Unified Convolution Framework: A compiler-based approach to support sparse convolutions

Jaeyeon Won [1] **Changwan Hong** [1] **Charith Mendis** [2] **Joel S. Emer** [1] **Saman Amarasinghe** [1]

Abstract

This paper introduces a Unified Convolution Framework (UCF) that incorporates various existing sparse convolutions in a unified abstraction. This work is in contrast to the common library-based approach that requires much engineering effort because each different sparse convolution must be implemented separately. Instead, it employs a tensor compiler approach that can flexibly explore convolutions with various program transformations; however, no compiler can currently support various sparse convolutions flexibly to our knowledge. In particular, the Tensor Algebra Compiler (TACO) can support a variety of sparse formats but cannot declare convolutions because a tensor cannot be accessed by a linear combination of index variables. We extend TACO's Einsum language to support an affine index expression to declare a convolution. Our method is also compatible with TACO's format and scheduling language, enabling various sparse convolution implementations to be explored. Our experimental results demonstrate that TACO-UCF achieves 1.32× and 8.3× average speedups on a filter sparse convolution and a submanifold sparse convolution, respectively, over state-of-the-art libraries on CPU. TACO-UCF on GPU outperforms the state-of-the-art GPU library on filter sparse convolution of ResNet50 by an average of 1.47× at 80% sparsity. We also demonstrate TACO-UCF outperforms on a neighbor retrieval of a submanifold sparse convolution by an average of 2.55× and 3.34× over MinkowskiEngine and TorchSparse on GPU, respectively.

1 Introduction

In deep neural networks, a convolution is one of the most important computations. However, a convolution is computationally expensive and is usually a performance and memory bottleneck of the entire network (Sze et al., 2020). Therefore, there have been many efforts to make convolutions as lightweight as possible. Among them, a popular approach is sparsifying the convolution because it effectively reduces the network size and computational cost.

There are many ways to introduce sparsity in a convolution. For example, a filter can have a lot of zeros after pruning a network (Han et al., 2015; Mao et al., 2017). An activation after the ReLU layer can have a large number of zeros (Gong et al., 2020; Kurtz et al., 2020). Submanifold sparse convolution (Graham et al., 2018) is another example, which restricts the points at which a convolution is performed to reduce the number of convolution operations.

While sparsifying a convolution can improve performance theoretically, writing sparse convolution code that achieves a speedup over a dense counterpart is not trivial. Developers have to optimize code that involves many indirect accesses to data stored in a sparse data representation. In addition, different operands must be stored in different sparse formats for each class of sparse convolution. Therefore, various sparse convolutions are viewed independently and implemented separately in different libraries.

While the libraries have a very individual and narrow focus, tensor compilers such as Halide (Ragan-Kelley et al., 2013), TVM (Chen et al., 2018), Tiramisu (Baghdadi et al., 2019), and TACO (Kjolstad et al., 2017) can support a wide range of tensor computations by exploring various optimizations. **However, existing tensor compilers currently cannot generate efficient code for sparse convolutions in various formats**. For instance, Halide, TVM, and Tiramisu excel at generating an efficient dense convolution but do not support most sparse formats. TACO, on the other hand, is capable of supporting various sparse formats. However, its language can only declare simple computations, such as matrix multiplication, and does not accept a full affine index expression, which is essential in defining a sliding window in convolution, e.g., $C_i = \sum_k A_{\underline{i+k}} * B_k$.

We introduce the Unified Convolution Framework (UCF), which can express all types of sparse convolutions, including sparse filter convolution, sparse activation convolution, and submanifold sparse convolution. We implemented TACO-UCF by extending TACO's Einsum language to allow defining a convolution and support the UCF. Furthermore, TACO-UCF can explore multiple sparse formats and loop trans-

[1]Massachusetts Institute of Technology, Massachusetts, USA [2]University of Illinois at Urbana-Champaign, Illinois, USA. Correspondence to: Jaeyeon Won <jaeyeon@mit.edu>.

Proceedings of the 6th MLSys Conference, Miami Beach, FL, USA, 2023. Copyright 2023 by the author(s).

	Sparse Convolutions					Backends	
Name	Filter SpConv	Activation SpConv	Submanifold SpConv	Dual SpConv	Formats	CPU	GPU
SkimCaffe	✓	✗	✗	✗	1	✓	✗
TorchSparse	✗	✗	✓	✗	1	✓	✓
DeepSparse	✓	✓	✗	✗	1	✓	✗
TACO	✗	✗	✗	✗	> 100	✓	✓
Our Work (TACO-UCF)	✓	✓	✓	✓	> 100	✓	✓

Table 1. Comparison of existing sparse convolution libraries and compiler with our work. SkimCaffe (Park et al., 2016), TorchSparse (Tang et al., 2022), and DeepSparse (Kurtz et al., 2020) are library-based, and TACO (Kjolstad et al., 2017) is a compiler-based approach. Activ. and Subm. mean an activation and submanifold sparse convolution, respectively. Dual Sparse. means a dual sparse convolution where both operands are sparse. A yellow checkmark indicates it is performant in some instances.

formations by utilizing TACO's format and scheduling language. Table 1 compares TACO-UCF with existing sparse convolution libraries and tensor compilers. The contributions of this paper are as follows:

- We extend sparse tensor algebra compiler theory, which currently only supports a single index variable, to handle full affine index expressions.
- We introduce the Unified Convolution Framework(UCF) that can express all different kinds of sparse convolutions. We implement TACO-UCF by extending the TACO compiler to support the UCF.
- We show that TACO-UCF has comparable performance to hand-implemented best of class convolutions on CPUs. TACO-UCF outperforms SkimCaffe on filter sparse convolutional layers of ResNet50 by an average of 1.32× at 80% sparsity. TACO-UCF significantly surpasses MinkowskiEngine on a submanifold sparse convolution by 8.3× on average.
- We also can produce GPU code that outperforms best of class in certain cases. TACO-UCF outperforms cuDNN and Escort on filter sparse convolutional layers of ResNet50 by an average of 1.61× and 1.47× at 80% sparsity, respectively. TACO-UCF outperforms MinkowskiEngine and TorchSparse on retrieving neighbors as part of a submanifold sparse convolution by 2.55× and 3.34× on average, respectively.

2 BACKGROUND

This section describes the abstractions and accompanying languages used in tensor compilers.

2.1 Tensor Expression Language

Some tensor compilers, such as Halide (Ragan-Kelley et al., 2013) and TVM (Chen et al., 2018), use a tensor expression language to describe a deep learning operator as a tensor computation. A tensor expression language allows the user to write a full affine index expression to describe tensor access. For example, a 1D convolution can be written in a tensor expression language : $C_i = A_{i+k} * B_k$

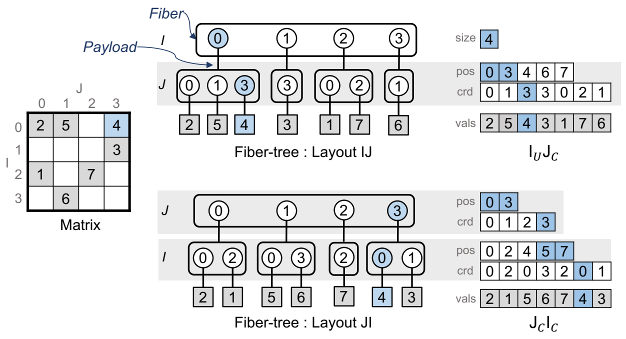


Figure 1. A sparse matrix with dimensions I and J, its fibertree abstraction (in both level orders) and concrete representations for two data layouts, $I_U J_C$ and $J_C I_C$.

In a tensor expression language, a computation is a traversal of all coordinates in the index variables. It performs the indicated tensor expression with a reduction if the same output is referred to more than once. In other words, a reduction happens along the index variable (k) not appearing in the output. The reduction can occur with various operations such as sum, min, and max. The reduction along the k indicates the sum in the convolution.

Conversely, TACO (Kjolstad et al., 2017; Henry et al., 2021) uses an Einsum language, a more restrictive subset of tensor expression language, that accepts **only a single index variable** to describe access in a single dimension. Thus, TACO can express a matrix multiplication ($C_{i,j} = A_{i,k} * B_{k,j}$) or a stride access ($C_i = A_{2*i}$), but not a convolution.

2.2 Format Language

While the tensor expression language describes what to compute, a format language describes how to store the tensor. Some format languages are based on the coordinate tree abstraction, which was first described in the format abstraction (Chou et al., 2018) in TACO and later abstracted further and formalized as the *fibertree* abstraction (Sze et al., 2020).

Figure 1 shows how the fibertree abstraction represents a matrix. Any tensor is first viewed as a tree with each fiber carrying a set of coordinates and a payload that is either a fiber at the next level or a value at the bottom of the tree.

The order of the levels indicates the data layout such as row-major or column-major. Once the layout is specified, a *level format* specifies what physical storage is used to store the fiber. An Uncompressed (U) level format encodes a dense coordinate interval $[0, N)$. A Compressed (C) level format encodes only non-zero coordinates in the fiber by explicitly storing coordinates. A format language $I_U J_C$ in the Figure 1 says the matrix is stored in the $I \rightarrow J$ layout (row-major), and level formats where I and J are Uncompressed and Compressed, respectively. We refer to a *position* as the index of an element in the concrete data representation. For example, the color-highlighted coordinate=3 in level J of the $I_U J_C$ has a position=2 in the crd array (assuming zero-based indexing).

A combination of level splitting, fusing, reordering, and level format choice can express many representations. For example, if we fuse two levels I and J and then store them with Compressed format ($(IJ)_C$), it represents a list of pairs of row and column coordinates, known as the COO format.

2.3 Scheduling Language

A scheduling language describes how to compute an algorithm defined by a tensor expression and format. Inspired by Halide (Ragan-Kelley et al., 2013), many tensor compilers decouple the schedule (how to compute) from the algorithm (what to compute). A scheduling language is often a set of basic program transformation primitives, such as loop tiling and reordering. Decoupling algorithms from schedules allows tensor compilers to explore many ways to traverse the iteration space defined by nested loops. The appropriate schedule will maximize data locality and parallelism in the code, and the appropriate data representation will store a sparse tensor compactly in memory.

3 Related Works

3.1 Library-based Approach

Different sparse convolutions are implemented in separate libraries. Since hand-crafted libraries requires significant engineering effort, they often support a limited number of formats, architectures, and layer shapes.

Filter Sparse Convolution. While XNNPACK (Elsen et al., 2020) implements the convolution by converting into SpMM via `im2col`, SkimCaffe (Park et al., 2016) implements a direct sparse convolution that avoids redundant input caused by `im2col`. Studies have also made pruning more hardware-friendly, making a sparsity pattern more structured (Wen et al., 2016; Mao et al., 2017; Niu et al., 2020). Both libraries support only a single format and specific layer shapes. For example, XNNPACK only supports a 1×1 filter, and SkimCaffe is not optimized for stride convolution.

Activation Sparse Convolution. Because of the nature of

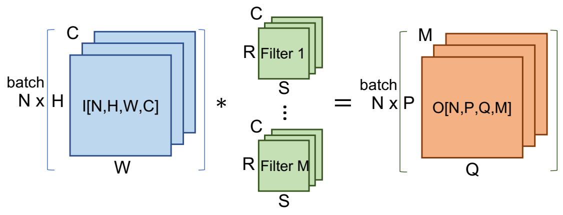


Figure 2. Visualization of 2D Convolution.

Rectified Linear Unit (ReLU), a non-trivial portion of the activations are sparse (Kurtz et al., 2020). DeepSparse (Gong et al., 2020) implements an efficient vectorized format conversion routine for sparse activation to reduce the conversion cost incurred during the inference. However, their implementation only supports the CPU as a backend.

Submanifold Sparse Convolution. A *submanifold sparse convolution* (Graham et al., 2018) takes a sparse input, such as 3D point clouds, and restricts the output sparsity pattern to match the input sparsity pattern, preventing the number of nonzeros from increasing (Figure 3). The key is to quickly find neighbors inside the filter window on the input stored in a sparse format. Existing libraries such as MinkowskiEngine (Choy et al., 2019) and TorchSparse (Tang et al., 2022) achieve this using a hash table. While TorchSparse outperforms MinkowskiEngine, their implementation is limited to 3D data and their CPU performance is not as streamlined as the GPU's.

3.2 Compiler-based Approach

TVM (Chen et al., 2018) and Tiramisu (Baghdadi et al., 2019) offer basic supports for generating sparse loops, but they are designed to optimize dense loops. Taichi (Hu et al., 2019) provides a format-agnostic programming language and compiler, allowing users to use sparse formats flexibly. However, Taichi does not support a co-iteration between multiple sparse operands, such as an intersection of two coordinate lists. TACO (Kjolstad et al., 2017), COMET (Tian et al., 2021), and sparse_tensor MLIR dialect (Bik et al., 2022) support efficient co-iteration between various sparse formats. However, these compilers use the Einsum as an interface, which does not support convolution. In this paper, we are bringing TACO's Einsum closer to a tensor expression language of TVM and Halide to express a convolution.

4 Unified Convolution Framework

In this section, we show how to extend existing tensor compilers to express all different sparse convolutions. This framework consists of a tensor expression language with full affine index expression, a format language, and a scheduling language. In Table 2, we unify existing implementations of sparse convolution. We first classify convolutions into two categories regarding mathematical equivalence: a conventional and a masked convolution.

Category	Name	Tensor Expression Language	Format Language	Detail
Conventional Convolution	Dense	$O_{n,p,q,m} = I_{n,p+r,q+s,c} * F_{r,s,c,m}$	All Dense	-
	SkimCaffe (Park et al., 2016)		$F : M_U(CRS)_C$	Section 4.1.1
	DeepSparse (Kurtz et al., 2020)		$I : N_U H_U W_U C_C$	Section 4.1.2
	XNNPACK (Elsen et al., 2020)	Im2col: $I'_{n,p,q,r,s,c} = I_{n,p+r,q+s,c}$ SpMM: $O_{(n,p,q),m} = I'_{(n,p,q),(r,s,c)} * F_{(r,s,c),m}$	$F : M_U(RSC)_C$	Section 4.1.1
Masked Convolution (Submanifold Convolution)	Direct approach	$O_{n,p,q,m} = Mask_{n,p,q} * I_{n,p+r,q+s,c} * F_{r,s,c,m}$	$O : N_U P_U Q_C M_U$ $Mask : N_U P_U Q_C$ $I : N_U H_U W_C C_U$ $F : R_U S_U C_U M_U$	Section 4.2.1
	MinkowskiEngine (Choy et al., 2019) TorchSparse (Tang et al., 2022)	$Map_{i,j,k} = Mask_{n,p,q} * I_{n,p+r,q+s,:} * F_{r,s,:,:}$ $(i = pos(Mask_{n,p,q}), j = pos(I_{n,p+r,q+s,:}),$ $k = pos(F_{r,s,:,:}))$ $OVal_{i,m} = Map_{i,j,k} * IVal_{j,c} * FVal_{k,c,m}$	$Map : (KIJ)_C$ $O : (NPQ)_C M_U$ $Mask : (NPQ)_C$ $I : (NHW)_C C_U$ $F : R_U S_U C_U M_U$	Section 4.2.2

Table 2. Various sparse convolutions described in a unified convolution framework (UCF).

4.1 Conventional Convolution

Although prior works view a filter sparse convolution and activation sparse convolution separately, we found that they are expressed identically in the tensor expression language, but are expressed differently in the format language. In a tensor expression language, a 2D conventional convolution (Figure 2) is defined as : $O_{n,p,q,m} = I_{n,p+r,q+s,c} * F_{r,s,c,m}$ There are variants of convolutions such as a strided, dilated (Yu & Koltun, 2015), and grouped convolution (Howard et al., 2017), which can also be described in a tensor expression language. However, this paper will mainly focus on the conventional convolution.

4.1.1 *Filter Sparse Convolution*

A pruned filter should be stored in a sparse data representation to leverage its sparsity. For example, SkimCaffe flattens (C, R, S) levels and stores a filter in $M_U(CRS)_C$ format. However, flattening a 4D filter into 2D may lose the information of the high-dimensional sparsity pattern. Many previous works (Wen et al., 2016; Mao et al., 2017) have investigated storing a pruned filter in more hardware-friendly sparsity structures. However, some structured sparsity patterns should be considered in a 4D tensor to fully utilize its pattern. In section 6.2.1, we found that the most compact format for storing the MC-structure (Figure 7(a)) is $M_C C_C R_U S_U$.

Another way to compute a convolution is turning it into a matrix multiplication via the `im2col` operation. XNNPACK converts a filter sparse convolution to a sparse matrix-dense matrix multiplication (SpMM).

4.1.2 *Activation Sparse Convolution*

Similar to filter sparse convolution, an input I should have a sparse data representation to leverage a sparsity in an activation. Once an activation comes out from the ReLU layer, an input activation I should be converted into a sparse format from a dense format during the inference. For example, DeepSparse Engine converts I into $N_U H_U W_U C_C$ from $N_U H_U W_U C_U$. In order to minimize the cost in format conversion, only the last level's format is changed to Compressed while the layout is kept the same. In UCF, a dual sparse convolution can be readily described by declaring both I and F to have sparse formats in a format language.

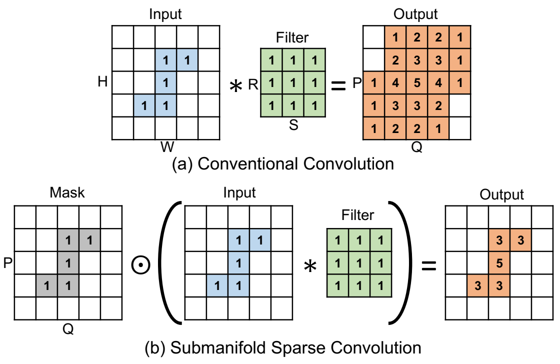
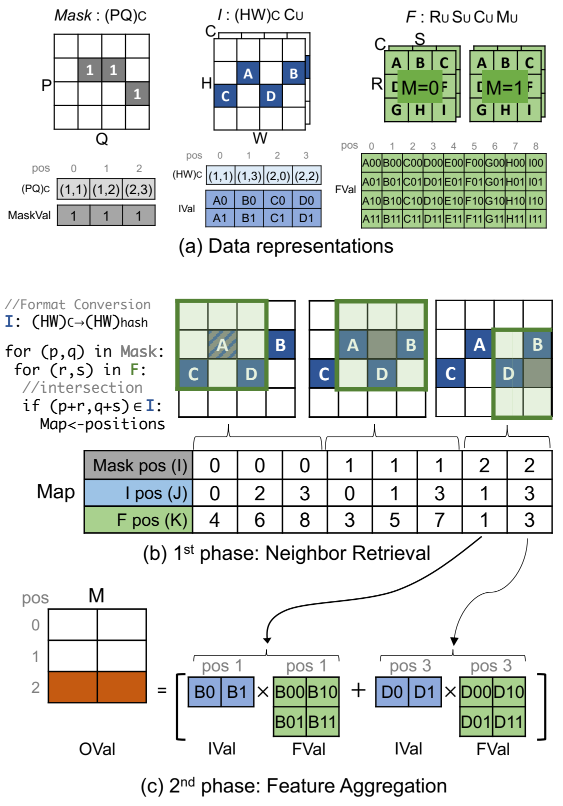

Input
Filter
Output
H
W
R
S
P
Q
(a) Conventional Convolution
Mask
Input
Filter
Output
P
Q
(b) Submanifold Sparse Convolution

Figure 3. In a submanifold sparse convolution, a mask has the same sparsity pattern as the input sparsity pattern.

4.2 Masked Sparse Convolution

2D batch masked sparse convolution (often called as a generalized sparse convolution (Choy et al., 2019)) can be represented in tensor expression language as follows: $O_{n,p,q,m}$=$Mask_{n,p,q}$*$I_{n,p+r,q+s,c}$*$F_{r,s,c,m}$ where a $Mask$ can only have a binary value, 0 or 1. The only difference from the conventional convolution is the existence of $Mask$. An element-wise multiplication of the sparse mask makes the output's sparsity pattern identical to $Mask$'s. If the $Mask$ is fully dense, it produces the same output as a conventional convolution. Submanifold sparse convolution is a special case of masked convolution where the sparsity pattern of the mask is the same as the input (Figure 3(b)). Note that a tensor expression language can be easily extended into 3D submanifold convolution as well.



```
//Format Conversion
I: (HW)C→(HW)hash

for (p,q) in Mask:
 for (r,s) in F:
  //intersection
  if (p+r,q+s)∈I:
   Map<-positions
```

Map								
Mask pos (I)	0	0	0	1	1	1	2	2
I pos (J)	0	2	3	0	1	3	1	3
F pos (K)	4	6	8	3	5	7	1	3

Figure 4. In map-based approach, a masked convolution is divided into two phases : (b) a neighbor retrieval and (c) a feature aggregation. Map stores the positions of tensors in a neighbor relationship. For example, (2,1,1) in the Map indicates ($Mask(2,3)$, $In(1,3,:)$, $F(0,1,:,:)$) from data representations depicted in (a).

4.2.1 *Direct Approach*

The direct approach computes a convolution by sliding a filter over an input directly according to the sparsity pattern of $Mask$. Except for the filter F, all operands in a masked convolution are stored in a sparse format. Input point clouds typically have sparse spatial coordinates (H and W in $I[N, H, W, C]$) and dense channel features. A format language has to place spatial dimensions in the upper layout with Compressed format and the channel at the last level. In section 6.3.1, we evaluate a direct approach with I stored in $N_U H_U W_C C_U$ format.

4.2.2 *Map-based Approach*

A map-based approach (Figure 4) is another method to implement a masked convolution. Existing libraries such as MinkowskiEngine and TorchSparse use this approach. Whereas the direct approach finds neighboring input points for each output and aggregates neighbors' features on the fly, a map-based approach separates the computation into two stages: (1) a neighbor retrieval and (2) a feature aggregation.

The first stage constructs a neighbor map with the positions of $Mask$, I, and F, in which they are in a neighbor relationship. Figure 4(b) shows how the map is constructed. For an efficient coordinate access, the format of I is converted into a hash table. Hash table lookup retrieves neighbor points in I near a point (p, q) in $Mask$ by searching for every point $(p + r, q + s)$ covered by a filter window.

Once a neighbor is found, corresponding positions are stored in $Map_{i,j,k}$ where $i = pos(Mask_{n,p,q}), j = pos(I_{n,p+r,q+s,:}), k = pos(F_{r,s,:,:})$. Once a neighbor map retrieves all the positions, it then performs a channel feature aggregation (Figure 4(c)) : $OVal_{i,m} = Map_{i,j,k} * IVal_{j,c} * FVal_{k,c,m}$ For each position (i,j,k) in Map, vector-matrix multiplications between input and filter are performed.

5 Implementation of Unified Convolution Framework

In this section, we describe the basics of TACO and how to implement UCF on top of TACO by extending TACO's Einsum language to support an affine index expression. We also describe how to make `.reorder()` schedule works along with our compiler technique.

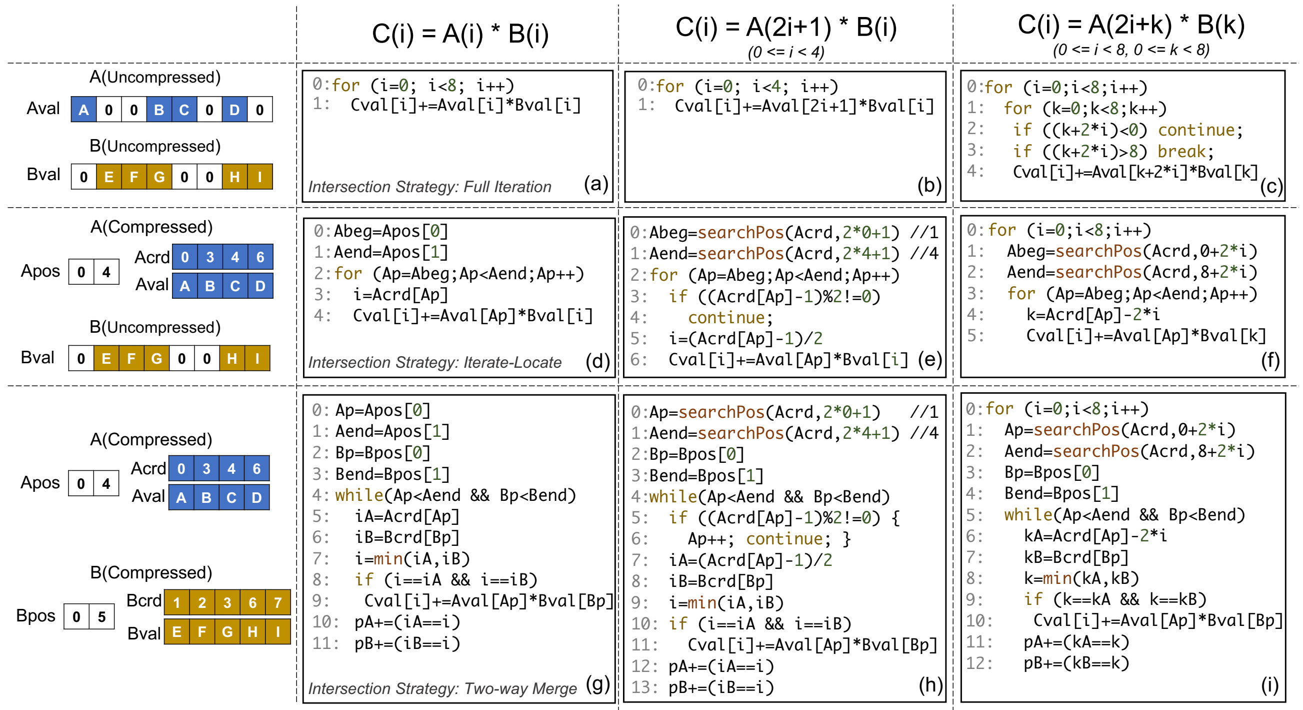
5.1 Tensor Algebra Compiler (TACO) Basics

TACO is a sparse tensor compiler that can generate efficient code to traverse a sparse iteration space while supporting various schedules and level formats (Kjolstad et al., 2017; Chou et al., 2018; Henry et al., 2021). Here, an efficient traversal often means an intersection (co-iteration) between two sparse operands to skip unnecessary computations from sparsity, i.e., $a * 0 = 0$. If two operands A_i and B_i are multiplied element-wise, TACO uses following intersection strategies depending on level formats.

A_i/B_i	**Uncompressed**	**Compressed**
Uncompressed	Full Iteration (Figure 5-(a,b,c))	Iterate-Locate
Compressed	Iterate-Locate (Figure 5-(d,e,f))	Two-way Merge (Figure 5-(g,h,i))

Figures 5 (a,d,g) show the full iteration, iterate-locate, and two-way merge intersection on $A_i * B_i$, respectively. The iterate-locate intersects A_i and B_i by iterating non-zero coordinates in the Compressed format of A (lines 2-3 in Figure 5(d)), then finds the corresponding coordinates on the B (`Bval[i]` in line 4). Two-way merge intersects two Compressed levels by incrementing a position with a smaller coordinate, as shown in lines 4-11 in Figure 5(g).

(Henry et al., 2021) proposed a technique to extend TACO's Einsum language to support a single variable affine index expression (SVAE), such as A_{2i+1}. Figure 5-(b,e,h) show generated codes for $C_i = A_{2i+1} * B_i$ within a window (often called as a range) $0 \le i < 4$. There are three changes compared to a code lowered from the Einsum language. ❶ The right start and end position in a Compressed format



(a)
```
0:for (i=0; i<8; i++)
1:  Cval[i]+=Aval[i]*Bval[i]
```

(b)
```
0:for (i=0; i<4; i++)
1:  Cval[i]+=Aval[2i+1]*Bval[i]
```

(c)
```
0:for (i=0;i<8;i++)
1:  for (k=0;k<8;k++)
2:   if ((k+2*i)<0) continue;
3:   if ((k+2*i)>8) break;
4:   Cval[i]+=Aval[k+2*i]*Bval[k]
```

(d)
```
0:Abeg=Apos[0]
1:Aend=Apos[1]
2:for (Ap=Abeg;Ap<Aend;Ap++)
3:  i=Acrd[Ap]
4:  Cval[i]+=Aval[Ap]*Bval[i]
```

(e)
```
0:Abeg=searchPos(Acrd,2*0+1) //1
1:Aend=searchPos(Acrd,2*4+1) //4
2:for (Ap=Abeg;Ap<Aend;Ap++)
3:  if ((Acrd[Ap]-1)%2!=0)
4:    continue;
5:  i=(Acrd[Ap]-1)/2
6:  Cval[i]+=Aval[Ap]*Bval[i]
```

(f)
```
0:for (i=0;i<8;i++)
1:  Abeg=searchPos(Acrd,0+2*i)
2:  Aend=searchPos(Acrd,8+2*i)
3:  for (Ap=Abeg;Ap<Aend;Ap++)
4:    k=Acrd[Ap]-2*i
5:    Cval[i]+=Aval[Ap]*Bval[k]
```

(g)
```
0: Ap=Apos[0]
1: Aend=Apos[1]
2: Bp=Bpos[0]
3: Bend=Bpos[1]
4: while(Ap<Aend && Bp<Bend)
5:   iA=Acrd[Ap]
6:   iB=Bcrd[Bp]
7:   i=min(iA,iB)
8:   if (i==iA && i==iB)
9:    Cval[i]+=Aval[Ap]*Bval[Bp]
10:  pA+=(iA==i)
11:  pB+=(iB==i)
```

(h)
```
0:Ap=searchPos(Acrd,2*0+1)   //1
1:Aend=searchPos(Acrd,2*4+1) //4
2:Bp=Bpos[0]
3:Bend=Bpos[1]
4:while(Ap<Aend && Bp<Bend)
5:  if ((Acrd[Ap]-1)%2!=0) {
6:    Ap++; continue; }
7:  iA=(Acrd[Ap]-1)/2
8:  iB=Bcrd[Bp]
9:  i=min(iA,iB)
10: if (i==iA && i==iB)
11:   Cval[i]+=Aval[Ap]*Bval[Bp]
12: pA+=(iA==i)
13: pB+=(iB==i)
```

(i)
```
0:for (i=0;i<8;i++)
1:  Ap=searchPos(Acrd,0+2*i)
2:  Aend=searchPos(Acrd,8+2*i)
3:  Bp=Bpos[0]
4:  Bend=Bpos[1]
5:  while(Ap<Aend && Bp<Bend)
6:    kA=Acrd[Ap]-2*i
7:    kB=Bcrd[Bp]
8:    k=min(kA,kB)
9:    if (k==kA && k==kB)
10:    Cval[i]+=Aval[Ap]*Bval[Bp]
11:   pA+=(kA==k)
12:   pB+=(kB==k)
```

Figure 5. Codes generated from three different tensor expression languages and three different pairs of format. Our compiler technique enables the TACO to generate codes in (c,f,i).

must be found according to the window (lines 0-1 in Figure 5(e)). ❷ A stride iteration on a Compressed format requires a guard to avoid invalid stride access (lines 3-4 in Figure 5(e)). ❸ Index variable is restored from each access on a Compressed format (line 5 in Figure 5(e)).

Challenge. Although TACO can declare the expression beyond the Einsum language, the compiler **does not accept multiple variables** in an affine index expression. However, supporting full affine index expression is essential in describing a sliding window in convolutions. In the rest of this section, we will explain how an affine index expression with multiple variables is lowered into an efficient code under existing intersection strategies.

5.2 Bringing Einsum to Support Full Affine Index Expression

Our core idea to support a full affine index expression is based on the transformation of a multi-variable affine expression (MVAE) into SVAE. In particular, just one variable in the MVAE is viewed as an index variable (*base-variable*) while the rest of the term is an *offset*. That is, MVAE turns into an SVAE form :

MVAE = *stride* * *base-variable* + *offset*

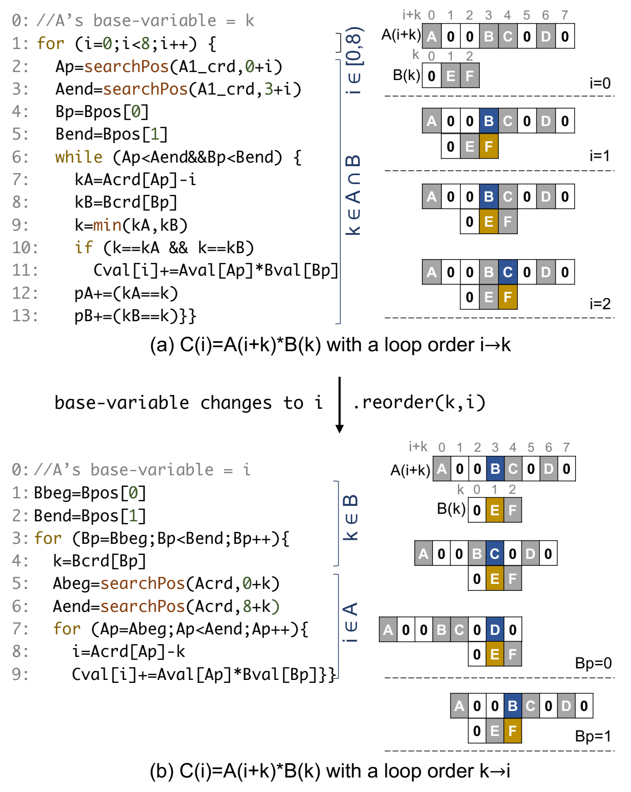
Once the MVAE is transformed, SVAE's lowering technique will be used to generate a code. For example, if there is an MVAE: $A_{2i+3j+k+3}$ and a *base-variable* j, then the access is rewritten into the SVAE form: $A_{3*\underline{j}+(2i+k+3)}$, where an underline means the base-variable.

Figure 5-(c,f,i) show how a 1D stride convolution $C_i = A_{2i+k} * B_k$ is lowered. Compared to other two tensor expressions in the left side of the figure, $C_i = A_{2i+k} * B_k$ has two index variables i and k, so it emits nested loops in which the order is $i \rightarrow k$. Because k is the base-variable here, an MVAE A_{2i+k} is rewritten into an SVAE $A_{\underline{k}+2i}$, and an intersection between $A_{\underline{k}+2i}$ and $B_{\underline{k}}$ occurs on an index variable k. While the offset in a SVAE has always been constant, the offset in a MVAE keeps changing as the iteration proceeds. In lines 1-2 in Figure 5(f), the generated code now searches start and end positions according to a window $[2i, 8+2i)$, and the window keeps sliding as the iteration i proceeds.

5.2.1 base-variable selection

In order to lower an MVAE into an SVAE form, a compiler has to choose the correct base-variable according to the loop order. Because variables in the offset have to be defined before they are used, loops of offset variables must be place outside a loop of the base-variable. Thus, the base-variable is a variable that is located in the innermost loop among the MVAE variables. In examples in Figure 5(c,f,i), k is the base-variable of the A_{2i+k} because the loop order is $i \rightarrow k$. Whenever the `.reorder()` schedule changes the loop order, a compiler has to find a new base-variable for each MVAE and rewrites it into a new SVAE form.

Effect of `reorder` schedules. Since the base-variable is



```
0: //A's base-variable = k
1: for (i=0;i<8;i++) {
2:   Ap=searchPos(A1_crd,0+i)
3:   Aend=searchPos(A1_crd,3+i)
4:   Bp=Bpos[0]
5:   Bend=Bpos[1]
6:   while (Ap<Aend&&Bp<Bend) {
7:     kA=Acrd[Ap]-i
8:     kB=Bcrd[Bp]
9:     k=min(kA,kB)
10:    if (k==kA && k==kB)
11:      Cval[i]+=Aval[Ap]*Bval[Bp]
12:    pA+=(kA==k)
13:    pB+=(kB==k)}}
```

(a) C(i)=A(i+k)*B(k) with a loop order i→k

```
base-variable changes to i    .reorder(k,i)
```

```
0: //A's base-variable = i
1: Bbeg=Bpos[0]
2: Bend=Bpos[1]
3: for (Bp=Bbeg;Bp<Bend;Bp++){
4:   k=Bcrd[Bp]
5:   Abeg=searchPos(Acrd,0+k)
6:   Aend=searchPos(Acrd,8+k)
7:   for (Ap=Abeg;Ap<Aend;Ap++){
8:     i=Acrd[Ap]-k
9:     Cval[i]+=Aval[Ap]*Bval[Bp]}}
```

(b) C(i)=A(i+k)*B(k) with a loop order k→i

Figure 6. Two generated codes with different loop orders and corresponding visualizations. A loop order $i \rightarrow k$ has a base-variable=k, while $k \rightarrow i$ has a base-variable=i for A_{i+k}.

dependent on the loop order, the intersection may or may not exist in the generated code. Figure 6 shows generated codes of $C_i = A_{i+k} * B_k$ in two different loop orders where both A and B are stored in the Compressed format. When the loop order is $i \rightarrow k$ and the base-variable is k, a two-way merge intersection happens between $A_{\underline{k}+i}$ and B_k at the loop k. Figure 6(a) visualizes how the computation proceeds by intersecting on k. However, when the loop order is changed into $k \rightarrow i$, an intersection does not happen in the generated code. As shown in the visualization in Figure 6(b), this is because each index variable iterates different tensors, i.e., i and k respectively iterate $A_{\underline{i}+k}$ and B_k, but not the same tensor. As a result, two programs are semantically equivalent, but different loop orders can change the computational complexity. The complexity of the loop $i \rightarrow k$ is $O(|i| * (nnz_A + nnz_B))$, and the complexity of the loop $k \rightarrow i$ is $O(nnz_A * nnz_B))$ where $|i|$ is the dimension of the i ($|i|$=8 in Figure 6) and nnz is the number of non-zeros.

5.2.2 *Fast Window Search on the Compressed Format*

When accessing a window in the Compressed format, a compiler always emits the code that searches positions of the start and end of the window (lines 5-6 in Figure 6(b)). We made this search faster with two optimizations.

First, we can remove unnecessary searches when the window is so large that it is always guaranteed to touch the outside boundary of the crd array of the Compressed format. For example, at line 6 in Figure 6(b), the end of the window $k + 8$ is always outside of the bound($[0, 7)$) of A. In such a case, a compiler can skip the search and emit the crd array's final position instead. If the compiler identifies the beginning or end of the window is always out of bound through a simple interval analysis, it then skips the search.

The second optimization exploits the fact that the window slides. $searchPos()$ searches the position of the window bound by linearly scanning the coordinate array from the position 0. Because the window slides, $searchPos()$ can be faster by starting the search from the previous result rather than scanning from position 0. From our simple experiment, a generated code after these optimizations was about 1.3× faster than before in 1D submanifold sparse convolution.

5.3 Discussion

Loop Bound Inference. TACO does not require loop bound analysis as it operates on basic Einsums where a loop bound is always identical to a dimension of the tensor. However, TACO-UCF needs to infer the correct loop bounds due to the affine index expression, similar to existing tensor compilers (Chen et al., 2018). To achieve this, we used a simple interval analysis (Ragan-Kelley et al., 2013). As a result, by using the MVAE lowering technique, TACO-UCF generates a code that iterates only the non-zero elements stored in a sparse format within the derived loop bounds.

Targeting Hardware Intrinsics. Currently, TACO-UCF generates code only for CPUs and GPUs, but it does not target accelerators such as Nvidia Sparse TensorCore. Targeting these accelerators is a promising direction, but it would require a non-trivial effort to analyze the nested loop representation while adapting our MVAE code generation technique. We leave this as a future direction.

6 Evaluation

6.1 Experimental Setup

Baseline. In section 6.2, we compare a 2D filter sparse convolution generated from TACO-UCF with oneDNN (Georganas et al., 2018), which is a set of expert-coded primitives for dense convolutions. We also compare TACO-UCF with SkimCaffe (Park et al., 2016), which is a hand-optimized filter sparse convolution library. For the GPU experiments, we compared TACO-UCF against cuDNN-8.3.2 (Chetlur et al., 2014), the dense counterpart of convolution, and Escort (Chen, 2018), a hand-optimized filter-sparse convolution on the GPU. In section 6.3, we compare a 3D masked convolution generated from TACO-UCF with state-of-the-art libraries, MinkowskiEngine(Choy et al., 2019)

	H	R	C	M	STR		H	R	C	M	STR
C1	112	1	64	256	2	**C2**	56	1	64	64	1
C3	56	1	256	64	1	**C4**	56	3	64	64	1
C5	56	1	64	256	1	**C6**	56	1	256	512	2
C7	56	1	256	128	1	**C8**	56	3	128	128	2
C9	28	1	512	128	1	**C10**	28	3	128	128	1
C11	28	1	128	512	1	**C12**	28	1	512	1024	2
C13	28	1	512	256	1	**C14**	28	3	256	256	2
C15	14	1	256	1024	1	**C16**	14	1	1024	256	1
C17	14	3	256	256	1	**C18**	14	1	256	1024	1
C19	14	1	1024	2048	2	**C20**	14	1	1024	512	1
C21	14	3	512	512	2	**C22**	7	1	512	2048	1
C23	7	1	2048	512	1	**C24**	7	3	512	512	1
C25	7	1	512	2048	1						

Table 3. Configurations of 2D convolution layers that has a unique shape in ResNet50 (He et al., 2016). All shapes of image and filter are square; $H = W$ and $R = S$. STR means a stride.

	Baseline	**80%**	**91%**	**96%**	**98%**
Top-1 Acc.	76.69%	76.52%	75.16%	72.71%	69.26%

Table 4. Top-1 accuracy of pruned ResNet50 (Gale et al., 2019) with different sparsity on ImageNet image classification task.

and TorchSparse(Tang et al., 2022), on both CPU and GPU.

Environment. All the CPU experiments are conducted on 12-core Intel Xeon E5-2680 v3 with `icc -O3 -march=native -mtune=native -qopenmp`. All the GPU experiements are conducted on Nvidia V100 32GB with CUDA-11.4 installed. All the experiments are done with FP32 and a single batch.

Dataset. There are two types of filter sparsity in network pruning: a structured sparsity and an unstructured sparsity. Structured network pruning sets the sparsity pattern in advance and prunes it accordingly. Unstructured network pruning sparsifies a filter without any specific pattern. We synthesized five previously studied structured patterns (Wen et al., 2016; Mao et al., 2017; Niu et al., 2020) as shown in Figure 7. For an unstructured sparsity, we used a pre-trained ResNet50 with 80%, 91%, 96%, and 98% sparsity from the prior work (Gale et al., 2019) which used a magnitude pruning. Table 4 reports the accuracies of pruned networks according to different sparsities.

In the evaluation of a 3D masked convolution, we used three indoor point clouds from S3DIS (Armeni et al., 2016) and two outdoor point clouds from SemanticKitti (Behley et al., 2019). We used a voxel size of 2cm and 5cm to quantize S3DIS and SemanticKitti dataset, respectively. The statistical properties of these datasets are shown in Table 5. We evaluate a single submanifold sparse convolutional layer with input and output channel size as 64 (C=M=64).

6.2 Filter sparse convolution

We evaluated filter space convolution kernels generated from TACO-UCF on CPU. We fixed the data representation of input and output feature map to be $N_U C_U H_U W_U$ and $N_U M_U P_U Q_U$, respectively.

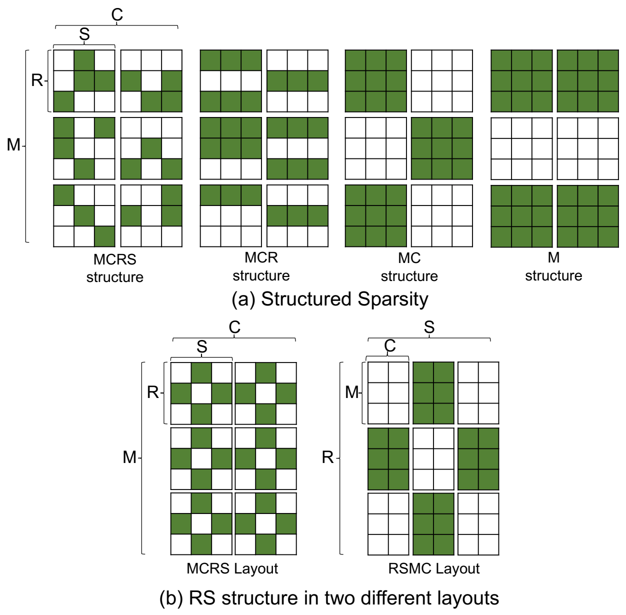


Figure 7. Five different structures of sparsity pattern in filter pruning. RS-structure exposes more regularity when it is stored in the RSMC layout than in the MCRS Layout.

	Name	H	W	D	nnz	$density$
Indoor (S3DIS)	Office	144	146	136	109,874	0.01%
	Lobby	369	232	154	293,855	0.01%
	Conference	296	526	143	585,064	3.84%
Outdoor (SemanticKITTI)	LIDAR1	2957	1859	154	87,164	2.23%
	LIDAR2	2991	1956	144	97,077	2.63%

Table 5. Dataset used to evaluate 3D masked convolution.

6.2.1 *Data Representation and Schedule*

Selecting a good data representation. We found out that choosing appropriate data representation of a sparse filter is important. Table 6 shows how different data representations changes the storage size of filter in the RS-structure. The RS-structure (Figure 7(b)) has the same $R \times S$ window's sparsity pattern for all m and c. If the filter $F(m, c, r, s)$ having RS-structure is stored in MCRS layout, the same (r, s) coordinates will be redundantly stored under all m and c. Thus, MCRS layout actually consumes more memory than the dense representation up to 67% sparsity despites the reduced number of nonzeros after the pruning. The natural layout and formats for RS-structure is $R_C S_C M_U C_U$ representation. $R_C S_C M_U C_U$ only stores pruned coordinates, (r, s), explicitly without storing any unnecessary zeros. Figure 7(b) shows that RS-structure stored in RSMC layout exhibits more regularity than MCRS layout. The last columns of the table shows that $R_C S_C M_U C_U$ data presentation saves memory as much as pruned (9×).

Selecting a good schedule. It is necessary to find not only a good data representation but also a good schedule to have good performance. To see how the schedule affects the per-

	Layout	MCRS		RSMC	
Sparsity	Format	$CCCC$	$UUCC$	$CCCC$	$CCUU$
56%	Size	1049KB	983KB	528KB	**262KB**
	vs. U^4	0.56×	0.60×	1.12×	**2.25×**
67%	Size	787KB	720KB	396KB	**196KB**
	vs. U^4	0.75×	0.82×	1.49×	**3.00×**
78%	Size	525KB	458KB	264KB	**131KB**
	vs. U^4	1.12×	1.29×	2.23×	**4.50×**
89%	Size	394KB	327KB	132KB	**65KB**
	vs. U^4	1.50×	1.80×	4.46×	**9.00×**

Table 6. Storage size of filter in layer C10 (M=C=128, R=S=3) in RS-structured sparsity pattern. vs. U^4 shows a memory save over a dense representation, all levels in Uncompressed formats.

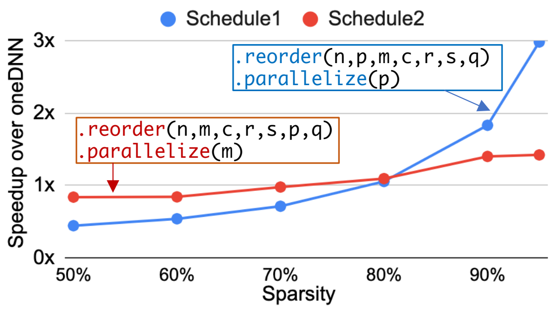


Figure 8. Speedup over oneDNN of two different schedules for layer C10 by increasing the sparsity in the M-structured sparsity.

formance, we compared the performance of two schedules at the layer C10 pruned in M-structure. Figure 8 illustrates the results. Schedule 1 parallelizes the computation by dividing the height of the output (p) across multiple cores, and Schedule 2 parallelizes the output channels (m). We store filter in the most natural representation with M-structure, $M_C C_U R_U S_U$. Because the loop order of Schedule 2 is concordant with the data layout of the output(NMPQ), it performs better than Schedule 1 at lower sparsity. However, as the sparsity increases, more output channels in M-structure get pruned. At 90% sparsity, only 13 output channels were left but the number is too small for multiple cores to process. On the other hand, Schedule 2 parallelizes the height of the output. Schedule 2 shows better performance than Schedule 1 at sparsity above 80% despite the loop discordantly proessing the output.

Since there are many choices to explore, we obtained the optimal data representation and schedule through an auto-tuning in section 6.2.2 and section 6.2.3. We used OpenTuner (Ansel et al., 2014) to choose the optimal parameter for the data representation, the order of the loop, a loop to parallelize, and OpenMP chunk size for load balancing. Our estimation of the size of the search space was about $2 * 10^4$, and we ran OpenTuner for 15 minutes for each layer.

6.2.2 *Structured Sparsity*

Figure 9 shows the performance of TACO-UCF on four different structured patterns. Overall, the better the pattern

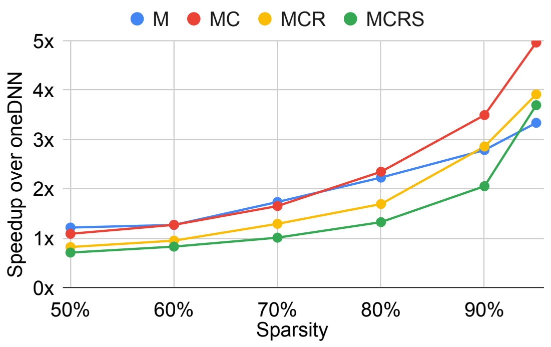


Figure 9. Speedup of four different sparse structures over oneDNN on layer C10. Formats and Schedules are found by OpenTuner.

Pruning Sparsity	80%	91%	96%	98%
cuDNN	1.0×	1.0×	1.0×	1.0×
Escort	0.78×	1.09×	1.35×	1.49×
TACO-UCF	**1.08×**	**1.61×**	**2.15×**	**2.57×**

Table 7. Normalized GPU performance relative to cuDNN of two filter sparse convolutions (Escort and TACO-UCF) on different sparsity levels of pruned ResNet50's convolutional layers.

was structured, the better the performance was. One observation is that MC-structure has comparable performance to M-structure and becomes faster after 80% sparsity. Even if MC-structure stores c in Compressed format, `icc` can generate an efficient register-blocked and vectorized kernel in the inner-most loop only with p, q, r, and s. A slowdown of the M-structure after 80% is because the M-structure does not expose sufficient parallelism, as shown in Figure 8.

6.2.3 *Unstructured Sparsity on CPU*

Not all pruning methods exploit structured sparsity. Instead, some methods prune the network without specific restrictions on patterns to preserve the accuracy as much as possible. Table 4 reports the top-1 accuracy of pruned models we used. Figure 10 compares TACO-UCF with SkimCaffe using the pruned ResNet80 at 80% sparsity. TACO-UCF shows similar or better performance with only 20 lines of code than SkimCaffe which is hand-written by experts over 488 lines of code. In addition, SkimCaffe does not support an efficient implementation of a stride convolution, so we only report TACO-UCF's performance at C21-C8 and C1.

While TACO-UCF supports a stride convolution, its performance was slower than oneDNN at 80% sparsity. We raised the sparsity of only the slower layers at 80% sparsity to test which sparsity makes the layer quicker than the dense counterpart. Figure 11 illustrates the results. TACO-UCF showed similar performance to oneDNN for most of layers at 91% sparsity and became faster as sparsity increases. Overall, when looking at all layers, TACO-UCF showed similar performance to oneDNN at the 80% sparsity. However, to have better performance, layers can be adaptively deployed with TACO-UCF or oneDNN.

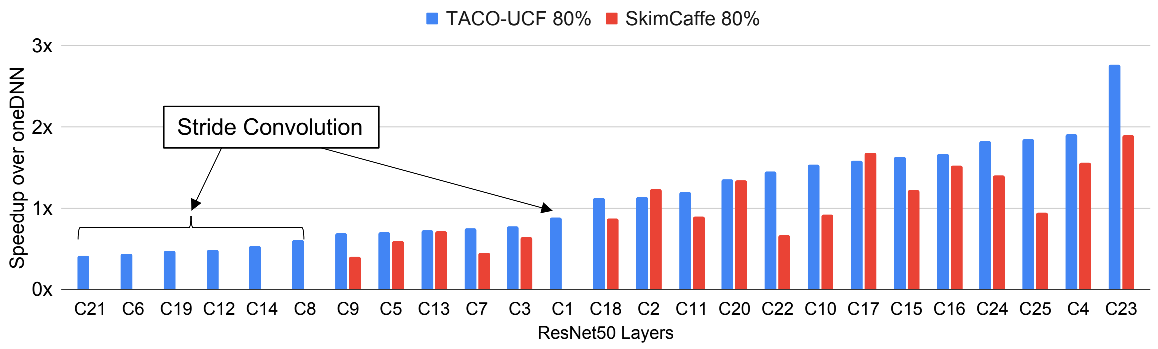


Figure 10. Speedup of TACO-UCF and SkimCaffe over oneDNN on ResNet50 layers at 80% sparsity. X-axis is sorted according to speedup of TACO-UCF. Data representations and Schedules are found by OpenTuner for each layer. We did not include SkimCaffe at C21-C8 and C1 because SkimCaffe does not support an efficient stride sparse convolution.

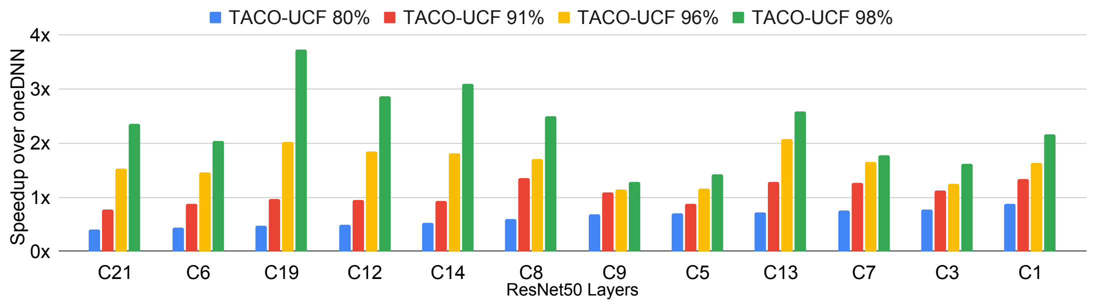


Figure 11. Speedup of TACO-UCF over oneDNN on layers that are slower than oneDNN in Figure 10 by increasing the sparsity.

6.2.4 *Unstructured Sparsity on GPU*

We conducted the same experiments as in Section 6.2.3 on the GPU. Unlike the CPU experiment, we used a fixed format and schedule for all layers in ResNet50 on the GPU. Table 7 shows the normalized performance relative to cuDNN. cuDNN performs dense convolution regardless of the sparsity level, so it serves as a baseline comparison. TACO-UCF shows higher performance compared to Escort and cuDNN at all sparsity levels of pruned ResNet50. While Escort, the state-of-the-art sparse convolution library, was able to match cuDNN's performance at 91% sparsity, TACO-UCF has now surpassed that threshold and is now faster than cuDNN at 80% sparsity. Moreover, TACO-UCF achieves even higher performance gains as the sparsity level increases.

6.3 Submanifold Sparse Convolution

6.3.1 *Direct Sparse Submanifold Convolution*

We implement a direct sparse submanifold convolution using TACO-UCF. We compare TACO-UCF with two baselines, MinkowskiEngine and TorchSparse, which implement the state-of-the-art map-based submanifold convolution.

CPU Result. Figure 12(a) reports the speedup of TACO-UCF's submanifold convolution over MinkowskiEngine at five different datasets on CPU. We did not report a CPU performance of TorchSparse because it was an order of magnitude slower than MinkowskiEngine (50× slower on average). TACO-UCF significantly outperforms

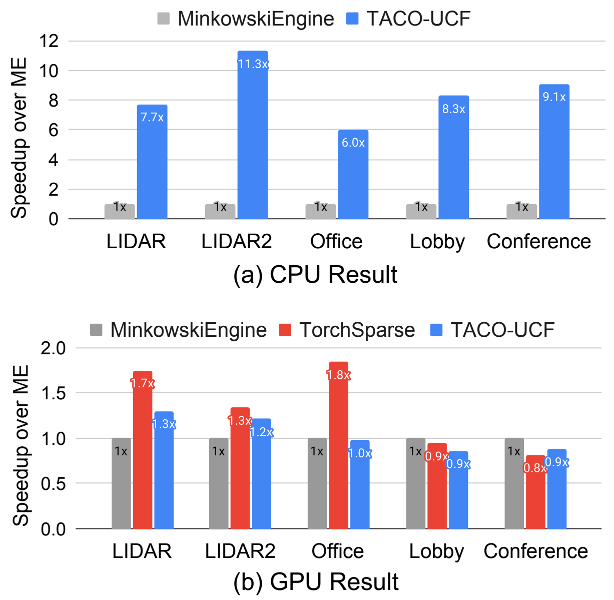


Figure 12. Normalized performance over MinkowskiEngine on CPU and GPU. In and out channels are C=M=64.

MinkowskiEngine. Existing libraries have yet to make efforts to optimize the CPU performance, and direct sparse convolution produces the neighbor map on the fly without having to write full intermediate results into the memory.

GPU Result. Figure 12(b) shows the results on GPU. Since the number of neighbors of each point significantly varies in

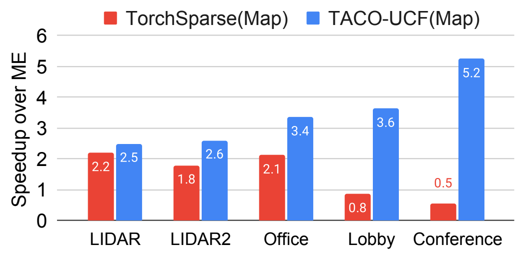


Figure 13. Normalized performance of each method over MinkowskiEngine in the neighbor map construction phase.

(a) CPU	Non-dual	RST-Structured			Unstructured		
Sparsity	0%	25%	50%	75%	90%	95%	99%
LIDAR	1.0×	1.4×	1.6×	2.5×	1.0×	1.0×	1.1×
LIDAR2	1.0×	1.2×	1.4×	1.9×	1.0×	1.1×	1.3×
Office	1.0×	1.4×	1.8×	2.5×	0.7×	0.8×	1.2×
Lobby	1.0×	1.5×	1.8×	3.1×	0.9×	1.0×	1.4×
Conference	1.0×	1.7×	2.0×	3.4×	0.9×	1.1×	1.3×

(b) GPU	Non-dual	RST-Structured			Unstructured		
Sparsity	0%	25%	50%	75%	90%	95%	99%
LIDAR	1.0×	1.1×	1.3×	1.9×	0.7×	1.1×	1.9×
LIDAR2	1.0×	1.3×	1.6×	2.1×	0.9×	1.5×	2.4×
Office	1.0×	1.3×	1.8×	2.9×	0.9×	1.4×	2.5×
Lobby	1.0×	1.3×	1.6×	2.6×	0.8×	1.3×	3.0×
Conference	1.0×	1.2×	1.5×	2.6×	0.8×	1.3×	2.8×

Table 8. Performance comparison of normalized dual sparse submanifold convolution of TACO-UCF over a normal sparse submanifold convolution of TACO-UCF (depicted as ”Non-dual” in the table) on (a) CPU and (b) GPU. The numbers were collected by sweeping a filter pruning sparsity in two different sparsity patterns.

the real-world point cloud, a load balancing scheme that utilizes a lot of cores is important to have a good performance in GPU. Map-based approaches can distribute workloads in a more balanced manner than a direct approach by looking at the distribution of neighbors created in the map construction phase. TACO-UCF has comparable performance against MinkowskiEngine, but is not as good as TorchSparse on some data. The slowdown is because of an imbalance each aggregation GPU thread handles in a direct convolution approach which finds neighbors on the fly.

6.3.2 *Map-based Sparse Submanifold Convolution*

We implement a map-based approach using TACO-UCF. Instead of implementing an entire map-based approach, we only implement the map construction (a neighbor retrieval) phase. A construction phase takes a considerable portion of the computation. It takes 37.6-87.4% and 27.6-59.1% of the execution time in TorchSparse and MinkowskiEngine, respectively.

Figure 13 reports the normalized performance of the map construction of each method on GPU. TACO-UCF outperforms baselines because it does not require building any hash table to retrieve neighbors like other methods, but can efficiently retrieve neighbors by intersecting coordinates stored in Compressed format. In particular, Torchsparse uses the cuckoo hash table (Pagh & Rodler, 2004) to construct the neighbor map, which is effective for small number of non-zeros. Still, as *nnz* increases in the point cloud, e.g., Lobby or Conference, it slows down quickly, and the construction phase takes account of a large portion. Replacing their map construction with TACO-UCF will significantly improve overall performance of the library.

6.4 Dual Sparse Convolution

Table 1 showcases the capabilities of TACO-UCF in generating code for dual sparse convolution where both the input activation and the filter are sparse. Specifically, we demonstrate its performance in two types of dual sparse convolution: (1) dual sparse conventional convolution, and (2) dual sparse submanifold convolution. In the following experiments, we compare dual sparse convolutions against non-dual sparse convolutions generated by TACO-UCF.

6.4.1 *Conventional Convolution with Dual Sparsity*

Dual sparse conventional convolution is in contrast to section 6.2, which only assumes sparsity on the filter. In our experiments, we investigated the performance of dual sparse convolution on CPU by varying the unstructured sparsity level of the input activation while fixing the filter sparsity at 80% on ResNet50. The input activation and filter weight were stored in $N_U H_U W_U C_U$ and $R_U S_U C_U M_C$ fixed format, respectively. The results showed that the generated code for dual sparse convolution was 1.6×, 1.9×, 2.4×, and 3.2× faster than filter-only sparse convolution at 50%, 60%, 70%, and 80% activation sparsity. However, it should be noted that when we tested dual sparse convolution with $NCHW$ layout, its performance was often worse compared to filter-only sparse convolution unless the filter weight had structured sparsity.

6.4.2 *Submanifold Convolution with Dual Sparsity*

The dual sparse submanifold convolution introduces filter sparsity to the normal submanifold sparse convolution. We conducted tests on two types of structured filter sparsity: (1) RST-structure filter sparsity, where we prune a spatial dimension of the filter (a 3D version of the RS-structure shown in Figure 7-(b)), and (2) unstructured filter sparsity. Table 8 presents the normalized performance over a submanifold sparse convolution without filter pruning. For both CPU and GPU, the dual sparse convolution achieves higher performance gains as the sparsity level increases. Similar to Figure 9, TACO-UCF can benefit from structured sparsity. Although we need to prune the filter until 95% sparsity with unstructured pattern to match the performance of non-dual sparse convolution, structured sparsity can easily achieve speedup compared to unstructured sparsity.

7 Conclusion

This paper introduces the Unified Convolution Framework (UCF) that integrates various existing sparse convolutions by extending TACO's Einsum language to express full affine index expressions. The resulting compiler efficiently generates code by exploring scheduling and format choices, achieving better performance than state-of-the-art libraries on filter, submanifold, and dual sparse convolution on both CPU and GPU. Although UCF can support many sparse convolutions that are not currently used in practice, we hope that this work will encourage their use. Additionally, the high-performance code generation of UCF offers practitioners the opportunity to develop bespoke convolution techniques that are better suited to their specific problems.

Acknowledgement

We thank anonymous reviewers for their valuable suggestions. We thank Teodoro Collin for reading early drafts of this paper and providing feedback. This work was supported by NSF/Intel PPoSS under award CCF-2217064; NSF under award CCF-2107244; and DOE SciDAC under award DE-SC0018121. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

Armeni, I., Sener, O., Zamir, A. R., Jiang, H., Brilakis, I., Fischer, M., and Savarese, S. 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1534–1543, 2016.

Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–205. IEEE, 2019.

Behley, J., Garbade, M., Milioto, A., Quenzel, J., Behnke, S., Stachniss, C., and Gall, J. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9297–9307, 2019.

Bik, A., Koanantakool, P., Shpeisman, T., Vasilache, N., Zheng, B., and Kjolstad, F. Compiler support for sparse tensor computations in mlir. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(4):1–25, 2022.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

Chen, X. Escoin: Efficient sparse convolutional neural network inference on gpus. *arXiv preprint arXiv:1802.10280*, 2018.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Chou, S., Kjolstad, F., and Amarasinghe, S. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA): 1–30, 2018.

Choy, C., Gwak, J., and Savarese, S. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3075–3084, 2019.

Elsen, E., Dukhan, M., Gale, T., and Simonyan, K. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14629–14638, 2020.

Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D., Henry, G., Pabst, H., and Heinecke, A. Anatomy of high-performance deep learning convolutions on simd architectures. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 830–841. IEEE, 2018.

Gong, Z., Ji, H., Fletcher, C. W., Hughes, C. J., and Torrellas, J. Sparsetrain: Leveraging dynamic sparsity in software for training dnns on general-purpose simd processors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 279–292, 2020.

Graham, B., Engelcke, M., and Van Der Maaten, L. 3d semantic segmentation with submanifold sparse convolutional networks. In *Proceedings of the IEEE conference* *on computer vision and pattern recognition*, pp. 9224–9232, 2018.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Henry, R., Hsu, O., Yadav, R., Chou, S., Olukotun, K., Amarasinghe, S., and Kjolstad, F. Compilation of sparse array programming models. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–29, 2021.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Hu, Y., Li, T.-M., Anderson, L., Ragan-Kelley, J., and Durand, F. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):1–16, 2019.

Kjolstad, F., Kamil, S., Chou, S., Lugato, D., and Amarasinghe, S. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 1–29, 2017.

Kurtz, M., Kopinsky, J., Gelashvili, R., Matveev, A., Carr, J., Goin, M., Leiserson, W., Moore, S., Shavit, N., and Alistarh, D. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning*, pp. 5533–5543. PMLR, 2020.

Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. Exploring the granularity of sparsity in convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 13–20, 2017.

Niu, W., Ma, X., Lin, S., Wang, S., Qian, X., Lin, X., Wang, Y., and Ren, B. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 907–922, 2020.

Pagh, R. and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., and Dubey, P. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6): 519–530, 2013.

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

Tang, H., Liu, Z., Li, X., Lin, Y., and Han, S. Torchsparse: Efficient point cloud inference engine. *Proceedings of Machine Learning and Systems*, 4:302–315, 2022.

Tian, R., Guo, L., Li, J., Ren, B., and Kestor, G. A high performance sparse tensor algebra compiler in mlir. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 27–38. IEEE, 2021.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems*, 29, 2016.

Yu, F. and Koltun, V. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

A Artifact Appendix

A.1 Abstract

The artifact provided in this paper consists of two directories that run experiments for filter sparse convolution and masked convolution. The experiments were conducted on Ubuntu running on Intel Xeon CPU E5-2680 v3 and Nvidia RTX 3090.

The artifact includes TACO-UCF, which implements the affine index expression support described in the paper, as well as a set of benchmark scripts to run the experiments described in the paper. The artifact can be found on Zenodo and GitHub.

A.2 Artifact check-list (meta-information)

- **Algorithm: Unified Convolution Framework**
- **Program: TACO-UCF**
- **Compilation: TACO-UCF is compiled with cmake. Codes generated by TACO-UCF are then compiled with gcc, icc, and nvcc.**
- **Data set: (1) Pruned ResNet50, (2) Two point clouds from SemanticKITTI, and (3) Three point clouds from S3DIS**
- **Run-time environment: Ubuntu 18.04 LTS**

- **Hardware: Intel Xeon CPU E5-2680 v3 and Nvidia RTX 3090**
- **Metrics: Elapsed Time (ms)**
- **Output: Elapsed time of each program**
- **Experiments: (1) Filter sparse convolution and (2) Submanifold sparse convolution**
- **Publicly available DOI: https://doi.org/10.5281/zenodo.7858518**
- **Publicly available Github: https://github.com/nullplay/Unified-Convolution-Framework**
- **Code licenses (if publicly available)?: MIT License**
- **Data licenses (if publicly available)?: CC BY-NC-SA 4.0**

A.3 Description

A.3.1 How delivered

Clone the repository from GitHub. Then follow the installation instruction in README.md.

A.3.2 Hardware dependencies

Some experiments in the artifact requires Nvidia GPU with a compute capability 6.1 or higher to run.

A.4 Installation

Building TACO-UCF using CMake 3.4.0 or greater

To build TACO-UCF, follow the steps below:

1. `cd <UCF-directory>`
2. `mkdir build`
3. `cd build`
4. `cmake -DCMAKE_BUILD_TYPE=Release ..`
5. `make -j8`
6. `export LD_LIBRARY_PATH= $(pwd)/lib/:$LD_LIBRARY_PATH`

Building for CUDA

To build TACO-UCF for NVIDIA CUDA, add the `-DCUDA=ON` flag to the CMake command above. For example:

```
cmake -DCMAKE_BUILD_TYPE=Release
      -DCUDA=ON ..
```

Please ensure that you have CUDA installed properly and that the following environment variables are set correctly:

```
export PATH=
  /usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=
  /usr/local/cuda/lib64:$LD_LIBRARY_PATH
export LIBRARY_PATH=
  /usr/local/cuda/lib64:$LIBRARY_PATH
```

A.5 Experiment workflow

Running Sparse Convolutions

To run filter-sparse convolution, navigate to the following directory:

```
cd ./benchmark/filter_sparse_convolution
```

To run masked (submanifold) sparse convolution, navigate to the following directory:

```
cd ./benchmark
    /submanifold_sparse_convolution
```

A.6 Evaluation and expected result

If the elapsed time of each program is displayed, the experiment ran successfully.