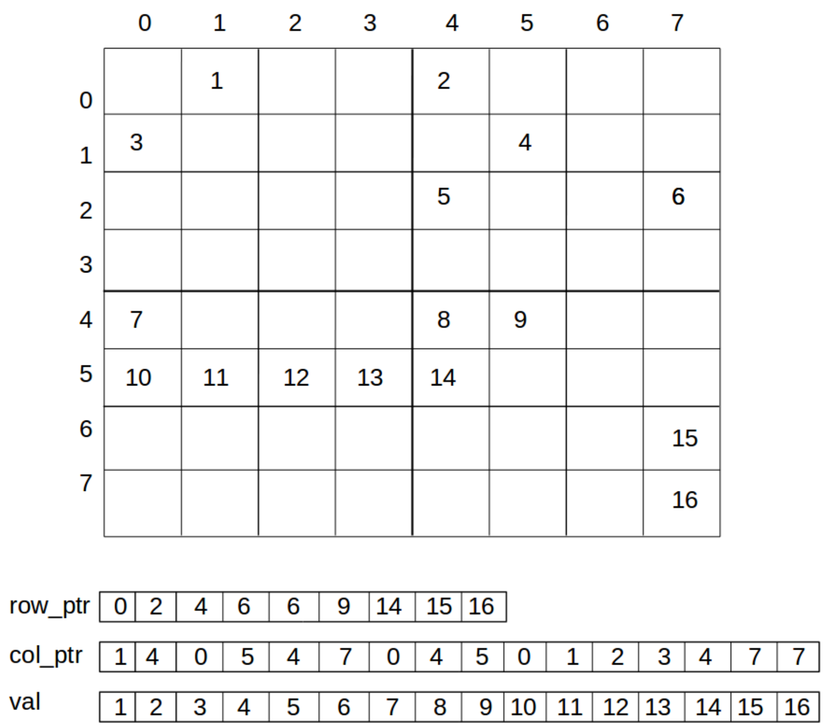

Figure 1: CSR Format

TACO presents a way to store higher dimensional sparse tensors. Our recursive storage format will work for all the formats supported by TACO.

2 Recursive Storage Format - Compressed Tree Storage(CTS)

In this section, we introduce a recursive storage format called Compressed Tree Storage(CTS) for higher dimensional sparse tensors. Our algorithm can convert any storage format to CTS if that storage format can be transformed to coordinate format(COO). The following algorithm CSR_To_CTS converts a CSR matrix M to CTS format.

Algorithm 1 CSR_To_CTS(M): M is a CSR matrix. Returns equivalent CTS matrix

M_{coo} = CSR_To_COO(M) // converts to COO format
base = top-left most indices of the minimal size orthogonal box X that contains all the elements.
len = length of a dimension of X.
M_{sp} = createSPTree($M_{coo}, base, len$) // create spatial tree
M_{CTS} = prune(M_{sp}) // compress the spatial tree
return M_{CTS}

2.1 Building Spatial Tree

Algorithm 2 createSPTree(M, base, len): M is in COO format. Returns an intermediate spatial tree

if M == NULL **then**
    return NULL
end if
if len$\leq$ B **then**
    M_Base = createBaseFormat(M, base, len)
    Let b be the next available position in base_list
    Insert M_Base at base_list[b]
    return b
end if
create node X
k = next available position at nodelist
Insert X at nodelist[k]
$len^{'} = len/2$
for i in [1, nOrthants] **do**
    create M[i] and $base_i$ // M[i] can be thought is an array of elements with indices from quadrant i.
end for
for Element e in M **do**
    put e in corresponding M[i], $i \in [1, nOrthants]$
end for
for i in [1, nOrthants] **do**
    X.child[i] = createSPTree(M[i], $base_i, len^{'}$)
end for
return k

2.2 Compressing Spatial Tree

In the worst case, the preceding algorithm createSPTree can generate a tree with $nnz \log n$ nodes. However, we need a recursive storage format that is of size $\theta(nnz)$. Here, we present an algorithm Prune to compress spatial tree and reduce their size to $\theta(nnz)$ without losing information.

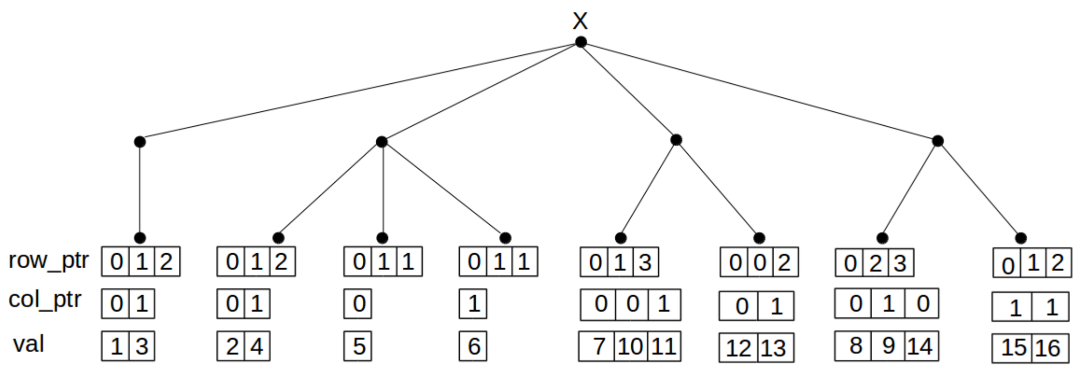

Figure 2: The CTS matrix

Algorithm 3 Prune(Node X)

if X == NULL **then**
  return NULL
end if
if X.nNonNullChild == 1 **then**
  Y = X.getNonNullChild() // returns non NULL children of a node
  if Y.nNonNullChild == 1 **then**
    Z = Y.getNonNullChild()
    X.addChild(Z)
    delete Y
  end if
end if
Prune(X.child[1]); Prune(X.child[2]); Prune(X.child[3]); Prune(X.child[4])

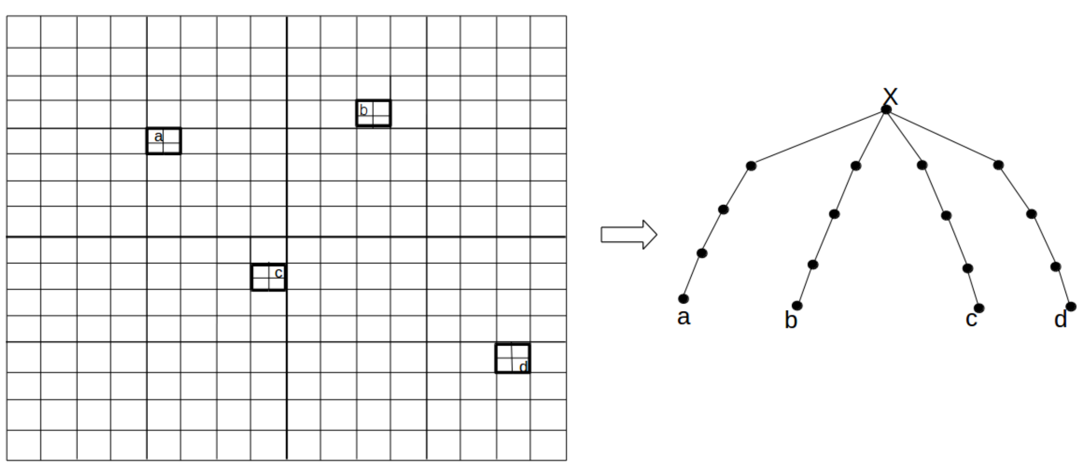

Figure 3: Before pruning spatial tree can have $\theta(nnz \log n)$ nodes

2.3 Creating CTS without building temporary spatial tree

The temporary spatial tree require $\theta(nnz \log n)$ space in worst case that is an extra space overhead compared to $\theta(nnz)$. Here, we present an algorithm that create the CTS format directly.

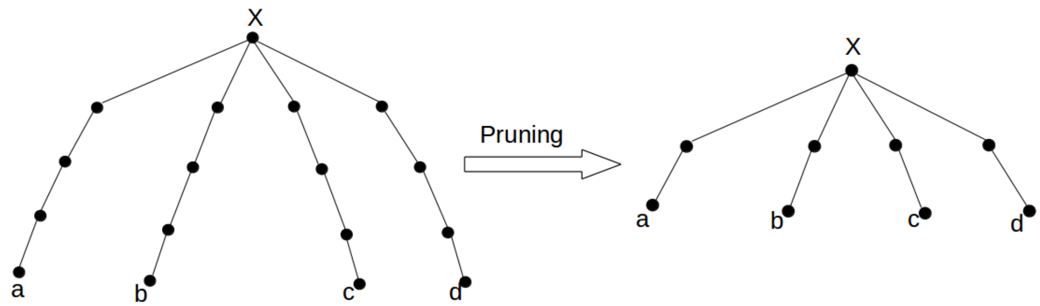

Figure 4: After pruning spatial tree has $\theta(nnz)$ edges

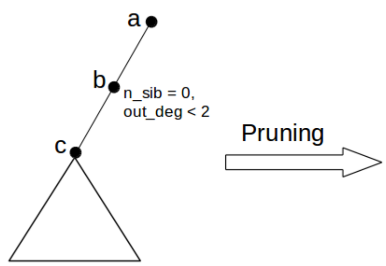

Figure 5: The pruning process

Algorithm 4 createCTS(M, base, len): M is in COO format. Returns an intermediate spatial tree

---

create root node R
Insert R at nodelist[0]
$len^{'} = len/2$
for i in [1, nOrthants] **do**
  create M[i] and $base_i$ // M[i] can be thought is an array of elements with indices from quadrant i.
end for
for e in M **do**
  put e in corresponding M[i], $i \in [1, nOrthants]$
end for
if Only one M[i] is not empty **then**
  has_sibling = FALSE
end if
for i in [1, nOrthants] **do**
  createSPTree($R, i, has\_sibling, M[i], base\_i, len^{'}$)
end for
return R

Algorithm 5 createSPTree($P, i, has\_sibling, M, base, len$): P is the parent node. M is in COO format. M will be transfered to CTS node X that will be i-th child of P. has_sibling denotes whether M has any sibling.

```
if M == NULL then
    return
end if
if len≤ B then
    M_Base = createBaseFormat(M, base, len)
    Let b be the next available position in base_list
    Insert M_Base at base_list[b]
    R[i] = b
end if
len' = len/2
for i in [1, nOrthants] do
    create M[i] and base_i // M[i] can be thought is an array of elements with indices from quadrant i.
end for
for e in M do
    put e in corresponding M[i], i ∈ [1, nOrthants]
end for
if Only one M[i] is not empty then
    has_sibling_child = FALSE
end if
if has_sibling == FALSE and has_sibling_child == FALSE then
    createSPTree(P, i, FALSE, M[i], base_i, len')
else
    Create node X
    Add X in next available position k in node_list.
    P[i] = k
    for i in [1, nOrthants] do
        createSPTree(X, i, has_sibling_child, M[i], base_i, len')
    end for
end if
```

2.4 Properties of CTS format

Every node x has 2 fields - base(starting coordinates of the bounding box corresponding to the node) and len(length of the dimension of the bounding box). Let C_x denote the children of a node x.

Property 2.1. *Every intermediate node x satisfies at least one of the following two conditions.*

1. x has at least 2 children.
2. x has a sibling(different child of same parent).

Property 2.2. *If a node x has more than one children, then $\forall c \in C_x, c.len = (x.len)/2$*

Definition 2.1. *A spatial tree storage for tensors is called **Compressed Tree Storage**(CTS), if it satisfies both Property2.1 and Property2.2.*

Lemma 1. *The size of a CTS is $\theta(nnz)$*

Proof. Every intermediate node has at least 2 children or has a sibling. □

Lemma 2. *The spatial tree created by algorithm is in CTS format.*

Proof. By construction it satifies the CTS properties. □

3 Matrix Multiplication using CTS

In this section, we present how to multiply two matrices X and Y in CTS format and get the result matrix Z in CTS format. Each node x has following two fields.

- base - the top-left most indices of the orthogonal box it corresponds to.
- len - the length of dimension of the orthogonal box it corresponds to.

Let $Par(x)$ denotes node x's parent, $Par^i(x) = Par(Par^{i-1}(x))$ for $i \geq 2$ and $Par^*(x) = Par^i(x)$ for some $i \geq 1$.

getNextBox(x, y) returns x if x is an immediate child of y, otherwise it returns y's immediate child node $x^{'}$ where $x^{'} = Par^*(x)$.

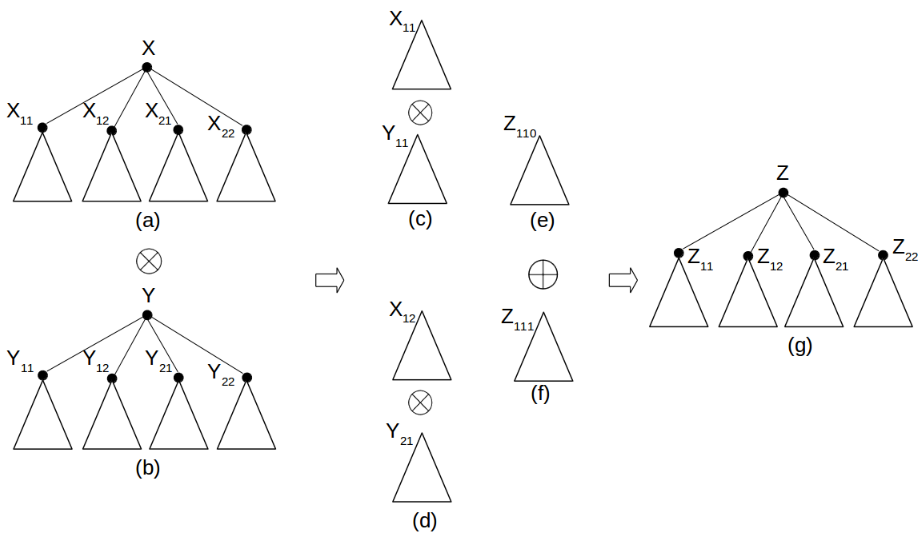

Figure 6: Dense matrix multiply using tree storage

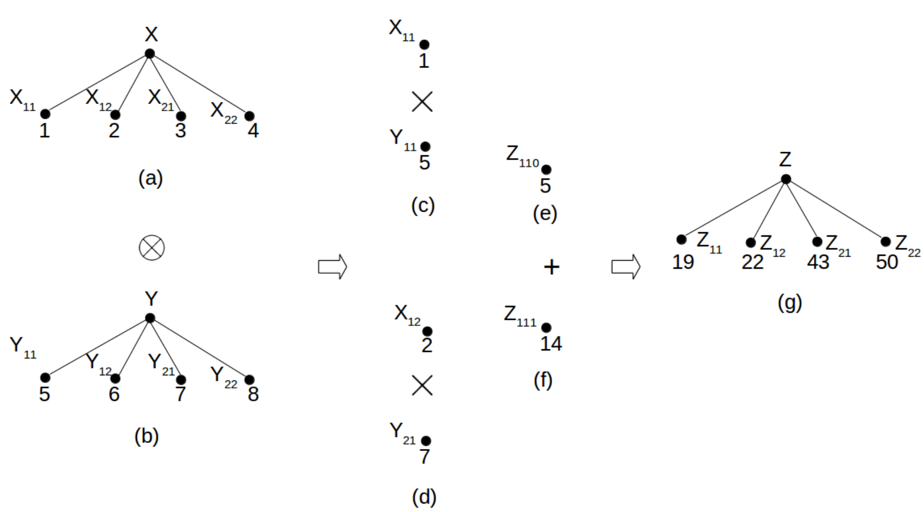

Figure 7: Matrix multiply base case

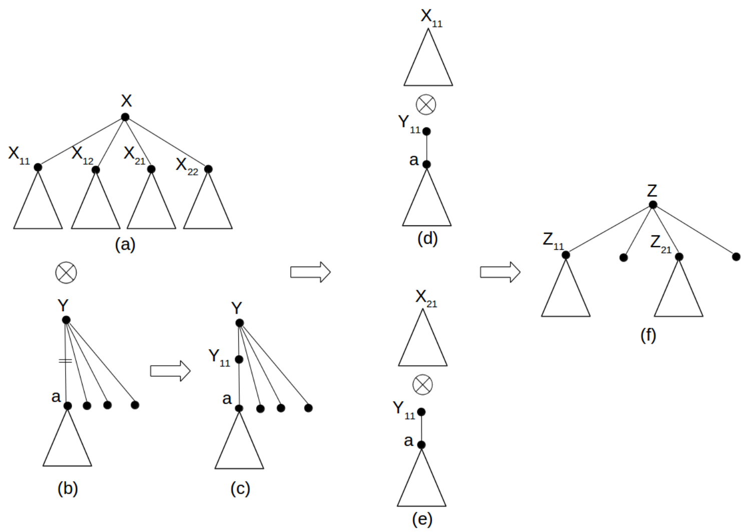

Figure 8: Sparse matrix multiply using compressed tree storage - case 1

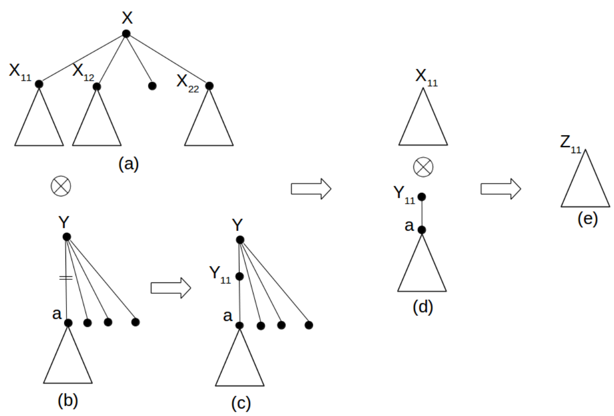

Figure 9: Sparse matrix multiply using compressed tree storage - case 2

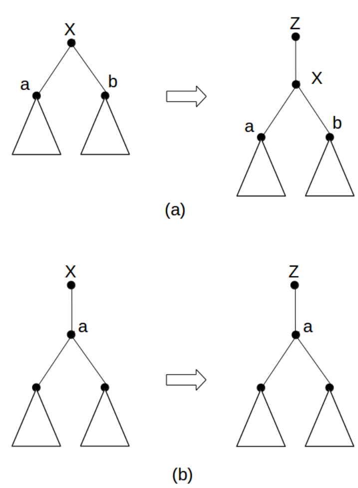

Figure 10: Changing root node of a returned tree, before doing merging with another tree

Algorithm 6 TensorProduct(X, X_l, Y, Y_l, Z) : X and Y are roots of input trees. X_l and Y_l are two nodes to keep order of the subtree multiplication. Z is a node to keep track of the output node properties. Returns a tree in CTS form

if X == NULL OR Y == NULL **then**
  return NULL
end if
if X.dim_len == X_l.dim_len and Y.dim_len == Y_l.dim_len **then**
  $X11_l$ = getNextBox(X11, X_l); $X12_l$ = getNextBox(X12, X_l); $X21_l$ = getNextBox(X21, X_l); $X22_l$ = getNextBox(X22, X_l)
  $Y11_l$ = getNextBox(Y11, Y_l); $Y12_l$ = getNextBox(Y12, Y_l); $Y21_l$ = getNextBox(Y21, Y_l); $Y22_l$ = getNextBox(Y22, Y_l)
  tree *t110 = TensorProduct(X11, $X11_l$, Y11, $Y11_l$, Z11); tree *t111 = TensorProduct(X12, $X21_l$, Y21, $Y21_l$, Z11)
  tree *t120 = TensorProduct(X11, $X11_l$, Y12, $Y12_l$, Z12); tree *t121 = TensorProduct(X12, $X12_l$, Y22, $Y22_l$, Z12)
  tree *t210 = TensorProduct(X21, $X21_l$, Y11, $Y11_l$, Z21); tree *t211 = TensorProduct(X22, $X22_l$, Y21, $Y21_l$, Z21)
  tree *t220 = TensorProduct(X21, $X21_l$, Y12, $Y12_l$, Z22); tree *t221 = TensorProduct(X22, $X22_l$, Y22, $Y22_l$, Z22)
  tree *t11 = merge(t110, t111); tree *t12 = merge(t210, t211); tree *t21 = merge(t210, t210); tree *t22 = merge(t220, t221)
else if X.dim_len == X_l.dim_len **then**
  $X11_l$ = getNextBox(X1, X_l); $X12_l$ = getNextBox(X2, X_l); $X3_l$ = getNextBox(X3, X_l); $X4_l$ = getNextBox(X4, X_l)
  $Y_l^{'}$ = getNextBox(Y_l, Y); int quad = getQuad($Y_l^{'}$, Y_l)
  if quad == 1 **then**
    tree *t11 = TensorProduct(X11, $X11_l$, Y, $Y_l^{'}$, Z11); tree *t21 = TensorProduct(X21, $X21_l$, Y, $Y_l^{'}$, Z21)
  else if quad == 2, 3 or 4 **then**
    // Similar as before
  end if
else if Y.dim_len == Y_l.dim_len **then**
  //Similar logic
else
  $X_l^{'}$ = getNextBox(X_l, X)
  $Y_l^{'}$ = getNextBox(Y_l, Y)
  int quad1 = getQuad($X_l^{'}$, X_l)
  int quad2 = getQuad($Y_l^{'}$, Y_l)
  if quad1 == 2 and quad2 == 3 **then**
    tree *$t^{'}$ = TensorProduct(X, $X_l^{'}$, Y, $Y_l^{'}$);
  else if Other 7 combinations of quad1 and quad2 **then**
    // Similar logic
  end if
end if
if $t^{'}$ is the only non-NULL child **then**
  return $t^{'}$
else if If more than one children is not NULL **then**
  tree* t = createTree(Z, t11, t12, t21, t22);
  return t
else if If all children are NULL **then**
  return NULL
end if

3.1 Merging two CTS tensors

Here, we present an algorithm that merges two CTS matrices X and Y where $root(X).len = root(Y).len$ and returns a CTS matrix Z where $root(X).len = root(Z).len$.

Algorithm 7 Merge(Node X, Node Y)

```
if X == NULL and Y == NULL then
    return NULL
else if X == NULL then
    return Y
else if Y == NULL then
    return X
else if X == LEAF and Y == LEAF then
    return Tensor_Addition(X,Y)
end if
create node Z with same length and base of node X
if X.nNonNullChild > 1 and Y..nNonNullChild > 1 then
    Implement algorithm from fig. 13
else if X.nNonNullChild > 1 and Y.nNonNullChild == 1 then
    Implement algorithm from fig. 12 and 13
else if Y.nNonNullChild > 1 and X.nNonNullChild ==1 then
    Implement algorithm from fig. 12 and 13
else if X.nNonNullChild == 1 and Y.nNonNullChild == 1 then
    Implement algorithm from fig. 14, 18, 16 and 19
end if
return Z
```

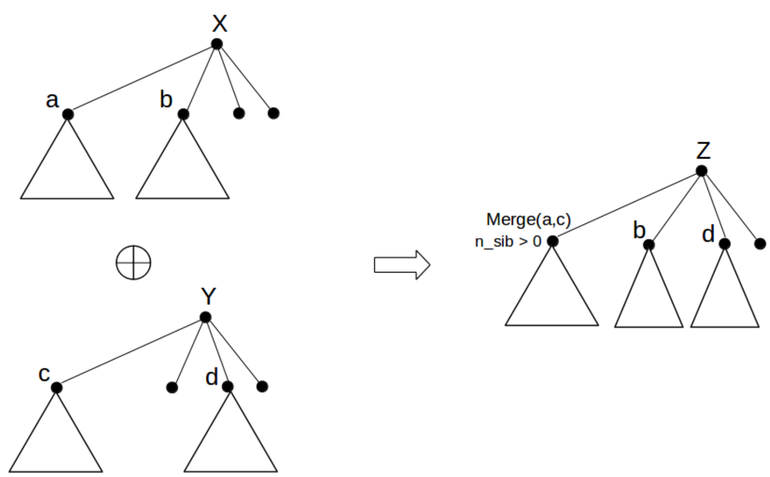

Figure 11: Merging two CTS tensors when both trees' roots have more than one children

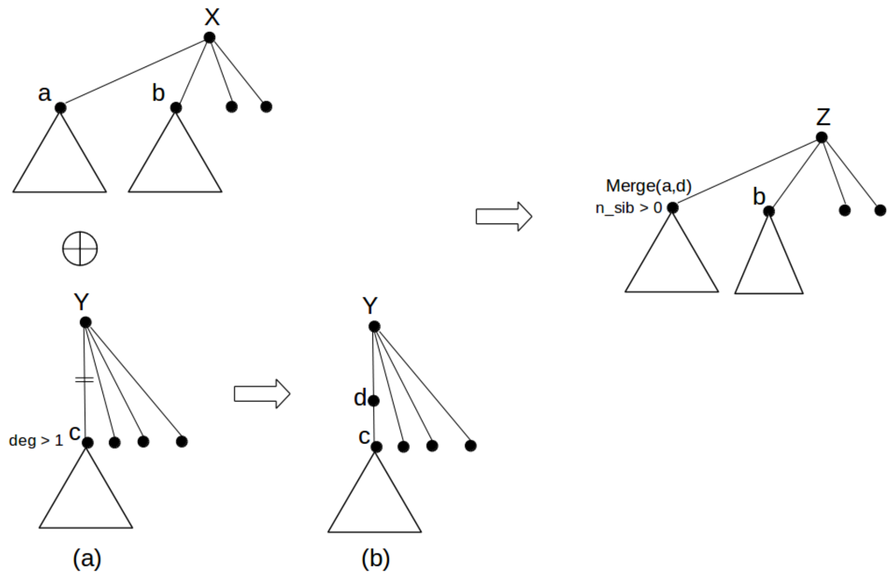

Figure 12: Merging two CTS tensors when only one tree's root has more than one children - case1

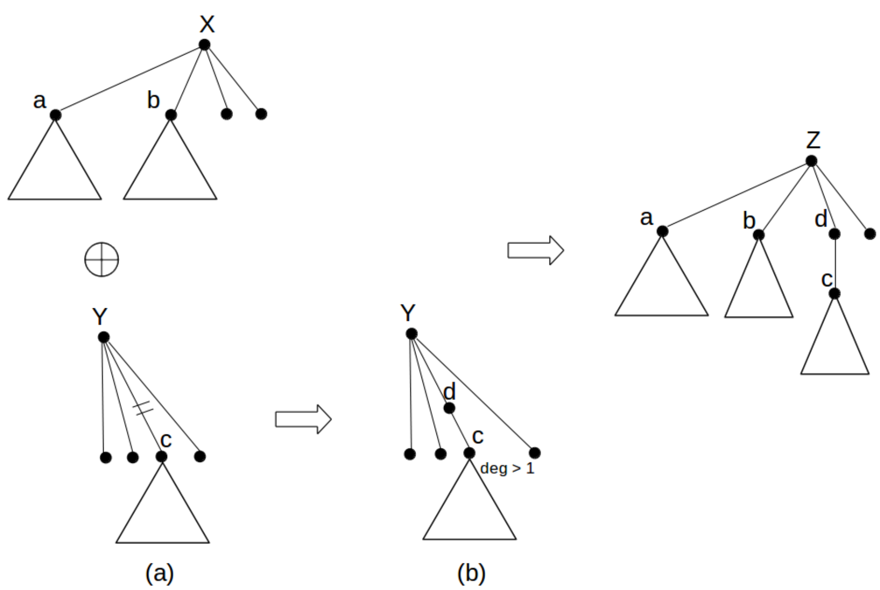

Figure 13: Merging two CTS tensors when only one tree's root has more than one children - case2

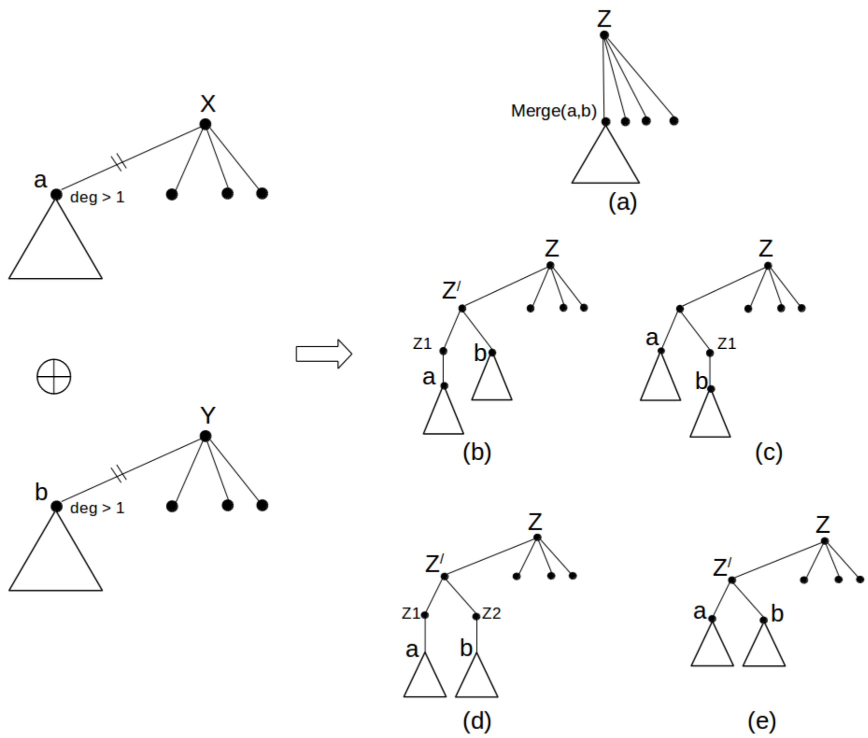

Figure 14: Merging two CTS tensors when both trees' roots have only one child - case1

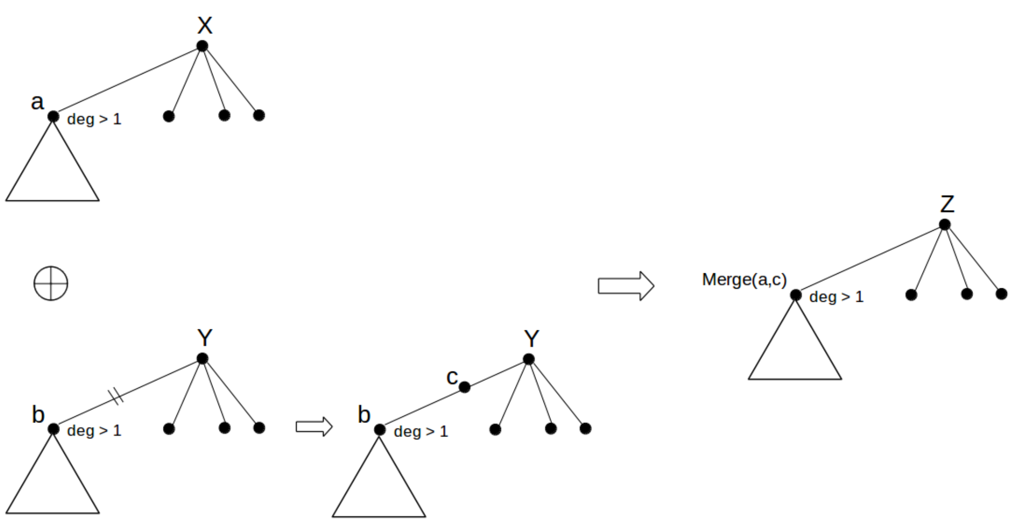

Figure 15: Merging two CTS tensors when both trees' roots have only one child - case2

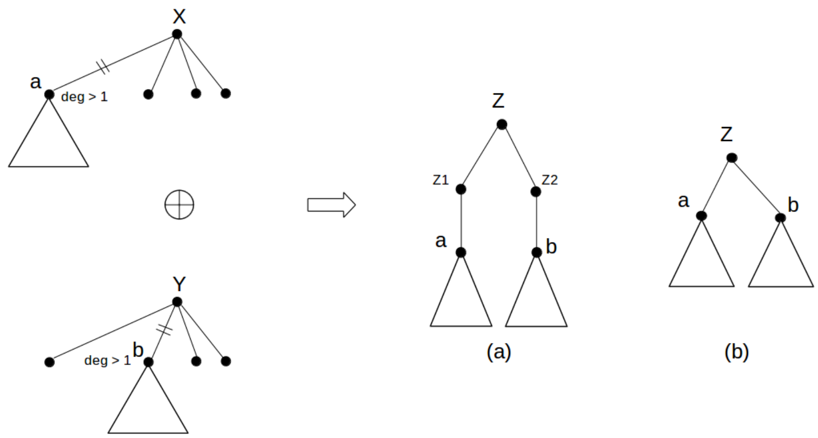

Figure 16: Merging two CTS tensors when both trees' roots have only one child - case3

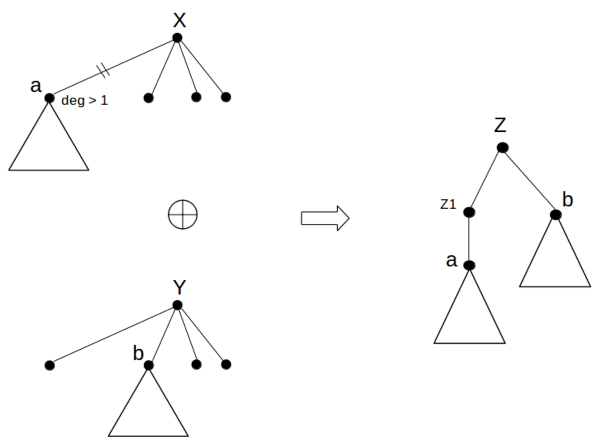

Figure 17: Merging two CTS tensors when both trees' roots have only one child - case4

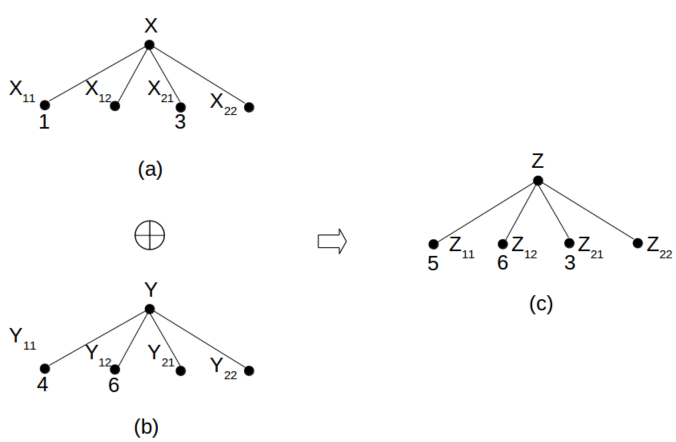

Figure 18: Merging base case

3.2 Avoiding malloc() and free() in function calls of tensor product

In serial execution, at any point of time, at most $12 \log n$ trees will be present. So, we created one block space for these trees and by using array redoubling method, we amortize the total space usage.

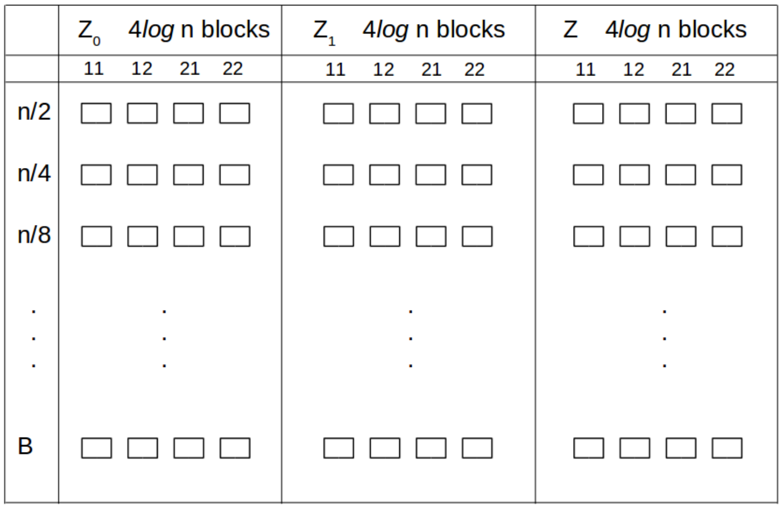

Figure 19: Merging two CTS tensors when both trees' roots have only one child - case4

3.3 Optimality condition for using any recursive storage

Let two sparse matrices X and Y get multiplied and the output be Z. Let X has r_{nnz} non-empty rows and Y has c_{nnz} non-empty columns. Also, let each row of X contains k_r non-zero elements(row-sparsity) and each column of Y contains k_c non-zero elements(column-sparsity). Then number of elementary multiplication is $\theta(r_{nnz} \times c_{nnz} \times (k_r + k_c))$. So, looping based algorithm will incur $\theta(r_{nnz} \times c_{nnz} \times (k_r + k_c)/B)$ cache misses.

Let X, Y, Z contain nnz_X, nnz_Y, nnz_Z non-zero elements respectively. The recursive storage conversion cost is lower bounded by $Sort(nnz_X) + Sort(nnz_Y) + Sort(nnz_Z)$. Also, the merging cost in tensor product is $\theta((nnz_Z \log(n^2/M))/B)$

Theorem 3. *If* $(Sort(nnz_X) + Sort(nnz_Y) + Sort(nnz_Z) + \theta((nnz_Z \log(n^2/M))/B)) = \omega((\theta(r_{nnz} \times c_{nnz} \times (k_r + k_c)/B)))$*, then no recursive storage format would give better cache complexity compared to looping based algorithms.*

Proof. Sorting and merging cost should be less than cost of elementary multiplications. □

3.4 In-place algorithm

Theorem 4. *There is no in-place recursive divide and conquer algorithm for sparse tensor product.*

Proof. Let us consider 2D matrix multiplication. Let Z_{11} is being created by doing product from two subproblems $A = X_{11} \times Y_{11}$ and $B = X_{12} \times Y_{21}$. Wlog, let suppose A is computed first. If the algorithm is in-place, then the compressed form of output is written in Z_{11}. Let suppose k be a position in the dense form of Z_{11}, that is currently zero after computing A. Then there will be no entry of k in Z_{11}. Now, if subproblem B creates a non-zero value for entry k, there is no space to hold it in the compressed form. So in that case, we can rewrite the Z_{11} with updated values from B - thus the algorithm becomes not-in-place. Otherwise, we can append the value at end of Z_{11}. However, in that case later scanning of Z_{11} will incur $\theta(nnz)$ cache misses, instead of $\theta(nnz/B)$ I/O, which is bad. This problem could have been solved provided, we know the output size of A and B ahead before creating space for Z_{11}. However, this requires to solve recursively the whole problem without allocating any space. This is a contradiction. Hence, no in-place recursive divide and conquer algorithm is possible for sparse tensor product. □