Reduction-Aware Directive-Based Programming via Multi-Dimensional Homomorphisms

Richard Schulze
University of Muenster
Münster, Germany
r.schulze@uni-muenster.de

Sergei Gorlatch
University of Muenster
Münster, Germany
gorlatch@uni-muenster.de

Ari Rasch
University of Muenster
Münster, Germany
a.rasch@uni-muenster.de

Abstract

Directive-based programming models have established themselves as an effective and productive paradigm for exploiting parallel architectures such as GPUs and CPUs. Widely used solutions—such as OpenMP and OpenACC—are popular due to their simplicity and broad applicability to general-purpose codebases. However, these approaches often struggle to deliver consistently high and portable performance, especially for reduction-intensive computations.

We introduce a novel directive design grounded in the formalism of *Multi-Dimensional Homomorphisms (MDH)*. Unlike existing directive-based methods, our MDH-based directive is explicitly crafted for data-parallel computations (such as tensor expressions), enabling superior and portable performance even on reduction-heavy workloads. At the same time, our approach preserves and often enhances programmer productivity, e.g., by leveraging Python as the host language.

Our experiments across diverse workloads—including linear algebra, stencil computations, data mining, quantum chemistry, and deep learning—show that our approach not only surpasses state-of-the-art directive-based methods but also achieves up to 5× speedups on CPUs and over 2× on GPUs compared to highly optimized vendor libraries Intel MKL/oneMKL and NVIDIA cuBLAS/cuDNN.

CCS Concepts

• **Software and its engineering → Parallel programming languages**; **Compilers**; • **Computer systems organization → Parallel architectures**.

Keywords

directives, reductions, multi-dimensional homomorphisms, Python

ACM Reference Format:
Richard Schulze, Sergei Gorlatch, and Ari Rasch. 2025. Reduction-Aware Directive-Based Programming via Multi-Dimensional Homomorphisms. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3731599.3767572

1 Introduction

Data-parallel computations cover a broad range of fundamental operations, from linear algebra routines to stencil computations, and represent performance-critical building blocks in domains such as deep learning, scientific computing, and data analytics. Consequently, achieving high performance for these computations—across diverse architectures (such as GPUs and CPUs) and varying data characteristics (e.g., size and memory layout)—remains a central research challenge, as the efficiency of important application areas critically depends on the performance of their data-parallel building blocks.

Directive-based approaches—ranging from established standards like OpenACC [27] and OpenMP [28], to polyhedral compilers such as PPCG [43] and Pluto [9], the increasingly adopted Numba [24]—provide directive-based mechanisms for generating high-performing GPU and CPU code from straightforward sequential code: users annotate their code with so-called *directives* which are simple program annotations that express opportunities for optimization and parallelization, thereby freeing the user from hardware details and manual low-level code optimizations.

While current directive-based approaches are widely adopted in the community—due to their ease of use and often wide applicability to general-purpose code—we argue in this paper that they leave room for improvement in terms of performance and portability[1]. In particular, our experiments show that for reduction-heavy computations, state-of-the-art approaches appear to be suboptimal—often even for simple computations such as dot products.

The formalism of *Multi-Dimensional Homomorphisms (MDH)* [32] was recently introduced as an algebraic framework for reasoning about and optimizing data-parallel computations. Its corresponding implementation has demonstrated the ability to generate high-performance, portable code—e.g., in CUDA for GPUs and in OpenCL for CPUs—from an optimization- and hardware-agnostic MDH-based *Domain-Specific Language (DSL)*. Despite abstracting away low-level concerns, DSL-based programming can still pose a barrier to domain scientists—such as chemists, physicists, or AI engineers—who may be unfamiliar with such abstractions.

This paper introduces a new directive-based approach grounded in the formalism of MDH. In contrast to existing solutions, our MDH directive intentionally specializes in data-parallel computations rather than general-purpose code[2], enabling significantly higher and more portable performance within this specific domain.

[1]By *portability*, we refer not only to *functional portability*—the ability to run the same source code on different hardware architectures—but also to *performance portability*, meaning that the code maintains consistently high performance across architectures and data characteristics.

[2]According to MDH [32], we consider a computation data-parallel iff expressible as:

$$\underset{i_1 \in I_1}{\circledast_1} \dots \underset{i_D \in I_D}{\circledast_D} f(a[i_1, \dots, \text{ß}_D])$$

for an arbitrary function f and operators $\circledast_d$ (as illustrated by examples later).

Furthermore, our approach is based on Python (unlike many existing frameworks that often rely on C++), thereby contributing to high user productivity. Our experimental evaluation across a range of application domains—including linear algebra, stencil computations, data mining, quantum chemistry, and deep learning—demonstrates that our generated code achieves competitive—and in some cases superior—performance compared to hand-optimized implementations provided by highly tuned vendor libraries from NVIDIA and Intel.

The remainder of this paper is organized as follows. Section 2 analyzes strengths and weaknesses of directive-based approaches for reductions. Section 3 summarizes the MDH formalism, and Section 4 introduces our MDH-based directive approach. Section 5 presents experimental results, Section 6 reviews related work, Section **??** concludes, and Section 8 outlines future directions.

2 State-of-the-Art Directive-Based Approaches

We review state-of-the-art directive-based approaches—PPCG/Pluto, OpenMP/OpenACC, and Numba—focusing on their handling of reduction computations, illustrated using the example of *Matrix-Vector Multiplication (MatVec)*.

PPCG/Pluto. Listing 1 illustrates how MatVec is optimized by polyhedral compilers—PPCG for GPUs and Pluto for CPUs—using the same annotated sequential C input.

We consider PPCG and Pluto highly productive, as they require only simple annotations of the target region (lines 6 and 13). However, this productivity often comes at a performance cost, particularly in reduction-heavy computations: reduction operators (such as + in the k-loop of MatVec in line 9) are not explicitly represented in the directive and thus remain unavailable for optimization. [3]

```
1   #define I  /* define I */
2   #define K  /* define K */
3   #define T float
4
5   void matvec_poly(const T *M, const T *v, T *w) {
6     #pragma scop
7     for (int i = 0; i < I; i++) {
8       w[i] = 0.0f;
9       for (int k = 0; k < K; k++) {
10        w[i] += M[i * K + k] * v[k];
11      }
12    }
13    #pragma endscop
14  }
```

Listing 1: PPCG/Pluto program for MatVec (C)

OpenMP. Listing 2 illustrates how MatVec is optimized for CPUs using OpenMP. The user annotates the sequential C++ implementation with two simple directives (lines 7 and 11), allowing the OpenMP compiler to automatically generate parallel code for MatVec. The directive in line 11 explicitly informs the compiler that the intermediate results of the k-loop should be combined using addition (+) as the reduction operator, enabling safe parallelization and optimization of the reduction across loop iterations.

[3]While reductions in simple cases (e.g., MatVec) may be inferred automatically, more complex scenarios (such as PRL in Listing 11) complicate static detection due to undecidability (Rice's theorem).

Compared to the input program used by polyhedral compilers (Listing 1), the OpenMP implementation requires a small modification to support efficient parallelization: a local variable sum must be introduced (lines 9, 11, 13, and 15) to hold intermediate results.

```
1   #define I  /* define I */
2   #define K  /* define K */
3   #define T float
4
5   void matvec_openmp(const T* M, const T* v, T* w)
6   {
7     #pragma omp parallel for
8     for (int i = 0; i < I; ++i) {
9       T sum = 0.0f;
10
11      #pragma omp simd reduction(+:sum)
12      for (int k = 0; k < K; ++k) {
13        sum += M[i * K + k] * v[k];
14      }
15      w[i] = sum;
16  } }
```

Listing 2: OpenMP program for MatVec (C++)

OpenACC. Listing 3 illustrates how MatVec is optimized for GPUs using OpenACC. The OpenACC implementation closely resembles the OpenMP version for CPUs (Listing 2), with the primary difference found in lines 7–8 of Listing 3: the OpenACC code includes explicit data transfers (via copyin and copyout) to manage data transfers between host and GPU memory.

```
1   #define I  /* define I */
2   #define K  /* define K */
3   #define T float
4
5   void matvec_openacc(const T* M, const T* v, T* w)
6   {
7     #pragma acc data copyin(M[0:I*K], v[0:K]) \
8                      copyout(w[0:I]) {
9       #pragma acc parallel loop
10      for (int i = 0; i < I; ++i) {
11        T sum = 0.0f;
12
13        #pragma acc loop reduction(+:sum)
14        for (int k = 0; k < K; ++k) {
15          sum += M[i * K + k] * v[k];
16        }
17        w[i] = sum;
18  } } }
```

Listing 3: OpenACC program for MatVec (C++)

Numba. Listing 4 presents a CPU-optimized version of MatVec using Numba. Although Numba also supports GPU offloading (see Listing 5), this requires a distinct, GPU-specific implementation using constructs such as cuda.grid (as in line 4 of Listing 5).

Numba uses Python as its host language, which—unlike the C/C++-based approaches in Listings 1–3—enhances productivity by leveraging Python's simplicity and broad adoption [1].

From Listing 4, we observe that similarly to polyhedral compilers PPCG and Pluto (Listing 1), Numba requires only minimal code annotations (line 2 in Listing 4). Consequently, Numba offers productivity comparable to, or even exceeding, that of PPCG/Pluto by using Python as its host language. However, it faces similar limitations as PPCG/Pluto when it comes to reduction-based computations: for example, parallelizing the reduction dimension (line 6) would require explicitly managing parallelism and synchronization, which diminishes the simplicity and ease of use that Numba typically provides.[4]

```
1  def matvec_numba_cpu(I, K):
2      @njit(parallel=True)
3      def matvec__I_K(w, M, v):
4          for i in range(I):
5              w[i] = 0.0
6              for k in range(K):
7                  w[i] += M[i, k] * v[k]
8      return matvec__I_K
```

Listing 4: Numba program for MatVec on CPU (Python)

```
1  def matvec_numba_gpu(I, K):
2      @cuda.jit
3      def matvec__I_K(w, M, v):
4          i = cuda.grid(1)
5          if i < I:
6              for k in range(K):
7                  w[i] += M[i, k] * v[k]
8      return matvec__I_K
```

Listing 5: Numba program for MatVec on GPU (Python)

3 The MDH Approach

We recapitulate the MDH approach which offers a formalism [32] for expressing data-parallel computations—called *High-Level Program Representation* in MDH—based on the algebraic properties of these computations[5]. We briefly summarize the high-level program representation of MDH, using the example of *Matrix-Vector Multiplication (MatVec)*.

```
1  def matvec( T:BasicType, I:int,K:int ):
2    @mdh()
3    def matvec__T_I_K():
4      return (
5        out_view[T]( w = [lambda i,k: (i)] ),
6        md_hom[I,K]( f_mul, (cc,pw(add)) ),
7        inp_view[T,T]( M = [lambda i,k: (i,k)] ,
8                       v = [lambda i,k: (k)  ] )
9      )
10   return matvec__T_I_K
```

Listing 6: MatVec expressed in MDH DSL (Python)

Listing 6 illustrates how MatVec is expressed in MDH's DSL, which is embedded in Python and implements the formal, high-level program representation of MDH. The computation takes as input a matrix `M` (line 7) and a vector `v` (line 8), both of type `T` (e.g., `fp32`), with sizes `I` × `K` (matrix) and `K` (vector), for `I,K` ∈ ℕ (line 6).

Input accesses are captured via the higher-order function `inp_view` (lines 7–8), which maps iteration-space indices to buffer indices: `(i,k)↦(i,k)` for the matrix and `(i,k)↦(k)` for the vector.

The data-parallel computation is expressed via `md_hom` (line 6), which applies `f_mul` (multiplication) to each `(M[i,k],v[k])`, concatenates results along the *i*-dimension using `cc`, and reduces over the *k*-dimension with `pw(add)`.

Output accesses are captured by function `out_view` (line 5). While straightforward here, it can express variants such as strided outputs (via the index function $(i,k) \mapsto (i * s)$ for stride $s \in \mathbb{N}$) or transposed layouts in other computations, etc.

A notable feature of MDH is its flexible handling of reduction operators. It supports user-defined reduction operators, such as point-wise operator $\texttt{pw}(\vec{\bullet})$ for arbitrary functions $\vec{\bullet} : \texttt{T} \times \texttt{T} \to \texttt{T}$, and fully custom operators [32] like *prefix sum*, which—unlike point-wise operations—preserve the size of the reduction dimension instead of collapsing it to a single element.

```
@mdh( ... )
def IDF ():
  return (
    out_view[ BSC_TYP ,..., BSC_TYP ](
             \_________________________/
                      #OB-times
       IDF   =  [ IDF_FNC ,..., IDX_FNC ]          \
                  \____________________/            |
                       #ACC^OB_1-times              |
                            ⋮                       } #OB-times
       IDF   =  [ IDX_FNC ,..., IDX_FNC ] ) ,       |
                  \____________________/            |
                       #ACC^OB_#OB-times           /

    md_hom[ SIZE ,..., SIZE ]( SF , ( CO ,..., CO ) ),
           \________________/        \___________/
                D-times                 D-times

    inp_view[ BSC_TYP ,..., BSC_TYP ](
             \_________________________/
                      #IB-times
       IDF   =  [ IDX_FNC ,..., IDX_FNC ]          \
                  \____________________/            |
                       #ACC^IB_1-times              |
                            ⋮                       } #IB-times
       IDF   =  [ IDX_FNC ,..., IDX_FNC ] )         |
                  \____________________/            |
                       #ACC^IB_#IB-times           /
  )
```

Listing 7: General structure of the MDH DSL (Python). Flexible parts are highlighted in gray.

Listing 7 illustrates the generic structure of the MDH DSL. In the listing, `#IB` and `#OB` generically represent the number of input and output buffers, respectively. The terms $\texttt{\#ACC}^{\texttt{IB}}_{\texttt{b}}$ and $\texttt{\#ACC}^{\texttt{OB}}_{\texttt{b}}$ denote the number of accesses to the b-th input or output buffer. For instance, three accesses are counted when specifying `v = [lambda i,k: (k+0), lambda i,k: (k+1), lambda i,k: (k+2)]`, as typically used in a 3-point stencil computation (discussed later), where all three index expressions target the same buffer `v`.

In the following, we introduce our MDH-based directive and show how it can be translated into an MDH DSL program (Listing 7).

[4]Numba can automatically parallelize simple reductions in certain cases, but often skips them as soon as they become slightly more complex [26].
[5]The MDH formalism [32] also introduces a *Low-Level Program Representation* for expressing optimizations and a fully automatic *Lowering Process* that translates high-level MDH instances into device- and data-optimized low-level ones. These are not discussed here, as they are not relevant to the directive design focus of this paper.

This translation enables us to leverage the existing DSL-based MDH pipeline [32, 33, 35, 36] for parallel code generation, which automatically produces auto-tuned parallel code (e.g., for GPUs and CPUs) with optimizations such as tiling, data movement, and parallelization.

4 The MDH Directive

We first introduce our MDH-based directive for data-parallel computations through practical examples in Section 4.1. Afterwards, we discuss its general structure in Section 4.2, and finally, we demonstrate how to generate an MDH DSL program from it in Section 4.3.

4.1 Introductory Examples

To promote user productivity, our approach adopts Python as the host language. We implement our directive as a *Python decorator* (the same as Numba—line 2 in Listing 4) which is a native Python language construct for code annotations.

Linear Algebra. Listings 8 and 9 show how *Matrix-Vector Multiplication (MatVec)* and *Matrix Multiplication (MatMul)* are optimized using our MDH directive (lines 2–4 in Listing 8, and lines 2–4 in Listing 9).

```
1  def matvec( T:BasicType, I:int,K:int ):
2    @mdh( out( w = Buffer[T]                ) ,
3          inp( M = Buffer[T], v = Buffer[T] ) ,
4          combine_ops( cc, pw(add) )          )
5    def mdh_matvec__T_I_K( w, M,v ):
6      for i in range(I):
7        for k in range(K):
8          w[i] = M[i,k] * v[k]
9    return mdh_matvec__T_I_K
```

Listing 8: MatVec optimized via MDH Directive (Python)

Listing 8 illustrates how our MDH directive specifies the input and output buffers along with their corresponding basic type `T` (lines 2–3), e.g., `T=fp32`. In particular, our directive captures the reduction operators (line 4), which are for MatVec concatenation `cc` and point-wise addition `pw(add)`.[6]

A key design difference of our approach, compared to the existing methods in Section 2, lies in how the loop body is structured (line 8). In our approach, the loop body computes a single point in the iteration space without performing reductions—i.e., the aggregation across the iteration space is not encoded directly in the loop body (using = in line 8 of Listing 8, rather than += as in line 13 of Listing 2, for example). Instead, reductions are semantically captured and expressed through our directive. As a result, reduction operations (such as +=) are abstracted away from the loop body.

Although this separation may appear unconventional and require some adaptation, it offers a major advantage: nested reductions can be expressed naturally and concisely in our approach (as elaborated later in this section), whereas related approaches generally lack native support and require complex workarounds (also discussed later).

[6]Operators cc and pw are pre-implemented by our system due to their frequent use (implementations provided in the Appendix, Section A, for the interested reader).

Compared to OpenMP and OpenACC (Listings 2 and 3), our approach does not require additional temporary variables for intermediate results (such as `sum` in Listings 2 and 3) or zero-initializing result buffers (as Numba, see line 5 in Listing 4).

```
1  def matmul( T:BasicType, I:int,J:int,K:int ):
2    @mdh( out( C = Buffer[T]                ) ,
3          inp( A = Buffer[T], B = Buffer[T] ) ,
4          combine_ops( cc, cc, pw(add) )      )
5    def matmul__T_I_J_K( C, A,B ):
6      for i in range(I):
7        for j in range(J):
8          for k in range(K):
9            C[i,j] = A[i,k] * B[k,j]
10   return matmul__T_I_J_K
```

Listing 9: MatMul optimized via MDH Directive (Python)

Listing 9 shows our dirdective for MatMul (lines 2–4), which closely resembles the directive for MatVec in Listing 8 and thus reflects the natural similarity between the two operations. Apart from different buffer names (lines 2–5), the MatMul directive introduces an additional `cc` dimension (line 4) to account for the extra `j`-dimension in its iteration space.

Stencil Computations. Listing 10 shows our directive applied to computation *Jacobi (Jacobi1D)*. This example is relatively straightforward, as Jacobi1D operates over a regular one-dimensional iteration space that does not involve any reduction computations.

```
1  def jacobi1d( T:BasicType, I:int ):
2    @mdh( out( y = Buffer[T] ) ,
3          inp( x = Buffer[T] ) ,
4          combine_ops( cc )    )
5    def jacobi1d__T_I( y, x ):
6      for i in range(I):
7        y[i] = ( x[i+0] + x[i+1] + x[i+2] ) / 3.0
8    return jacobi1d__T_I
```

Listing 10: Jacobi1D optimized via MDH Directive (Python)

Data Mining. Listing 11 shows our directive for *Probabilistic Record Linkage (PRL)*—a popular example used in data mining to find duplicate entries in a database [34]. A key characteristic of PRL is its use of the point-wise reduction operator `pw` (line 26): instead of relying on a simple addition operator—as commonly used in linear algebra routines (e.g., in line 4 of Listing 4)—it employs a PRL-specific customization function (lines 6–19 in Listing 11).

Deep Learning. Listing 12 shows our directive for *Multi-Channel Convolution (MCC)*—a generalization of standard convolution commonly used in deep learning. In contrast to previous examples, MCC uses unconventional buffer sizes: according to MCC's usage in ResNet-50, the `img` buffer (lines 13–14) is artificially enlarged in the second and third dimension (lines 4–5)[7].

[7]If not explicitly specified (Listings 8–11), buffer sizes are automatically inferred from the iteration space and index functions.

MBBS. Listing 13 illustrates how *Maximum Bottom Box Sum (MBBS)* [14], which computes prefix sums over accumulated column vectors of a matrix, is optimized using our directive. Unlike previous examples, MBBS relies on the prefix sum reduction operator ps [32] (line 4) instead of cc and pw.[8]

```python
def prl( N:int,I:int ):
  dbl8  = { 'values':fp64[8]  }
  chr2  = { 'values':char[2]  }
  chr46 = { 'str':char[46] }

  @pw_custom_func()
  def prl_max( res:Scalar[int64,fp64,int32] ,
               lhs:Scalar[int64,fp64,int32] ,
               rhs:Scalar[int64,fp64,int32] ):
    if( lhs['id_measure'] == 14 and
        rhs['id_measure'] != 14 ):
      res['match_id']     = lhs['match_id']
      res['match_weight'] = lhs['match_weight']
      res['id_measure']   = lhs['id_measure']
    ...
    else:
      res['match_id']     = rhs['match_id']
      res['match_weight'] = rhs['match_weight']
      res['id_measure']   = rhs['id_measure']

  @mdh( out( match_id     = Buffer[int64] ,
             match_weight = Buffer[fp64]  ,
             id_measure   = Buffer[int32] ) ,
        inp( probM        = Buffer[dbl8]  ,
             ...   ) , # 64 further buffers
        combine_ops( cc, pw(prl_max) )      )
  def prl__N_I( match_id,match_weight,id_measure,
                probM,
                ... ): # 64 further buffers
    for n in range(N):
     for i in range(I):
      tmp_match_weight:fp64
      tmp_id_measure:int32
      # ... (tmp_match_weight & tmp_id_measure)
      match_id[n]     = i_id[i]
      match_weight[n] = tmp_match_weight
      id_measure[n]   = tmp_id_measure
  return prl__N_I
```

Listing 11: PRL optimized via MDH Directive (Python)

```python
def mcc(T:BasicType,N:int,P:int,Q:int,K:int,
                    R:int,S:int,C:int      ):
  @mdh( out( res=Buffer[T] ),
        inp( img=Buffer[T,[N,(2*P)+R-1,
                             (2*Q)+S-1,C]],
             flt = Buffer[T] ),
        combine_ops( cc,cc,cc,cc,
                     pw(add),pw(add),pw(add) ) )
  def mcc__T_N_P_Q_K_R_S_C( res, img,flt ):
    for n in range(N):
      ⋱
        for c in range(C):
          res[n,p,q,k] = img[n,(2*p)+r,
                             (2*q)+s,c] * \
                         flt[k,r,s,c]
  return mcc__T_N_P_Q_K_R_S_C
```

Listing 12: MCC optimized via MDH Directive (Python)

[8]Like cc and pw, the ps operator is pre-implemented by our system (see Appendix A).

```python
1  def mbbs( T:BasicType, I:int,J:int ):
2    @mdh( out( Out = Buffer[T] )          ,
3          inp( Inp = Buffer[T] )          ,
4          combine_ops( ps(add), pw(add) ) )
5    def mdh_mbbs__T_I_J( O, I ):
6      for i in range(I):
7        for j in range(J):
8          O[i] = I[i,j]
9    return mdh_mbbs__T_I_J
```

Listing 13: MBBS optimized via MDH Directive (Python)

4.2 MDH Directive: General Structure

Listing 14 illustrates the general structure of our MDH directive. It targets perfect loop nests with an arbitrary depth D. The loop body consists of an arbitrary but pure scalar function SF[9], mapping elements within input buffers to output buffers.

Inputs and outputs are declared via the inp(...) and out(...) clauses. We support an arbitrary number of input and output buffers—denoted generically as #IB and #OB in the listing—each identified by a user-defined, symbolic identifier IDF and a basic type BSC_TYP.

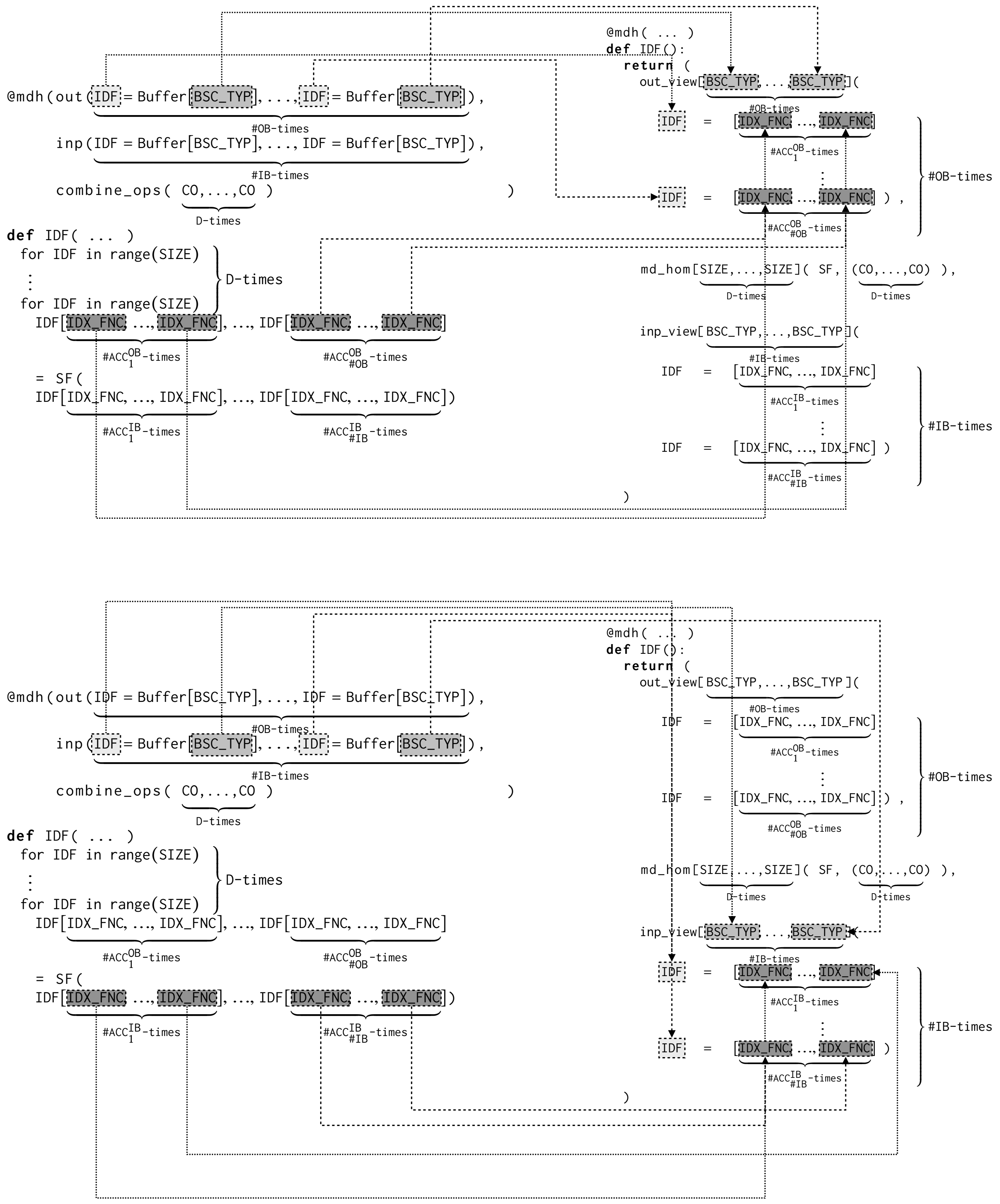
```
@mdh( out( IDF = Buffer[ BSC_TYP ],     ⎫
               ⋮                        ⎬ #OB-times,
             IDF = Buffer[ BSC_TYP ]),  ⎭
       inp( IDF = Buffer[ BSC_TYP ],    ⎫
               ⋮                        ⎬ #IB-times,
             IDF = Buffer[ BSC_TYP ]),  ⎭

          combine_ops( CO ,..., CO )
                       \____D-times____/
)
def IDF ( ... )
  for IDF in range( SIZE )  ⎫
   ⋮                        ⎬ D-times
  for IDF in range( SIZE )  ⎭

      IDF [ IDF_FNC ,..., IDF_FNC ],
            \__#ACC_1^OB-times__/
       ⋮
      IDF [ IDX_FNC ,..., IDX_FNC ]
            \__#ACC_#OB^OB-times__/
    = SF (  IDF [ IDX_FNC ,..., IDX_FNC ],
                  \__#ACC_1^OB-times__/
             ⋮
            IDF [ IDX_FNC ,..., IDX_FNC ])
                  \__#ACC_#OB^OB-times__/
```

Listing 14: General structure of the MDH Directive (Python). Flexible parts are highlighted in gray.

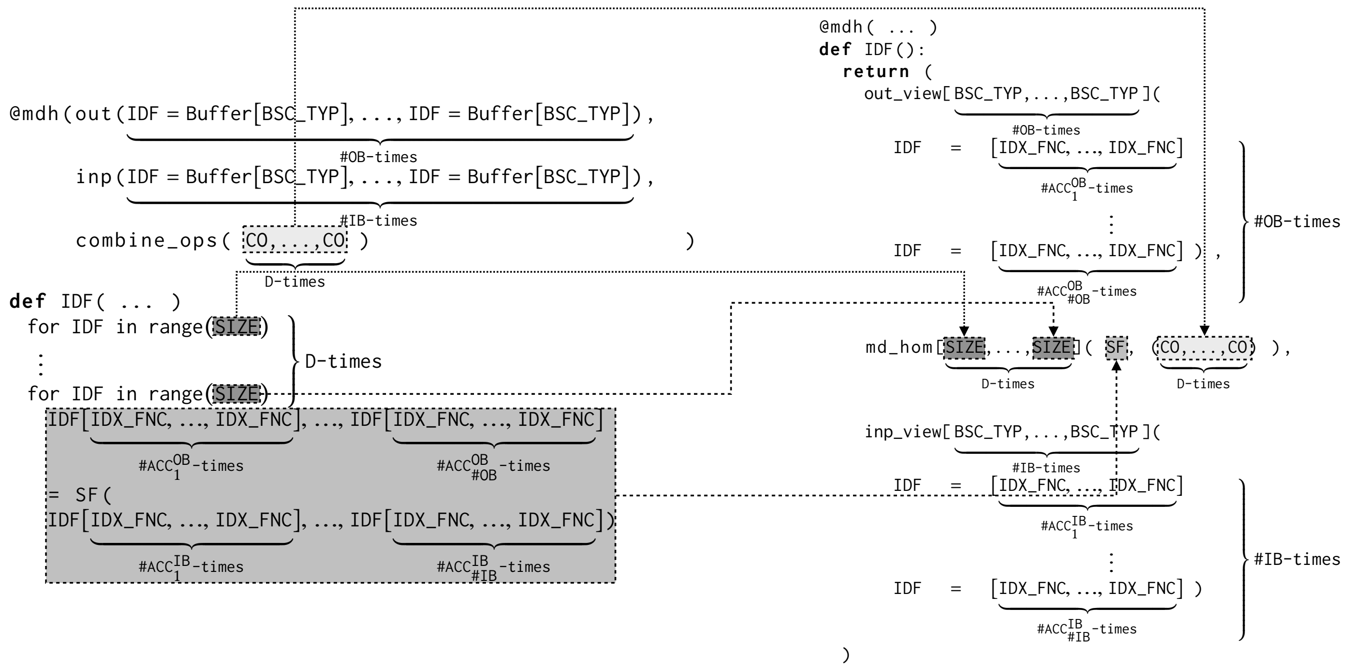
[9]We support exactly the same imperative-style program code as the MDH DSL [32], e.g., code involving loops, if statements, etc.



```
@mdh( out( IDF = Buffer[BSC_TYP], ..., IDF = Buffer[BSC_TYP] ),
                              #OB-times
      inp( IDF = Buffer[BSC_TYP], ..., IDF = Buffer[BSC_TYP] ),
                              #IB-times
      combine_ops( CO,...,CO )                                    )
                   D-times
def IDF( ... )
  for IDF in range(SIZE) ┐
   ⋮                     │ D-times
  for IDF in range(SIZE) ┘
    IDF[IDX_FNC, ..., IDX_FNC], ..., IDF[IDX_FNC, ..., IDX_FNC]
         #ACC_1^OB-times                #ACC_#OB^OB-times
    = SF(
    IDF[IDX_FNC, ..., IDX_FNC], ..., IDF[IDX_FNC, ..., IDX_FNC])
         #ACC_1^IB-times                #ACC_#IB^IB-times
```

```
@mdh( ... )
def IDF():
  return (
    out_view[BSC_TYP,...,BSC_TYP](
                #OB-times
      IDF  =  [IDX_FNC, ..., IDX_FNC]           ┐
               #ACC_1^OB-times                  │
                         ⋮                      │ #OB-times
      IDF  =  [IDX_FNC, ..., IDX_FNC] ) ,       │
               #ACC_#OB^OB-times                ┘
    md_hom[SIZE,...,SIZE]( SF, (CO,...,CO) ),
             D-times             D-times
    inp_view[BSC_TYP,...,BSC_TYP](
                #IB-times
      IDF  =  [IDX_FNC, ..., IDX_FNC]           ┐
               #ACC_1^IB-times                  │
                         ⋮                      │ #IB-times
      IDF  =  [IDX_FNC, ..., IDX_FNC] )         │
               #ACC_#IB^IB-times                ┘
  )
```

Figure 1: Transformation of an MDH directive into MDH-DSL (data accesses)



```
@mdh(out(IDF = Buffer[BSC_TYP], ..., IDF = Buffer[BSC_TYP]),
     inp(IDF = Buffer[BSC_TYP], ..., IDF = Buffer[BSC_TYP]),
     combine_ops( CO,...,CO )                                  )
def IDF( ... )
  for IDF in range(SIZE)
   :
  for IDF in range(SIZE)
    IDF[IDX_FNC, ..., IDX_FNC], ..., IDF[IDX_FNC, ..., IDX_FNC]
    = SF(
    IDF[IDX_FNC, ..., IDX_FNC], ..., IDF[IDX_FNC, ..., IDX_FNC])
```

```
@mdh( ... )
def IDF():
  return (
    out_view[ BSC_TYP,...,BSC_TYP ](
        IDF   =   [IDX_FNC, ..., IDX_FNC]
                          :
        IDF   =   [IDX_FNC, ..., IDX_FNC] ) ,
    md_hom[SIZE,...,SIZE]( SF,  CO,...,CO ),
    inp_view[ BSC_TYP,...,BSC_TYP ](
        IDF   =   [IDX_FNC, ..., IDX_FNC]
                          :
        IDF   =   [IDX_FNC, ..., IDX_FNC] )
  )
```

Figure 2: Transformation of an MDH directive into MDH-DSL (computations)

Each loop level in the nest must be associated with a reduction operator CO.[10] The scalar function SF may access multiple elements per buffer (e.g., to express stencil computations, as in Listing 10). These accesses are denoted generically in the listing as $\#ACC^{IB}_{b}$ and $\#ACC^{OB}_{b}$ for the b-th input or output buffer, respectively.

Buffers can optionally declare their size (omitted in Listing 14 for brevity)—this is required, for example, when buffers are larger than the accessed region (as in Listing 12).

4.3 Transformation: MDH Directive to DSL

Figures 1 and 2 illustrate the transformation from the MDH directive (Listing 14) to an MDH DSL program (Listing 7). Through this transformation, we can reuse the existing DSL-based MDH compilation pipeline [32] to generate auto-tuned code for GPUs and CPUs, incorporating tiling, data-movement, and parallelization optimizations.

Figure 1 highlights the data-centric aspects of this transformation. The top and bottom parts of the figure show output and input data, respectively. The directive encodes all information needed to instantiate the DSL's higher-order functions `out_view` (for outputs) and `inp_view` (for inputs).

Figure 2 focuses on the computation-centric aspects. It shows how the directive provides all information required to instantiate the `md_hom` higher-order function, which captures the data-parallel computation in the MDH DSL.

[10]The MDH formalism [32] refers to reduction operators as *Combine Operator (CO).*

5 Experimental Results[11]

We evaluate the performance of the GPU and CPU code generated from our directive against state-of-the-art directive-based approaches: OpenACC and PPCG on GPUs, and OpenMP, Pluto, and Numba on CPUs. For completeness, we also include highly tuned vendor libraries, such as NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN, which provide assembly-optimized implementations that often achieve near-peak hardware performance. In addition, we evaluate TVM [12], a state-of-the-art DSL for portable high-performance tensor computations on both GPUs and CPUs.

Auto-Tuning. Our approach is built on MDH and thus employs fully automatic auto-tuning for both GPU and CPU code [32], using the *Auto-Tuning Framework (ATF)* [35, 38]. Similarly, TVM uses its own auto-tuning engine [47]. To minimize the impact of tuning variability and ensure fair evaluation, we allocate a generous tuning time of 12 hours for both frameworks. As auto-tuning performance is not the primary focus of this work, we refer interested readers to Rasch et al. [35] for an in-depth discussion.

By contrast, PPCG and Pluto use heuristics but also support auto-tuning; for completeness, we report both heuristic and ATF-tuned results (using also 12 h of tuning time).

Vendor libraries (cuBLAS, cuDNN, oneMKL, oneDNN) do not support auto-tuning, presumably to avoid tuning overhead—even though this overhead is often amortized in practice, especially in deep learning workloads where tuned kernels are reused extensively after one-time auto-tuning. Similarly, OpenMP, OpenACC, and Numba also lack native auto-tuning support.[12]

[11]All experiments are fully reproducible using our artifact implementation [37].

[12]OpenACC provides a `tile` directive, and OpenMP allows manual tiling. Consequently, both can potentially use auto-tuned tile sizes, but this requires the user to

	Computation Characteristics				Data Characteristics			
Computation	Iter. Space	Red. Dim.	Data Acc.	Inp.	Sizes		Basic Type	Domain
`Dot`	1D	✓	Inj.	1	2^{24}	2^{24}	fp32	Simulation
				2	10^{7}	10^{7}	fp32	Simulation
`MatVec`	2D	✓	Non-Inj.	1	4096x4096	4096	fp32	Simulation
				2	8192x8192	8192	fp32	Simulation
`MatMul`	3D	✓	Non-Inj.	1	1024x1024	1024x1024	fp32	Simulation
				2	1x2048	2048x1000	fp32	Deep Learning
`MatMul^T`	3D	✓	Non-Inj.	1	64x10	500x64	fp32	Deep Learning
`bMatMul`	4D	✓	Non-Inj.	1	16x10x64	16x64x500	fp32	Deep Learning
`Gaussian_2D`	2D		Non-Inj.	1	224x224		fp32	Image Processing
				2	4096x4096		fp32	Image Processing
`Jacobi_3D`	3D		Non-Inj.	1	254x254x254		fp32	Simulation
				2	510x510x510		fp32	Simulation
`PRL`	2D	✓	Non-Inj.	1	2^{10}	2^{15}	{int64, chr46, fp64, ...}	Data Mining
				2	2^{15}	2^{15}	{int64, chr46, fp64, ...}	Data Mining
`CCSD(T)`	7D	✓	Non-Inj.	1	24x16x16x16	24x16x24x24	fp32	Quantum Chem.
				2	24x16x24x16	24x16x24x16	fp32	Quantum Chem.
`MCC`	7D	✓	Non-Inj.	1	1x512x7x7	512x512x3x3	fp32	Deep Learning
				2	1x230x230x3	64x7x7x3	fp32	Deep Learning
`MCC_Caps`	10D	✓	Non-Inj.	1	16x230x230x3x4x4	64x7x7x3x4x4	fp32	Deep Learning
				2	1x230x230x3x4x4	67x7x7x3x4x4	fp32	Deep Learning

Figure 3: Characteristics of computations (left part of figure) and data (right part). For each computation, we denote the dimensionality of its iteration space and whether it contains reduction dimensions. For each data set, we denote its size, its basic type, and its domain. Column "`No.`" denotes the number of the data set per study.

Case Studies and Data Sets. Figure 3 summarizes our real-world case studies, which exhibit a wide range of computational characteristics: linear algebra routines (`Dot`, `MatVec`, and several variants of `MatMul`), stencil computations (`Gaussian_2D` and `Jacobi_3D`), a data mining example (`PRL` [34]), quantum chemistry workloads (`CCSD(T)` [23]), and deep learning examples (`MCC` and `MCC_Caps`—a generalized variant of MCC for *capsule-style networks*, known to be particularly challenging to optimize [6]).

All our experiments use real-world data characteristics from their respective domains, as detailed in Figure 3. For instance, for PRL, which detects duplicate entries in databases, we use real-world data from the German cancer registry *EKR* [19].

5.1 Experimental Setup

We conduct experiments on an NVIDIA A100-PCIE-40GB GPU and an Intel Xeon Gold 6140 CPU.

Our evaluation is based on a software environment comprising: OpenACC from the NVIDIA HPC SDK 22.1, OpenMP from the Intel oneAPI Base Toolkit 2022.0.0, PPCG 0.08.4, Pluto (commit 12e075a), Numba 0.61.2, and TVM 0.8.0. As vendor libraries, we use NVIDIA cuBLAS and cuDNN from CUDA Toolkit 11.4, and Intel oneMKL and oneDNN from Intel oneAPI Base Toolkit 2022.0.0. In addition, we evaluate the EKR library [19], executed on Java SE 1.8.0_281.

In all experiments, we collect measurements until the 99% confidence interval was within 5% of our reported means, according to the guidelines for *scientific benchmarking of parallel computing systems* by Hoefler and Belli [20].

5.2 Performance Results

Figure 4 shows the performance of our MDH-based directive approach, generating CUDA code for GPUs and OpenCL code for CPUs, compared to related approaches. We observe that our method consistently achieves higher performance, often exceeding other approaches by orders of magnitude.

Compared to OpenACC, our approach achieves > 150× speedup for quantum-chemistry computation CCSD(T) across both input sizes. This large gap stems from OpenACC's inability to automatically apply tiling optimizations. Manual tiling improves performance but is error-prone and nontrivial: for instance, applying the `tile` directive to all reduction-free loops—a seemingly safe choice—produces incorrect results without warning. Tiling only the first reduction-free loop slows execution, whereas empirically

explicitly express tiling, select tiling candidates, and integrate an external tuning framework (e.g., ATF), as discussed later in this section.

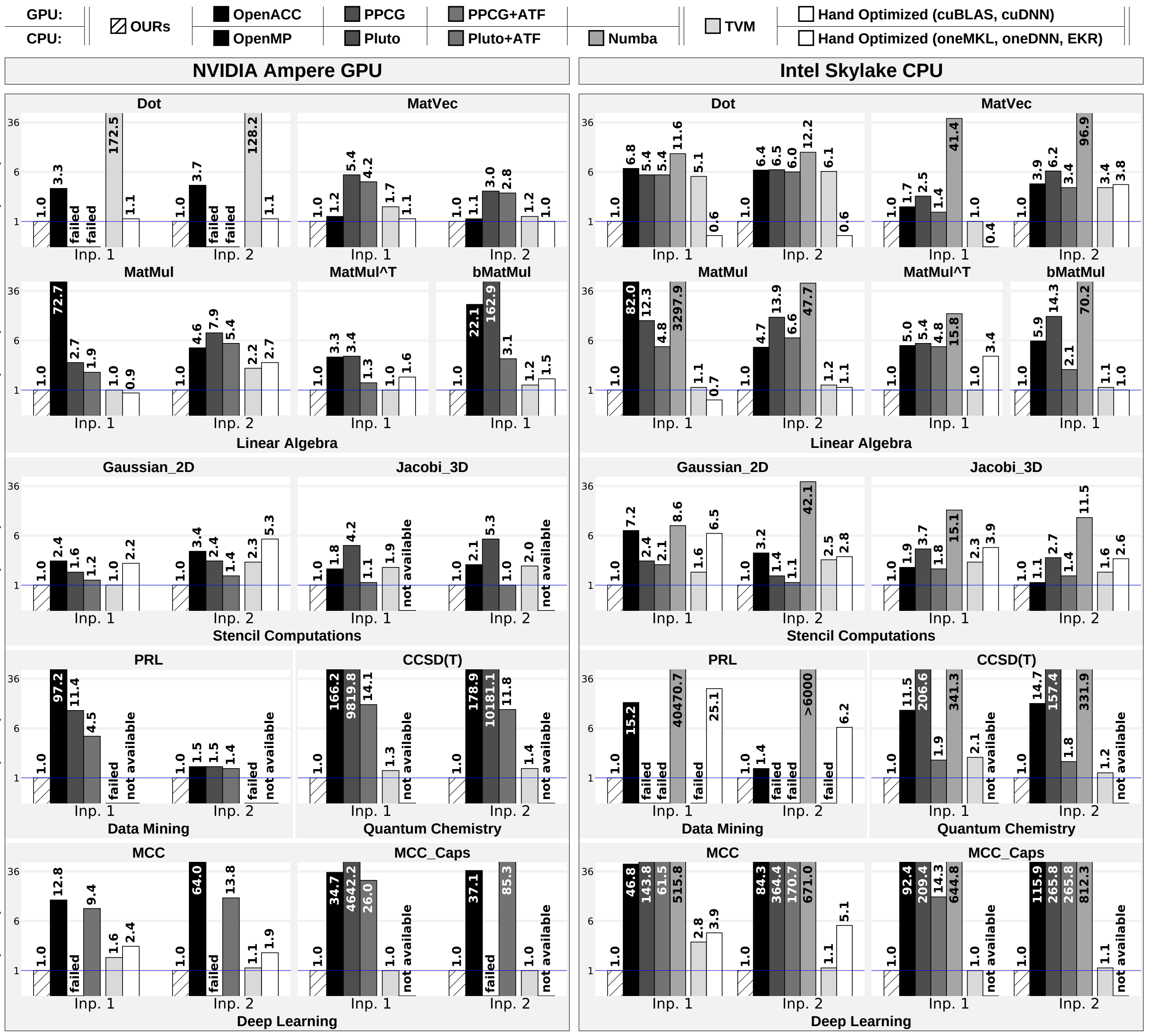


Figure 4: Speedup (higher is better for our approach) of our generated code compared to state-of-art approaches

identifying and tiling four specific loops (via trial and error) raises OpenACC's performance to about 60× slower than our approach, compared to 150× slower without tiling. Further manual tuning migh narrow the gap but demands substantial programming effort, thereby undermining the goal of directive-based approaches aimed at full automation.

OpenMP exhibits similar limitations, as it requires explicit manual tiling to achieve higher performance. Unlike OpenACC, however, it provides no built-in `tile` directive, which makes tiling technically cumbersome to express.

These challenges also account for the superior performance of our approach over OpenACC and OpenMP in other benchmarks such as MatMul and deep-learning computations.

Compared to polyhedral compilers, our approach achieves higher performance, primarily because it handles reduction computations more effectively—an area where polyhedral techniques still face challenges [13].

An interesting observation involves the `dot` and PRL benchmarks. The `dot` computation is reduction-heavy, and polyhedral compilers—despite their strong optimizations—fail to deliver high performance. This is because their directives do not convey the necessary semantic information (i.e., the reduction operators) needed to optimize reductions effectively (see Section 2).

For PRL, OpenACC and OpenMP achieve high performance for the second input size (`Inp. 2` in the figure) but not for the first (`Inp.1`). The first input size (see Figure 3) features a large reduction dimension (2^{15} database entries, each representing an existing cancer patient) and a comparatively small non-reduction dimension (2^{10} new cancer patients to be added without duplicates), which reflects real-world requirements. OpenMP and OpenACC underperform in this setting because they cannot express the PRL-specific reduction operator (see Section 2), preventing them from optimizing the reduction computation in PRL. For the second input size, we artificially increased the non-reduction dimension to 2^{15} new cancer patients (instead of 2^{10}), making the computation more favorable for OpenMP and OpenACC; however, this configuration does not represent realistic scenarios.

Numba often fails to deliver high performance because, like polyhedral compilers, it does not capture semantic information about reduction operators in its directives. Moreover, Numba most likely[13] does not automatically apply important optimizations—such as tiling—in its generated CPU code.

Note that some approaches fail to generate code for certain case studies. For example, PPCG cannot generate GPU code for `dot` due to its reduction-heavy nature and crashes with an *out of resources* error on deep learning computations when ATF-tuned tile sizes are not used. For PRL, Pluto fails with *"Error extracting polyhedra from source"*, likely because it struggles to handle `if` statements in the PRL code (see Listing 11). Similarly, TVM often encounters difficulties with user-defined reduction operators [2, 3].

6 Related Work

Simplifying programming for modern parallel architectures, such as GPUs and multi-core CPUs, is a primary motivation for many popular approaches.

Directive-based methods (discussed in Section 2) offer high productivity for non-expert application developers—e.g., physicists, chemists, AI scientists—by automatically generating optimized parallel code from straightforward sequential code annotated with simple directives. However, these methods often struggle to deliver high, portable performance, especially for reduction-heavy computations, as confirmed experimentally in Section 5.

For instance, polyhedral compilers like PPCG and Pluto cannot parallelize or optimize reduction computations effectively because they rely on overly simple directives that lack information about reduction operators. This simplicity makes them easy to use but sacrifices performance. In contrast, Numba can parallelize simple reductions through program analysis, while OpenMP and OpenACC only optimize reduction computations if their operators are natively supported—mainly addition, multiplication, and similar. Furthermore, many of these approaches target specific architectures, such as OpenMP, Pluto, and Numba focusing on CPUs, and OpenACC and PPCG primarily targeting NVIDIA GPUs.

[13]We cannot confidently explain Numba's performance results, as its assembly code generation obscures the underlying optimizations.

Unlike these general-purpose directive-based approaches, our method deliberately focuses on the class of data-parallel computations. This specialization enables us to achieve significantly higher performance and portability. In particular, our approach provides stronger support for reduction operators, which substantially enhances both performance and portability for reduction-heavy data-parallel workloads.

Additional approaches aiming to simplify programming parallel architectures rely on high-level DSLs [4, 5, 7, 8, 10–12, 16–18, 21, 22, 25, 29, 31, 39–42, 44, 46], rather than code annotations via simple directives. These DSLs, such as Halide [31] for image processing and TVM [12] for deep learning, offer productive interfaces for parallel architectures and often deliver high performance. However, they require developers to adopt DSL-specific programming models, which can be unconventional for many application programmers. In addition, they are typically restricted to narrow domains: for instance, while TVM is highly effective for deep learning and tensor computations, it cannot easily express our data-mining examples PRL (Listing 11) and MBBS (Listing 13) [3], as both depend on reduction operators uncommon in deep learning.

High-performance libraries [15, 30, 45], such as NVIDIA cuBLAS and cuDNN, as well as Intel oneMKL and oneDNN, deliver state-of-the-art performance but are typically tied to a specific combination of hardware architecture and application domain. For instance, cuBLAS and oneMKL are limited to linear algebra routines, and in addition are restricted in hardware support: cuBLAS to NVIDIA GPUs and oneMKL to CPUs.

Compared to the MDH approach [32], which serves as the foundation of our work, we develop a novel directive-based programming approach, on top of MDH, that enables users to express data-parallel computations using familiar, straightforward Python code. This directive-based model avoids the need for DSL programming, which can still pose a significant barrier to domain scientists who may not be familiar with DSL abstractions or the specific details of these specialized languages. By relying on widely known Python constructs, our model lowers the entry barrier and allows users to focus on the computational logic of their applications rather than the complexities of parallel programming or DSL design. In our experimental evaluation, we present the first systematic comparison of the MDH approach with directive-based approaches, including established standards such as OpenMP and OpenACC as well as the widely used Python-based framework Numba.

7 Conclusion

We present a novel directive-based programming approach built on the formalism of Multi-Dimensional Homomorphisms (MDH). By explicitly capturing reduction operators and separating them from loop bodies, our design enables concise and expressive specifications of data-parallel computations, while preserving the simplicity of familiar Python code.

Our evaluation across domains such as linear algebra, stencils, data mining, quantum chemistry, and deep learning shows that our approach achieves state-of-the-art performance on CPUs and GPUs, with speedups of up to 5× on CPUs and over 2× on GPUs, even against highly optimized vendor libraries from Intel and NVIDIA across both reduction-based and reduction-free real applications.

8 Future Work

As future work, we plan to explore incorporating our directive into OpenMP and OpenACC, thereby paving the way for MDH-based optimizations to become part of widely adopted directive standards and thus broadly accessible also for C, C++, and Fortran programmers.

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project number 470527619 (MDH.DL).

References

[1] 2025. TIOBE – The Software Quality Company. https://www.tiobe.com/tiobe-index/.

[2] Apache TVM Community. 2022. Expressing nested reduce operations. https://discuss.tvm.apache.org/t/expressing-nested-reduce-operations/8784.

[3] Apache TVM Community. 2022. Invalid comm_reducer. https://discuss.tvm.apache.org/t/invalid-comm-reducer/12788.

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. doi:10.1109/CGO.2019.8661197

[5] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 128–138. doi:10.1145/2854038.2854048

[6] Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. doi:10.1145/3317550.3321441

[7] Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. 2022. Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 313–324. doi:10.1109/CGO53902.2022.9741260

[8] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[10] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. doi:10.1145/1926354.1926358

[11] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Technical Report 08-897, U. of Southern California.

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions. arXiv:1505.07716 [cs.PL] https://arxiv.org/abs/1505.07716

[14] Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-Conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 610–624. doi:10.1145/3314221.3314612

[15] M. Frigo and S.G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, Vol. 3. 1381–1384 vol.3. doi:10.1109/ICASSP.1998.681704

[16] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317. doi:10.1007/s10766-006-0012-3

[17] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. doi:10.1145/3410463.3414632

[18] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. doi:10.1145/3062341.3062354

[19] K Hentschel et al. 2008. Das Krebsregister-Manual der Gesellschaft der epidemiologischen Krebsregister in Deutschland eV Zuckschwerdt Verlag. (2008).

[20] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. doi:10.1145/2807591.2807644

[21] Wayne Kelly and William Pugh. 1998. *A framework for unifying reordering transformations.* Technical Report. Technical Report UMIACS-TR-92-126.1.

[22] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. *ACM Trans. Archit. Code Optim.* 9, 4, Article 31 (jan 2013), 25 pages. doi:10.1145/2400682.2400690

[23] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A Code Generator for High-Performance Tensor Contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 85–95. doi:10.1109/CGO.2019.8661182

[24] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. doi:10.1145/2833157.2833162

[25] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. doi:10.1145/3498717

[26] Numba. 2025. Automatic parallelization with @jit. https://numba.readthedocs.io/en/stable/user/parallel.html.

[27] OpenACC-Standard.org. 2025. *OpenACC: More Science, Less Programming.* https://www.openacc.org/specification

[28] OpenMP. 2025. *OpenMP: The OpenMP API specification for parallel programming.* https://www.openmp.org/specifications/

[29] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (aug 2021), 29 pages. doi:10.1145/3473593

[30] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. doi:10.1109/JPROC.2004.840306

[31] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[32] Ari Rasch. 2024. (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms. *ACM Trans. Program. Lang. Syst.* 46, 3, Article 10 (Oct. 2024), 74 pages. doi:10.1145/3665643

[33] Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2020. dOCAL: high-level distributed programming with OpenCL and CUDA. *The Journal of Supercomputing* 76, 7 (2020), 5117–5138. doi:10.1007/s11227-019-02829-2

[34] Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019. High-performance probabilistic record linkage via multi-dimensional homomorphisms. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association

for Computing Machinery, New York, NY, USA, 526–533. doi:10.1145/3297280.3297330

[35] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1, Article 1 (Jan. 2021), 26 pages. doi:10.1145/3427093

[36] Ari Rasch, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2018. OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 408–416. doi:10.1109/PADSW.2018.8644541

[37] Richard Schulze. 2025. WACCPD25 Artifact. GitLab repository. https://gitlab.com/mdh-project/waccpd25_artifact

[38] Richard Schulze, Sergei Gorlatch, and Ari Rasch. 2025. pyATF: Constraint-Based Auto-Tuning in Python. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) *(CC '25)*. Association for Computing Machinery, New York, NY, USA, 35–47. doi:10.1145/3708493.3712682

[39] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 205–217. doi:10.1145/2784731.2784754

[40] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2020. Meta-programming for cross-domain tensor optimizations. *SIGPLAN Not.* 53, 9 (apr 2020), 79–92. doi:10.1145/3393934.3278131

[41] Joel Svensson, Mary Sheeran, and Koen Claessen. 2011. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–173.

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. doi:10.1145/3355606

[43] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. doi:10.1145/2400682.2400713

[44] Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. 2022. LoopStack: a Lightweight Tensor Algebra Compiler Stack. doi:10.48550/ARXIV.2205.00618

[45] R.C. Whaley and J.J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 38–38. doi:10.1109/SC.1998.10004

[46] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. doi:10.1145/3519939.3523437

[47] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

A Pre-Implemented Reduction Operators

Listings 15–17 show the implementation of reduction operators cc (concatenation), pw (point-wise), and ps (prefix sum).

```
def cc( T:ScalarType, D:int, d:int ):
  @combine_operator(
    index_set_function  = lambda I: I,
    scalar_type         = T,
    dimensionality      = D,
    operating_dimension = d
  )
  def cc__T_D_d( I, P,Q ):
    def cc__T_D_d__I_PQ( res, lhs,rhs ):

      for i[1,...,d-1] in I[1,...,d-1]:
        for i[d+1,...,D] in I[d+1,...,D]:

          for i[d] in P:
            res[ i[1,...,d,...,D] ] =
                           lhs[ i[1,...,d,...,D] ]
          for i[d] in Q:
            res[ i[1,...,d,...,D] ] =
                           rhs[ i[1,...,d,...,D] ]
    return cc__T_D_d__I_PQ
  return cc__T_D_d
```

Listing 15: Reduction operator cc (concatenation) (Python)

```
def pw( cf:PW_CustomFunc ):
  def pw__cf( T:ScalarType, D:int, d:int ):
    @combine_operator(
      index_set_function  = lambda I: {0},
      scalar_type         = T,
      dimensionality      = D,
      operating_dimension = d
    )
    def pw__cf__T_D_d( I, P,Q ):
      def pw__cf__T_D_d__I_PQ( res, lhs,rhs ):
        for i[1,...,d-1] in I[1,...,d-1]:
          for i[d+1,...,D] in I[d+1,...,D]:
            cf(
              res[ i[1,...,d-1],0,i[d+1,...,D] ],
              lhs[ i[1,...,d-1],0,i[d+1,...,D] ],
              rhs[ i[1,...,d-1],0,i[d+1,...,D] ])
      return pw__cf__T_D_d__I_PQ
    return pw__cf__T_D_d
  return pw__cf
```

Listing 16: Reduction operator pw (point-wise) for arbitrary customizing function cf (Python)

```
def ps( cf:PS_CustomFunc ):
  def ps__cf( T:ScalarType, D:int, d:int ):
  @combine_operator(
   index_set_function  = lambda I: I,
   scalar_type         = T,
   dimensionality      = D,
   operating_dimension = d
  )
  def ps__cf__T_D_d( I, P,Q ):
   def ps__cf__T_D_d__I_PQ( res, lhs,rhs ):
    for i[1,...,d-1] in I[1,...,d-1]:
     for i[d+1,...,D] in I[d+1,...,D]:
      for i[d] in P:
       q_sm_i_d = set(q for q in Q if q < i[d])
       if q_sm_i_d:
        cf(
         res[ i[1,...,d-1] ,
              i[d]         ,
              i[d+1,...,D] ],
         lhs[ i[1,...,d-1] ,
              i[d]         ,
              i[d+1,...,D] ],
         rhs[ i[1,...,d-1] ,
              max( q_sm_i_d ),
              i[d+1,...,D] ] )
       else:
        res[ i[1,...,d-1] ,
             i[d]         ,
             i[d+1,...,D] ] =
        lhs[ i[1,...,d-1] ,
             i[d]         ,
             i[d+1,...,D] ]
      for i[d] in Q:
        # ... (analogous to above)
   return ps__cf__T_D_d__I_PQ
  return ps__cf__T_D_d
 return ps__cf
```

Listing 17: Reduction operator ps (prefix sum) for arbitrary customizing function cf (Python)

B Artifact Description (AD)

B.1 Relation to Contributions

Our artifact [37] experimentally assesses the main contributions of this paper by providing implementations, benchmarks, and workflows that reproduce the results presented in the evaluation in Section 5.

B.2 Expected Results

The artifact reproduces the results presented in Figure 4, thereby substantiating the main contributions of this paper.

B.3 Expected Reproduction Time

The total runtime of the artifact is approximately 12 hours if the experiments are executed in parallel.

B.4 Artifact Setup

Hardware. Described in Section 5 and in the artifact README [37].

Software. Described in Section 5 and in the artifact README [37].

Datasets/Input. Described in Figure 3 and provided in our artifact package [37].

Installation and Deployment

Described in the artifact README [37].

B.5 Artifact Evaluation

The experimental workflow, including the individual tasks and their dependencies, as well as the experimental parameters and repetitions, is described in the artifact README [37].

B.6 Artifact Analysis

The artifact analysis workflow, including the post-processing of raw experimental results into the final plots and tables presented in this paper, is documented in the accompanying artifact README [37].