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

Richard Schulze, Sergei Gorlatch, Ari Rasch
University of Münster, Germany

Who are we?

Compilers for AI & HPC — our research projects**:**

https://mdh-lang.org

Code Generation

https://atf-tuner.org

Code Optimization

Code Execution

This work is based on MDH

https://arirasch.net
a.rasch@uni-muenster.de

Ari Rasch

https://richardschulze.net
r.schulze@uni-muenster.de

Richard Schulze

Hunloh Brothers (Lars & Jens)

Message of this Talk

Reductions are ubiquitous:

Linear Algebra

```
def matmul(A, B, I, J, K):
    C = [[0.0 for _ in range(J)] for _ in range(I)]
    for i in range(I):
        for j in range(J):
            for k in range(K):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

Algorithmic

```
def mbbs(I, J):
    O = [0.0 for _ in range(I)]
    prefix = 0.0

    for i in range(I):
        row_sum = 0.0
        for j in range(J):
            row_sum += I[i][j]

        prefix += row_sum
        O[i] = prefix

    return O
```

Deep Learning

```
def mcc(img, flt, N, P, Q, K, R, S, C):
    res = [[[[0.0 for _ in range(K)] for _ in range(Q)]
                for _ in range(P)] for _ in range(N)]
    for n in range(N):
        for p in range(P):
            for q in range(Q):
                for k in range(K):
                    for r in range(R):
                        for s in range(S):
                            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 res
```

... (Quantum Chemistry, Data Mining, etc)

often use more *complex reduction operators* than +=

Awareness of reductions in directive-based programming can **significantly enhance performance**

State-of-the-Art Directive-Based Approaches

Analysis of the state of the art focussing on their reduction handling:

Polyhedral Compilers PPCG & Pluto:

```
void matvec_poly(const T *M, const T *v, T *w) {
  #pragma scop
  for (int i = 0; i < I; i++) {
    w[i] = 0.0f;
    for (int k = 0; k < K; k++) {
      w[i] += M[i * K + k] * v[k];
    }
  }
  #pragma endscop
```

Fully automatic & formal foundation, but lack reduction-operator semantics

OpenMP (& OpenACC):

```
void matvec_openmp(const T* M, const T* v, T* w)
{
  #pragma omp parallel for
  for (int i = 0; i < I; ++i) {
    T sum = 0.0f;

    #pragma omp simd reduction(+:sum)
    for (int k = 0; k < K; ++k) {
      sum += M[i * K + k] * v[k];
    }
    w[i] = sum;
} }
```

Productive, but limited reduction-operator semantics

Numba:

```
def matvec_numba_cpu(I, K):
    @njit(parallel=True)
    def matvec__I_K(w, M, v):
        for i in range(I):
            w[i] = 0.0
            for k in range(K):
                w[i] += M[i, k] * v[k]
    return matvec__I_K
```

Fully automatic & Python-based, but lacks reduction-operator semantics & limited GPU productivity

All approaches:

Further performance potential

(also for reduction-free computations)

Contribution of this Work

Introduce **reduction-aware directive** addressing limitations of state of the art:

Directive *name*: @mdh explained and motivated later)

```
def matvec( T:BasicType, I:int,K:int ):
  @mdh( out( w = Buffer[T]                 ),
        inp( M = Buffer[T], v = Buffer[T] ),
        combine_ops( cc, pw(add) )
      )
  def mdh_matvec__T_I_K( w, M,v ):
    for i in range(I):
      for k in range(K):
        w[i] = M[i,k] * v[k]
  return mdh_matvec__T_I_K
```

Declaration of *output data*:
name & basic type

Declaration of *input data*:
name & basic type

Expressing *reductions*:
`cc` (*concatenation*) &
`pw(add)` (*pointwise* via *addition*)

Key Strength:

- **Explicit** representation of reduction operators
- Supports **arbitrary, user-defined reductions** (`cc` & `pw` are pre-implemented)
- **Python**-based interface

Our **MDH directive** is designed to **efficiently express reductions**

Excursion: User-Defined Operators

In a nutshell

Explicit implementation of reduction operator ***concatenation* (cc)**:

```
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
```

Name

Meta Parameters

Input Sizes (generic)

Output & Inputs

Iterating over `lhs` & `rhs`

Writing `lhs` consecutively to `res`

Writing `rhs` consecutively to `res`

Our interface for custom operators is **expressive** (but may require some familiarization)

Code Generation

<u>How to generate *high-performance executable code* (e.g., in CUDA) from our reduction-aware, directive-annotated Python programs:</u>

Overview | Getting Started | Code Examples | Publications | Citations | Contact

MDH

Multi-Dimensional Homomorphisms (MDH)

An Algebraic Approach Toward Performance & Portability & Productivity for Data-Parallel Computations

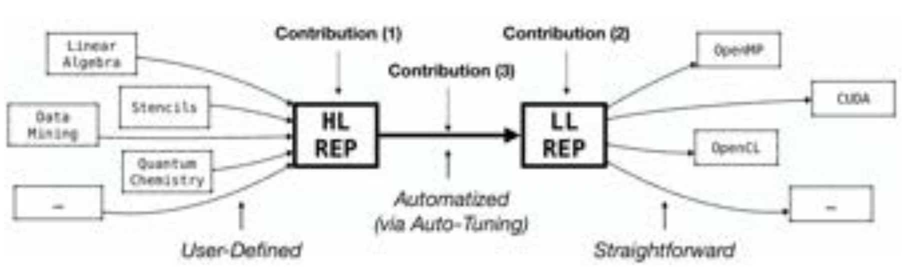
Overview

The approach of Multi-Dimensional Homomorphisms (MDH) is an algebraic formalism for systematically reasoning about *de-composition* and *re-composition* strategies of data-parallel computations (such as linear algebra routines and stencil computations) for the memory and core hierarchies of state-of-the-art parallel architectures (GPUs, multi-core CPU, multi-device and multi-node systems, etc).

The MDH approach (formally) introduces:

1. *High-Level Program Representation* *(Contribution 1)* that enables the user conveniently implementing data-parallel computations, agnostic from hardware and optimization details;
2. *Low-Level Program Representation* *(Contribution 2)* that expresses device- and data-optimized de- and re-composition strategies of computations;
3. *Lowering Process* *(Contribution 3)* that fully automatically lowers a data-parallel computation expressed in its high-level program representation to an optimized instance in its low-level representation, based on concepts from automatic performance optimization (a.k.a. *auto-tuning*), using the Auto-Tuning Framework (ATF).

The MDH's low-level representation is designed such that Code Generation from it (e.g., in OpenMP for CPUs, CUDA for NVIDIA GPUs, or OpenCL for multiple kinds of architectures) becomes straightforward.



Our Experiments report encouraging results on GPUs and CPUs for MDH as compared to state-of-practice approaches, including NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN which are hand-optimized libraries provided by vendors.

<u>https://mdh-lang.org</u>

<u>ACM TOPLAS 2024</u>

(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms

ARI RASCH, University of Muenster, Germany

Data-parallel computations, such as linear algebra routines and stencil computations, constitute one of the most relevant classes in parallel computing, e.g., due to their importance for deep learning. Efficiently de-composing such computations for the memory and core hierarchies of modern architectures and re-composing the computed intermediate results back to the final result—we say *(de/re)-composition* for short—is key to achieve high performance for these computations on, e.g., GPU and CPU. Current high-level approaches to generating data-parallel code are often restricted to a particular subclass of data-parallel computations and architectures (e.g., only linear algebra routines on only GPU or only stencil computations), and/or the approaches rely on a user-guided optimization process for a well-performing (de/re)-composition of computations, which is complex and error prone for the user.

We formally introduce a systematic (de/re)-composition approach, based on the algebraic formalism of Multi-Dimensional Homomorphisms (MDHs). Our approach is designed as general enough to be applicable to a wide range of data-parallel computations and for various kinds of target parallel architectures. To efficiently target the deep and complex memory and core hierarchies of contemporary architectures, we exploit our introduced (de/re)-composition approach for a correct-by-construction, parametrized cache blocking, and parallelization strategy. We show that our approach is powerful enough to express, in the same formalism, the (de/re)-composition strategies of different classes of state-of-the-art approaches (scheduling-based, polyhedral, etc.), and we demonstrate that the parameters of our strategies enable systematically generating code that can be fully automatically optimized (auto-tuned) for the particular target architecture and characteristics of the input and output data (e.g., their sizes and memory layouts). Particularly, our experiments confirm that via auto-tuning, we achieve higher performance than state-of-the-art approaches, including hand-optimized solutions provided by vendors (such as NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN), on real-world datasets and for a variety of data-parallel computations, including linear algebra routines, stencil and quantum chemistry computations, data mining algorithms, and computations that recently gained high attention due to their relevance for deep learning.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; *Machine learning*; • **Theory of computation → Program semantics**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Code generation, data parallelism, auto-tuning, GPU, CPU, OpenMP, CUDA, OpenCL, linear algebra, stencils computation, quantum chemistry, data mining, deep learning

A full version of this article is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this article relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader to Rasch [2024] for formal details that should not be required for understanding the basic ideas and concepts of our approach.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—project PPP-DL (470527619).

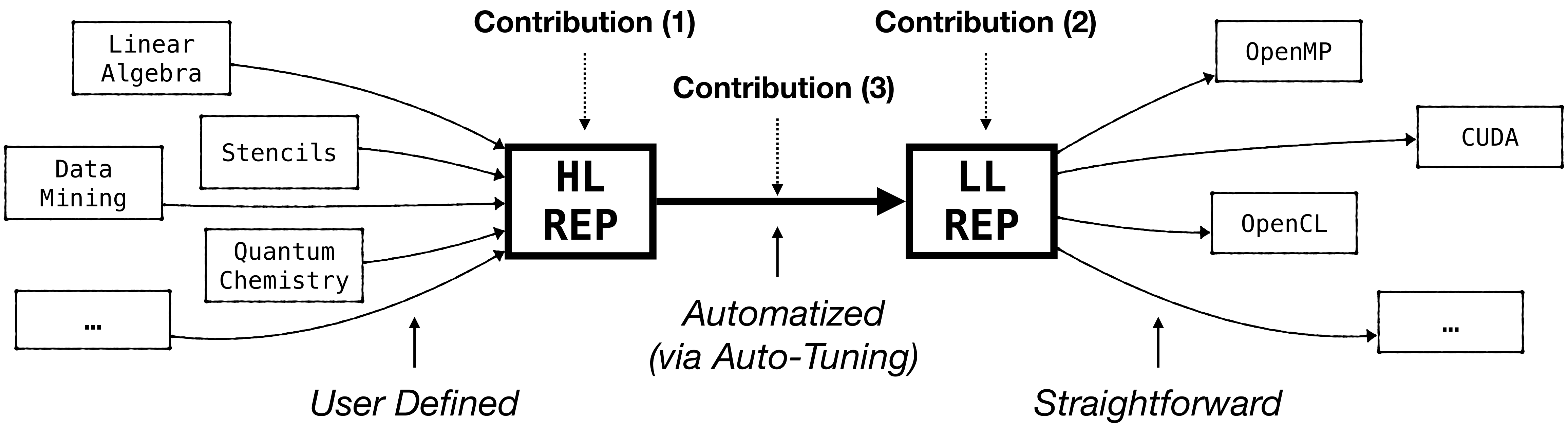
Author's Contact Information: Ari Rasch (Corresponding author), University of Muenster, Muenster, Germany; e-mail: a.rasch@uni-muenster.de.

Exploit ***MDH Approach*** for driving ***Code Generation*** and ***Optimization***

The MDH Approach

In a nutshell

MDH is a (formal) framework for *expressing* & *optimizing* data-parallel computations:



MDH generates high-performance code for programs expressed in its ***High-Level Representation***:

```
MatVec<T∈TYPE|I,K∈ℕ> :=  out_view<T>( w:(i,k)↦(i) ) ∘
                             md_hom<I,K>( *, (++,+) ) ∘
                                 inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

MDH High-Level Representation of MatVec

What is happening here:

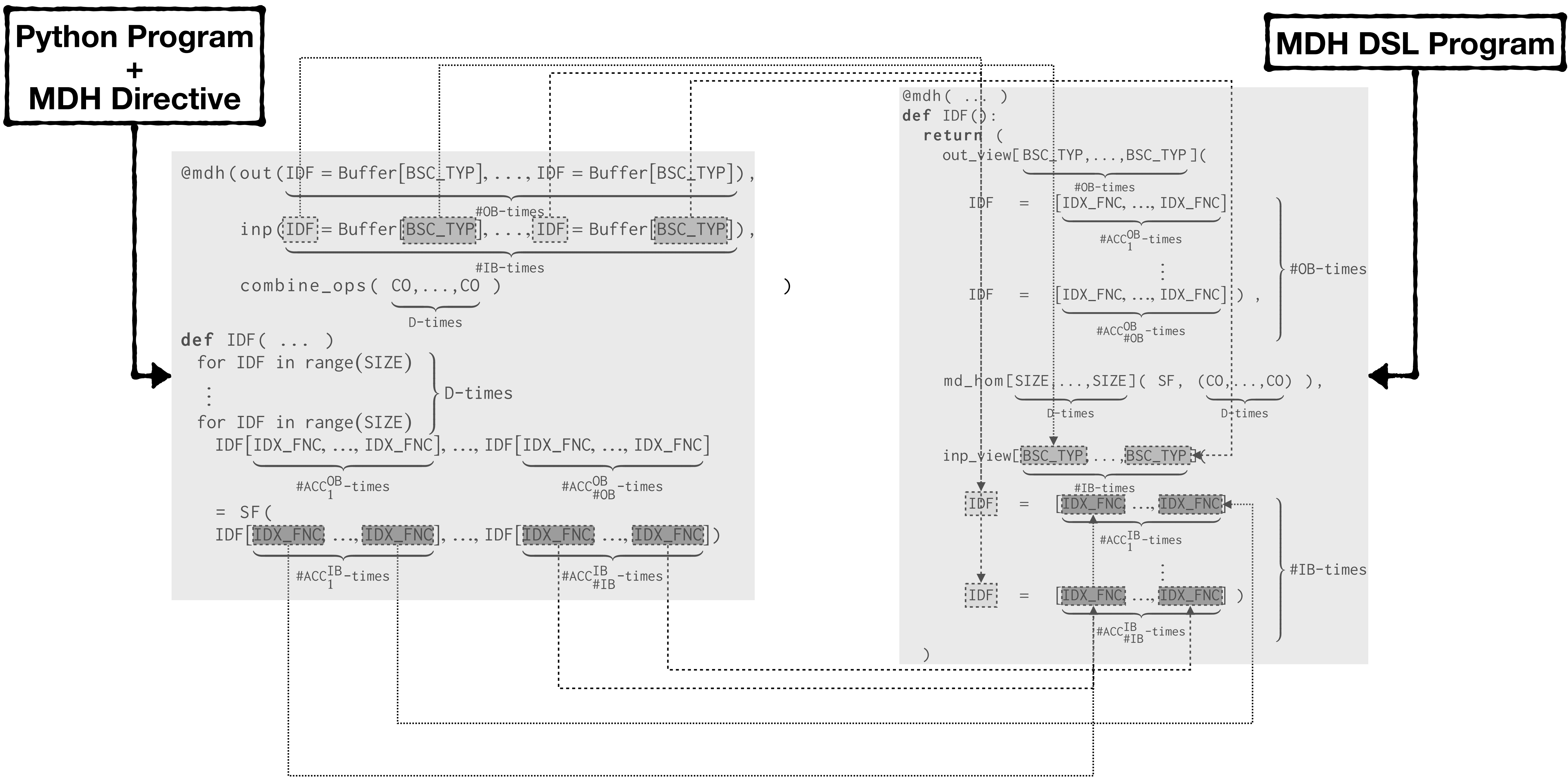
`inp_view` captures the accesses to input data

`out_view` captures the accesses to output data

`md_hom` expresses the data-parallel computation (incl. *reductions*)

Key Transformation

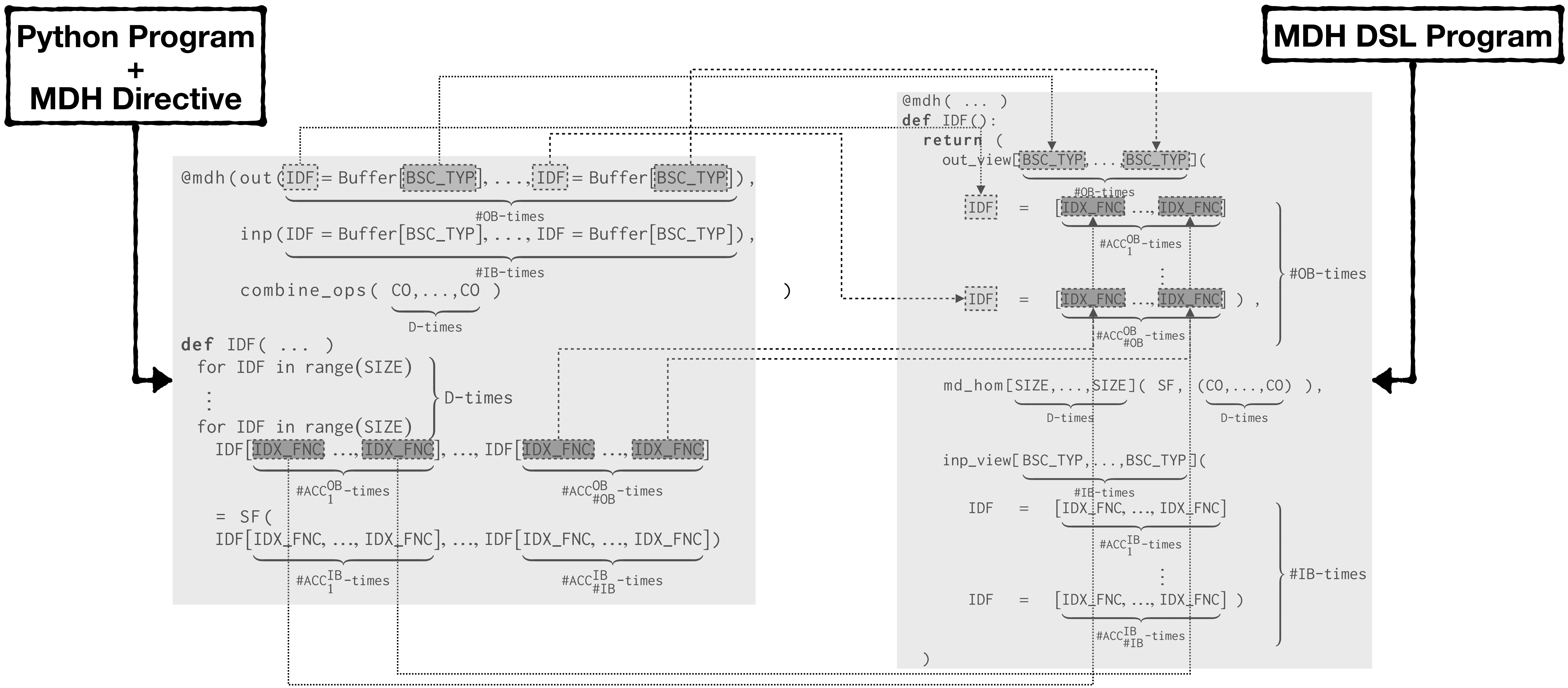
<u>Transform our MDH-directive-annotated Python programs into MDH-DSL representation:</u>



Our MDH-annotated Python code **captures** all **<u>input</u>-relevant building blocks of the MDH-DSL program**

Key Transformation

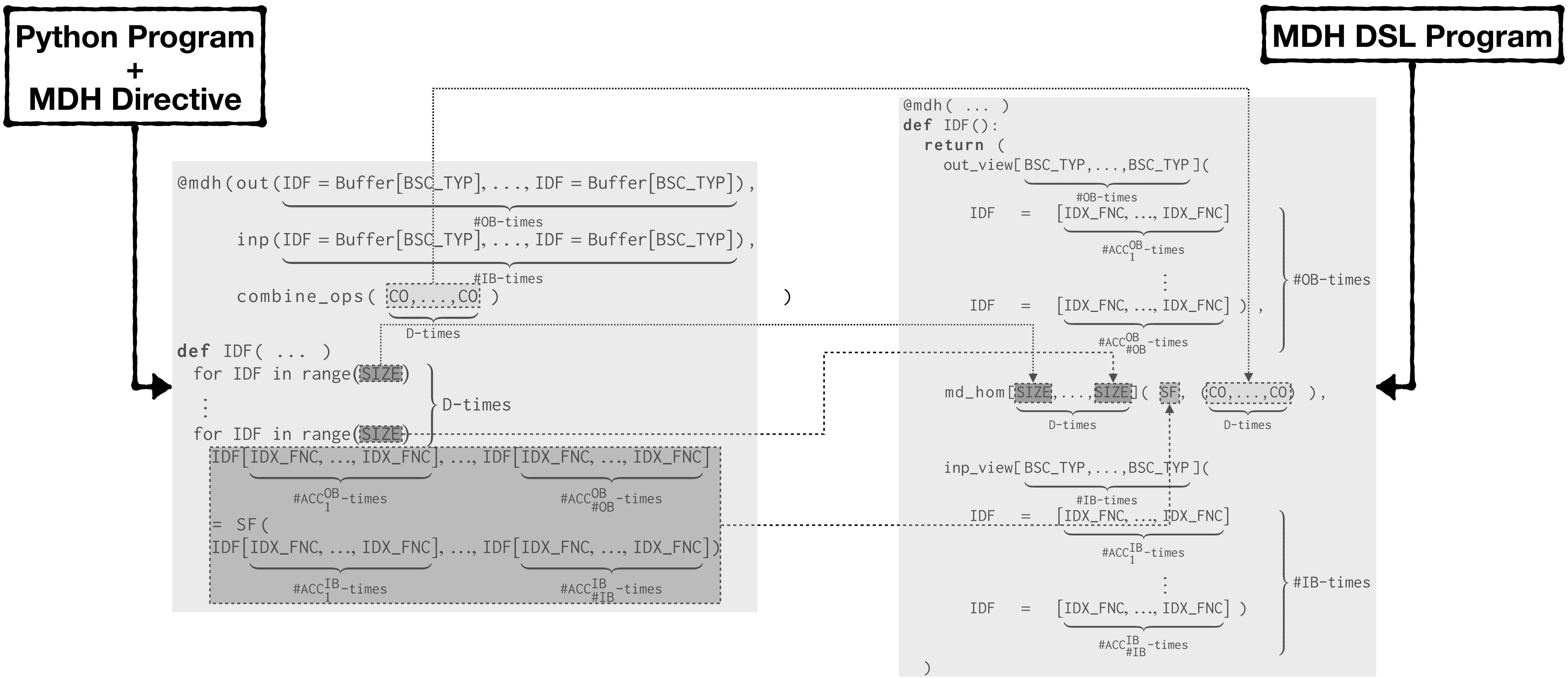
Transform our MDH-directive-annotated Python programs into MDH-DSL representation:



Our **MDH-annotated** Python code **captures** all **<u>output</u>-relevant building blocks of the MDH-DSL program**

Key Transformation

Transform our MDH-directive-annotated Python programs into MDH-DSL representation:



Our MDH-annotated Python code captures all computation-relevant building blocks of the MDH-DSL program

Experimental Results

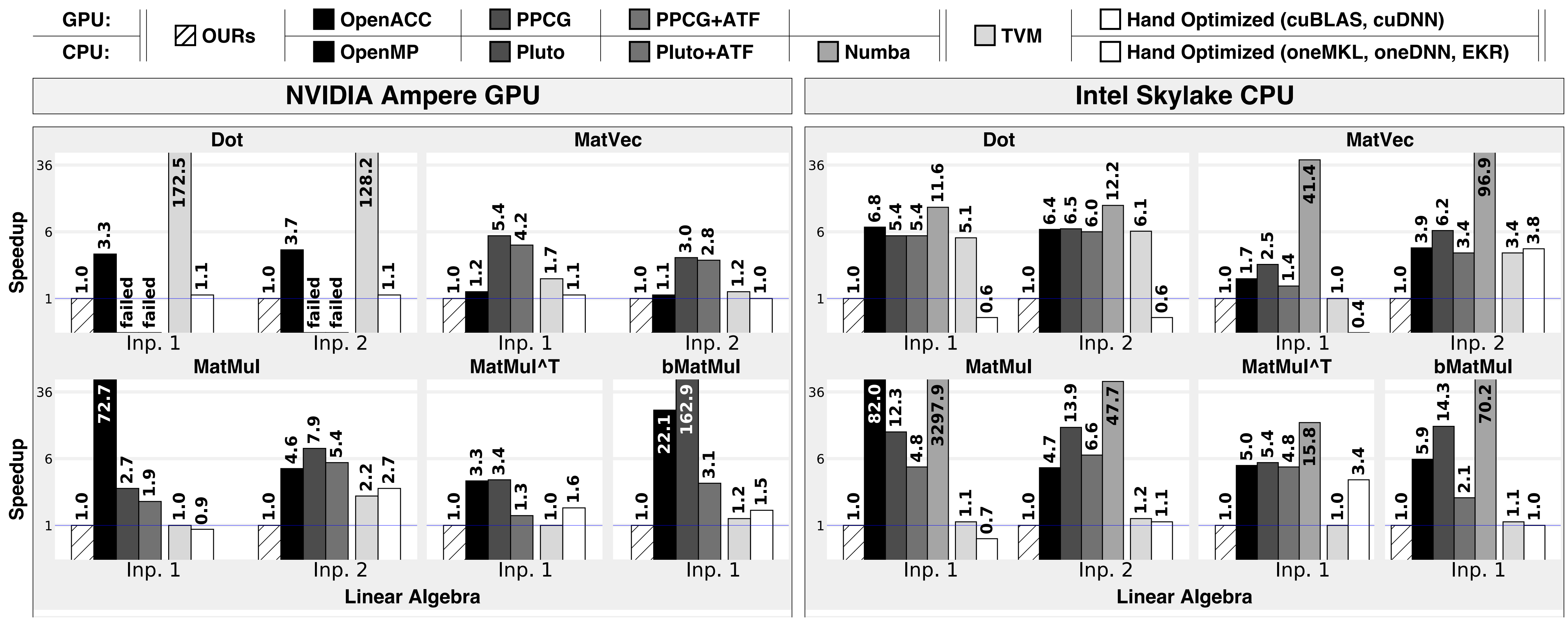
<u>Case Studies & Data Characteristics:</u>

	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

Evaluation across **real-world case studies** and **data characteristics**

Experimental Results

Linear Algebra:

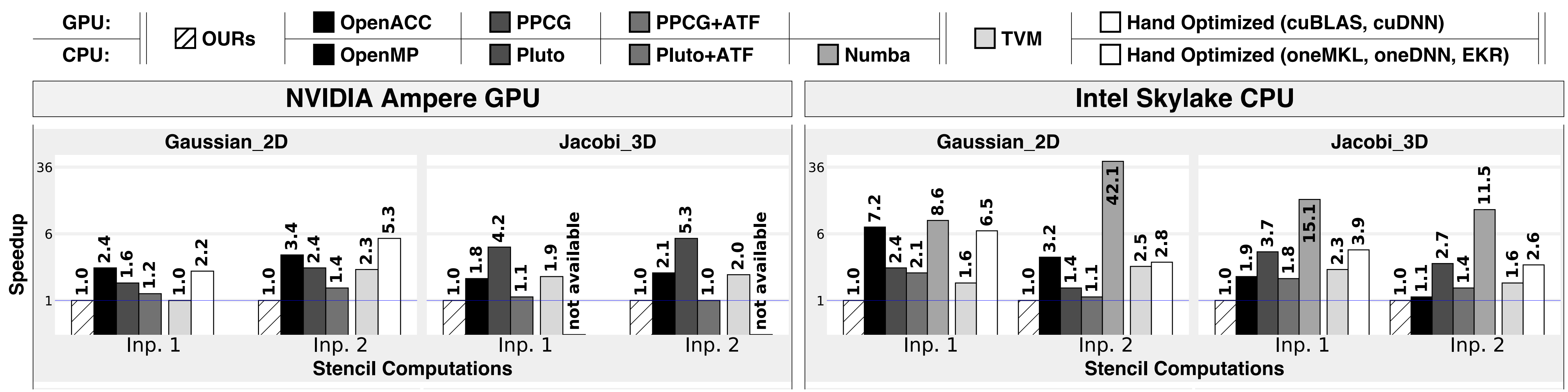


Highlights (*speedups* — higher is better for us):

- **GPU:**
  - <u>OpenACC:</u> 1.1x – 72.7x
  - <u>PPCG(+ATF):</u> failed – 162.9x
  - *<u>TVM:</u> 1.0x – 172.5x*
  - *<u>NVIDIA cuBLAS:</u> 0.9x – 2.7x*
- **CPU:**
  - <u>OpenMP:</u> 1.7x - 82.0x
  - <u>Pluto(+ATF):</u> 1.4x – 14.3x
  - <u>Numba:</u> 11.6x – 3297.9x
  - *<u>TVM:</u> 1.0x – 6.1x*
  - *<u>Intel oneMKL:</u> 0.4x - 3.8x*

Experimental Results

Stencil Computations:

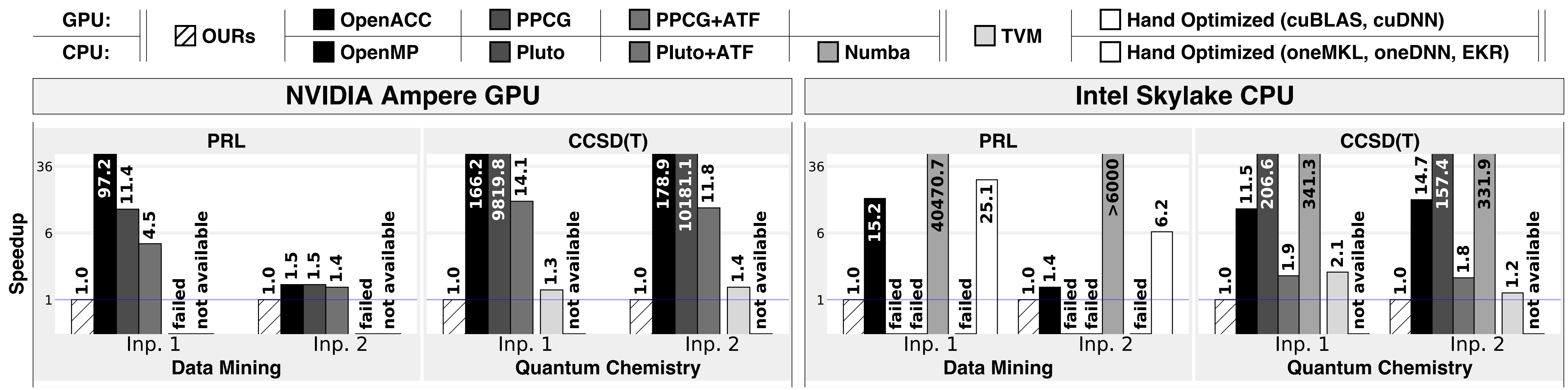


Highlights (*speedups* — higher is better for us):

- **GPU:**
  - <u>OpenACC:</u> **1.8x – 3.4x**
  - <u>PPCG(+ATF):</u> **1.0x – 5.3x**
  - ***<u>TVM:</u> 1.0x – 2.3x***
  - ***<u>NVIDIA cuDNN:</u> N/A – 5.3x***
- **CPU:**
  - <u>OpenMP:</u> **1.1x – 7.2x**
  - <u>Pluto(+ATF):</u> **1.1x – 3.7x**
  - <u>Numba:</u> **8.6x – 42.1x**
  - ***<u>TVM:</u> 1.0x – 2.5x***
  - ***<u>Intel oneDNN:</u> 2.6x – 6.5x***

Experimental Results

Data Mining & Quantum Chemistry:

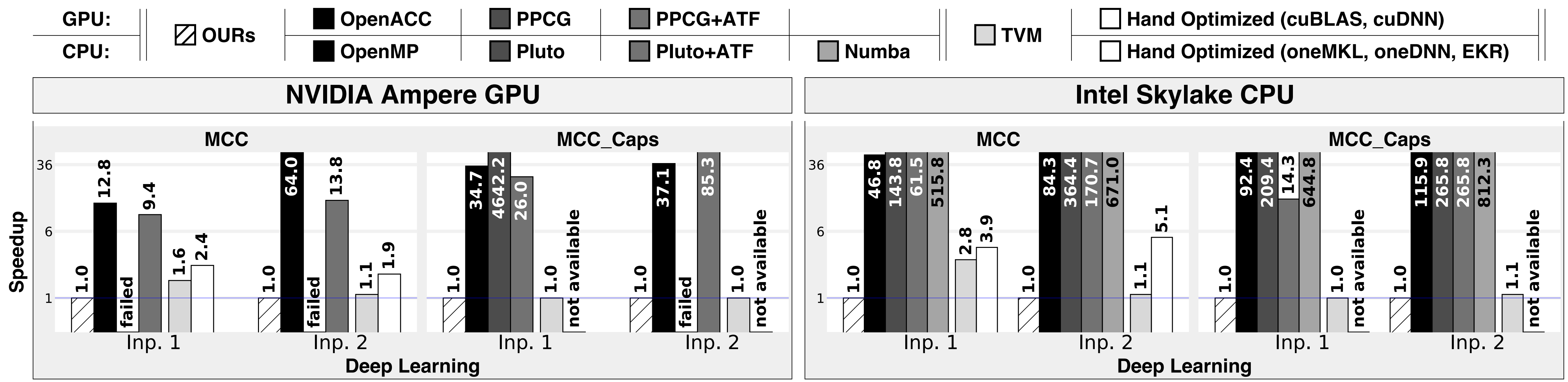


Highlights (*speedups* — higher is better for us):

- **GPU:**
  - OpenACC: **1.5x – 178.9x**
  - PPCG(+ATF): **1.5x – 10181.1x**
  - *TVM:* ***failed – 1.4x***
  - *Hand Optimized:* ***N/A***
- **CPU:**
  - OpenMP: **1.1x – 7.2x**
  - Pluto(+ATF): **1.1x – 3.7x**
  - Numba: **8.6x – >6000x**
  - *TVM:* ***1.0x – 2.5x***
  - *Intel oneDNN/EKR:* ***N/A – 25.1x***

Experimental Results

Deep Learning:



Highlights (*speedups* — higher is better for us):

- **GPU:**
  - <u>OpenACC:</u> **12.8x – 64.0x**
  - <u>PPCG(+ATF):</u> **failed – 4642.2x**
  - ***<u>TVM:</u> 1.0x – 1.6x***
  - ***<u>NVIDIA cuDNN:</u> N/A – 2.4x***
- **CPU:**
  - <u>OpenMP:</u> **46.8x – 115.9x**
  - <u>Pluto(+ATF):</u> **14.3x – 364.4x**
  - <u>Numba:</u> **515.8x – 812.3x**
  - ***<u>TVM:</u> 1.0x – 2.8x***
  - ***<u>Intel oneDNN:</u> N/A – 5.1x***

Experimental Results

Why MDH outperforms related approaches:

MDH vs OpenACC/OpenMP (*):

- Limited reduction support (e.g., for PRL)
- Rigid, heuristic-driven optimizations
- Limited tiling efficiency: manual tiling can improve performance, but is *complex*, *error-prone*, and *contrary to the directive-based philosophy*

MDH vs Numba (*):

- Missing semantic information about reductions
- Seems to not apply important optimizations in its generated code, e.g., tiling

MDH vs Polyhedral Compilers:

- Missing semantic information about reductions
- Polyhedral transformation often chosen toward too rigid optimization goals, e.g., always outer parallelization

MDH vs Vendor Libraries (*):

- Optimized toward average high performance over data characteristics
- MDH auto-tunes for particular sizes

(*) Performance results are difficult to interpret with certainty, as the generated assembly (PTX/LLVM) obscures the underlying optimization decisions

Higher performance through

efficient reduction handling and ***MDH-driven optimizations***

Conclusion

<u>We present a ***reduction-aware* directive** for optimizing data-parallel computations:</u>

- provided in the easy-to-use ***Python*** programming language
- supports **user-defined** reduction operators
- ***formally*** *grounded* in the MDH formalism
- experimental *real-world studies* show ***encouraging performance results***: e.g., speedups of up to 6.5x over hand-optimized libraries from NVIDIA and Intel

Our approach is ***reduction-aware***, not *reduction-focused* —
designed to achieve **high performance also for reduction-free computations**

<u>Future Work:</u>

Collaborate with the OpenMP/OpenACC community to incorporate reduction-awareness into these approaches.

Please feel free to reach out
if you are interested in collaborating!

Universität
Münster

Questions?

https://richardschulze.net
r.schulze@uni-muenster.de

Richard Schulze

https://mdh-lang.org

Code Generation

https://atf-tuner.org

Code Optimization

https://arirasch.net
a.rasch@uni-muenster.de

Ari Rasch

Distinctive Design Aspect

<u>The *scalar operation* is clearly separated from *reduction computations* in our approach:</u>

```
def matvec( T:BasicType, I:int,K:int ):
  @mdh( out( w = Buffer[T]                ) ,
        inp( M = Buffer[T], v = Buffer[T] ) ,
        combine_ops( cc, pw(add) )          )
  def mdh_matvec__T_I_K( w, M,v ):
    for i in range(I):
      for k in range(K):
        w[i] = M[i,k] * v[k]
  return mdh_matvec__T_I_K
```

<u>We use "=" instead of "+=":</u>

- loop body computes single point in the iteration space
- reductions are expressed through our directive
- aggregation across the iteration space is not encoded in the loop body

Such separation allows exploiting MDH-driven optimizations

Pre-Implemented Operators

CC

```
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
```

PW

```
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
```

PS

```
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
```