

Universität  
Münster



EuroLLVM 2024

# Linalg vs MDH: A Comparison of two MLIR Dialects

Jens Hunloh, Lars Hunloh, Richard Schulze  
Ari Rasch, Tobias Grosser



UNIVERSITY OF  
CAMBRIDGE

# Who are we?

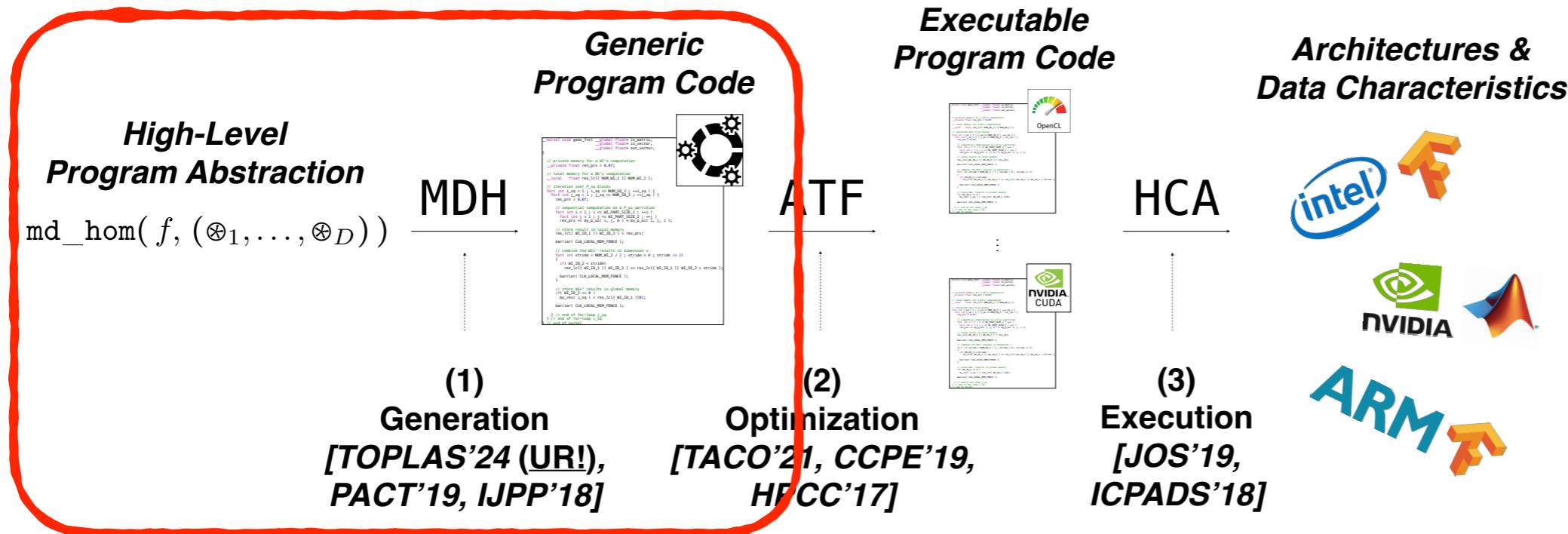
We are the developers of the **MDH+ATF+HCA** approaches:



Lars & Jens  
Hunloch



Richard Schulze



## Focus Today

A holistic approach to code generation (MDH) & optimization (ATF) & execution (HCA):

- (1) **MDH (Multi-Dimensional Homomorphisms)**: How to generate automatically optimizable (auto-tunable) code?
- (2) **ATF (Auto-Tuning Framework)**: How to optimize (auto-tune) code?
- (3) **HCA (Host Code Abstraction)**: How to execute code on (distr.) multi-dev. systems?



Ari Rasch

# Who are we?



UNIVERSITY OF  
CAMBRIDGE

## Primary objectives are:

- making compilation more modular, predictable, automatic, and trustworthy
- bringing open-source compiler innovation to an increasingly broad set of targets from GPUs over FPGAs to custom hardware, and
- breaking down the barriers between compilers and programmers by enabling their interaction via the programming language environment.



Tobias Grosser

**These are also objective of MDH**  
→ motivated collaboration with Tobias

**Tobias also has extensive experience with MLIR  
(whereas we are newcomers)**

# Agenda

## 1. Introduction to MDH (~5min), by Ari

- Brief overview of what MDH formalism is



## 2. MDH in MLIR (~5min), by Jens

- The MDH formalism implemented as MLIR dialect

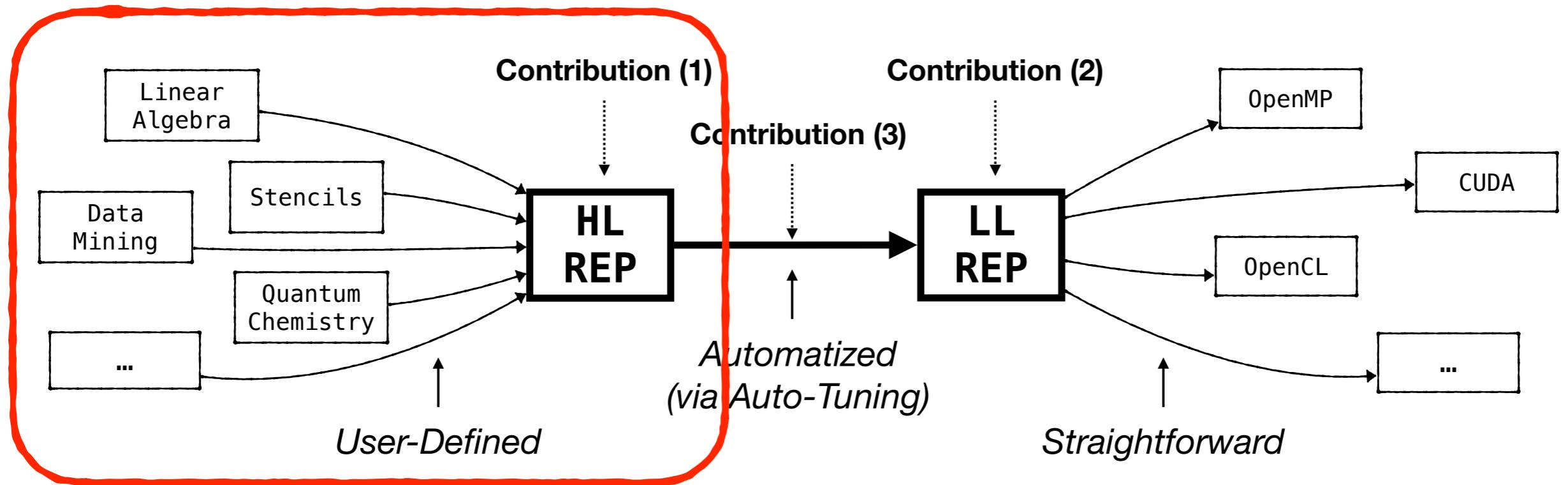


## 3. Linalg vs MDH (~5min), by Lars

- Comparison of Linalg with MDH's MLIR dialect



# The MDH Approach



**Focus Today**

The MDH approach [1] (formally) introduces:

**Note: We have implemented all three contributions into MLIR**

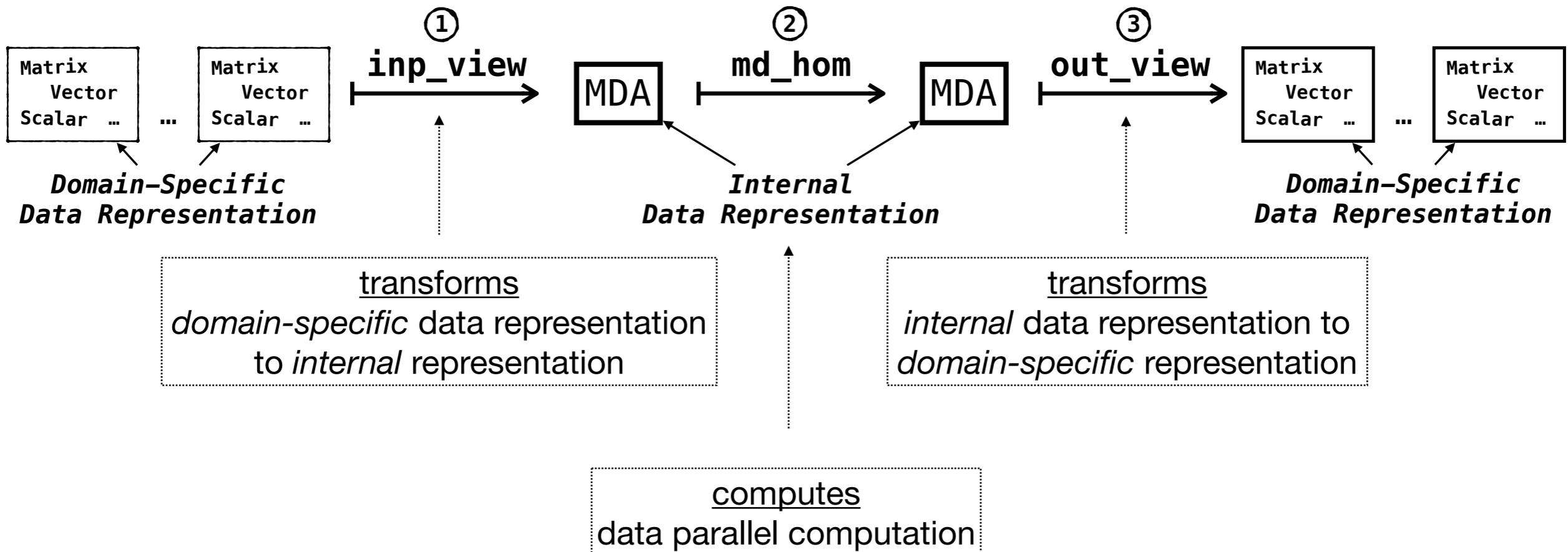
- (1) High-Level Program Representation for conveniently expressing data-parallel computations, agnostic from hardware and optimization details
- (2) Low-Level Program Representation that expresses device- and data-optimized de- and re-composition strategies of computations & straightforwardly transformable to executable program code
- (3) Lowering Process that *fully automatically* lowers a high-level MDH program to a device- and data-optimized low-level MDH program (based on auto-tuning [2])

[1] "(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms" (*under review at ACM TOPLAS*)

[2] "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)", *TACO'21*

# The MDH High-Level Representation

## Overview:



Our high-level representation expresses any data-parallel computation  
– *agnostic from hardware and optimization details* –  
using exactly *three, straightforwardly composed* higher-order functions only

# The MDH High-Level Representation

The MDH's high-level program representation illustrated:

```
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<sup>1</sup> for MatVec**

What is happening here:

- `inp_view` captures the accesses to input data
- `md_hom` expresses the data-parallel computation
- `out_view` captures the accesses to output data

<sup>1</sup>We can generate such MDH expressions also automatically from straightforward (annotated) C code [IMPACT'19]

# High-Level Representation

md_hom	f	$\otimes_1$	$\otimes_2$	$\otimes_3$	$\otimes_4$	Views	inp_view		out_view	
				A	B	C				
Dot	*	+	/	/	/	Dot	(k) $\mapsto$ (k)	(k) $\mapsto$ (k)	(k) $\mapsto$ ()	
MatVec	*	++	+	/	/	MatVec	(i,k) $\mapsto$ (i,k)	(i,k) $\mapsto$ (k)	(i,k) $\mapsto$ (i)	
MatMul	*	++	++	+	/	MatMul	(i,j,k) $\mapsto$ (i,k)	(i,j,k) $\mapsto$ (k,j)	(i,j,k) $\mapsto$ (i,j)	
MatMul <sup>T</sup>	*	++	++	+	/	MatMul <sup>T</sup>	(i,j,k) $\mapsto$ (k,i)	(i,j,k) $\mapsto$ (j,k)	(i,j,k) $\mapsto$ (j,i)	
bMatMul	*	++	++	++	+	bMatMul	(b,i,j,k) $\mapsto$ (b,i,k)	(b,i,j,k) $\mapsto$ (b,k,j)	(b,i,j,k) $\mapsto$ (b,i,j)	

md_hom	f	$\otimes_1$	$\otimes_2$	$\otimes_3$	$\otimes_4$	$\otimes_5$	$\otimes_6$	$\otimes_7$	$\otimes_8$	$\otimes_9$	$\otimes_{10}$
Conv2D	*	++	++	+	+	/	/	/	/	/	/
MCC	*	++	++	++	++	+	+	+	/	/	/
MCC_Capsule	*	++	++	++	++	+	+	+	++	++	+

Views	inp_view		out_view	
	A	B	C	
Views				
Dot	(k) $\mapsto$ (k)	(k) $\mapsto$ (k)	(k) $\mapsto$ ()	
MatVec	(i,k) $\mapsto$ (i,k)	(i,k) $\mapsto$ (k)	(i,k) $\mapsto$ (i)	
MatMul	(i,j,k) $\mapsto$ (i,k)	(i,j,k) $\mapsto$ (k,j)	(i,j,k) $\mapsto$ (i,j)	
MatMul <sup>T</sup>	(i,j,k) $\mapsto$ (k,i)	(i,j,k) $\mapsto$ (j,k)	(i,j,k) $\mapsto$ (j,i)	
bMatMul	(b,i,j,k) $\mapsto$ (b,i,k)	(b,i,j,k) $\mapsto$ (b,k,j)	(b,i,j,k) $\mapsto$ (b,i,j)	

Views	inp_view		out_view	
	I	F		O
Views				
Conv2D	(p,q,r,s) $\mapsto$ (p+r,q+s)	(p,q,r,s) $\mapsto$ (r,s)	(p,q,r,s) $\mapsto$ (p,q)	
MCC	(n,p,...) $\mapsto$ (n,p+r,q+s,c)	(n,p,...) $\mapsto$ (k,r,s,c)	(n,p,...) $\mapsto$ (n,p,q,k)	
MCC_Capsule	(n,p,...) $\mapsto$ (n,p+r,q+s,c,cm,ck)	(n,p,...) $\mapsto$ (k,r,s,c,ck,cn)	(n,p,...) $\mapsto$ (n,p,q,k,cm,cn)	

md_hom	f	$\otimes_1$	$\otimes_2$	Views	inp_view		out_view	
					I	0		
MBBS	id	$\text{++prefix-sum}(+)$	+	MBBS	(i,j) $\mapsto$ (i,j)	(i) $\mapsto$ (i)		
MBBS								

md_hom	f	$\otimes_1$	$\otimes_2$	Views	inp_view		out_view	
					I	0		
Jacobi1D	J <sub>1D</sub>	++	/	Jacobi1D	(i) $\mapsto$ (i+0), (i) $\mapsto$ (i+1), (i) $\mapsto$ (i+2)	(i) $\mapsto$ (i)		
Jacobi2D	J <sub>2D</sub>	++	++	Jacobi2D	(i,j) $\mapsto$ (i,j+1), (i,j) $\mapsto$ (i+1,j), ...	(i,j) $\mapsto$ (i,j)		
Jacobi1D								
Jacobi2D								

md_hom	f	$\otimes_1$	$\otimes_2$	Views	inp_view		out_view	
					I	0		
PRL	wght	$\text{++}$	$\text{max}_{\text{PRL}}$	PRL	(i,j) $\mapsto$ (i)	(i,j) $\mapsto$ (j)	(i,j) $\mapsto$ (i)	
PRL								

md_hom	f	$\otimes_1$	Views	inp_view		out_view	
				I	0		
scan( $\oplus$ )	id	$\text{++prefix-sum}(\oplus)$	scan( $\oplus$ )	(i) $\mapsto$ (i)	(i) $\mapsto$ (i)		
scan( $\oplus$ )							

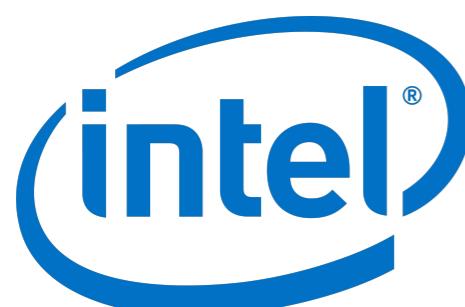
md_hom	f	$\otimes_1$	$\otimes_2$	Views	inp_view		out_view	
					Bins	Elems		
Histo	f <sub>Histo</sub>	$\text{++}$	+	Histo	(b,e) $\mapsto$ (b)	(b,e) $\mapsto$ (e)	(b,e) $\mapsto$ (b)	
Histo								

**MDH is capable of expressing various kinds of data-parallel computations [1]**

# Experimental Results

**Highlights only  
(for DL)**

Deep Learning	NVIDIA Ampere GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.00	1.26	1.05	2.22	0.93	1.42	0.88	1.14	0.94	1.00
PPCG	3456.16	8.26	-	7.89	1661.14	7.06	5.77	5.08	2254.67	7.55
PPCG+ATF	3.28	2.58	13.76	5.44	4.26	3.92	9.46	3.73	3.31	10.71
cuDNN	0.92	-	1.85	-	1.22	-	1.94	-	1.81	2.14
cuBLAS	-	1.58	-	2.67	-	0.93	-	1.04	-	-
cuBLASEx	-	1.47	-	2.56	-	0.92	-	1.02	-	-
cuBLASLt	-	1.26	-	1.22	-	0.91	-	1.01	-	-



Deep Learning	Intel Skylake CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.05	1.14	1.20	1.97	1.14	2.38	1.27	3.01	1.40
Pluto	355.81	49.57	364.43	13.93	130.80	93.21	186.25	36.30	152.14	75.37
Pluto+ATF	13.08	19.70	170.69	6.57	3.11	6.29	53.61	8.29	3.50	25.41
oneDNN	0.39	-	5.07	-	1.22	-	9.01	-	1.05	4.20
oneMKL	-	0.44	-	1.09	-	0.88	-	0.53	-	-
oneMKL(JIT)	-	6.43	-	8.33	-	27.09	-	9.78	-	-

**MDH achieves encouraging experimental results [1]**

# MDH in MLIR



MLIR is a compiler framework that offers a solid, uniform infrastructure for compiler developers to conveniently design and implement *Domain-Specific Abstractions* (a.k.a. *dialect* in MLIR terminology)



## (Potential) advantages of an MDH dialect for MLIR:

### 1. Expressivity:

MDH targets data-parallel computations in general [1]

### 2. Code Generation:

MDH (formally) describes its lowering to imperative-style code [1]

### 3. Performance:

MDH achieves performance competitive to hand-optimized solutions (e.g., cuBLAS/cuDNN & oneMKL/oneDNN) [1]

### 4. Portability:

MDH offers a (formal) recipe for targeting new parallel architectures [1]



**Implemented  
by Lars & Jens Hunloch**

**We aim to contribute to MLIR!**

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

*in MDH Formalism*

MatVec

*in MDH MLIR*

*in C++*

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [ [ affine_map<( i,k ) -> ( i,k )> ],  
          [ affine_map<( i,k ) -> ( k )> ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [ [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128, 64 ],  
        out_types = [ f32 ]  
    }(%M, %v):  
    ( memref<128x64xf32>, memref<64xf32> )  
        -> memref<128xf32>
```

MDH-HL-MLIR



# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

Goal

Implementing MDH into MLIR,  
as close as possible to the  
formalism

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k ) > ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128,64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
    ( memref<128x64xf32>,memref<64xf32> )  
        -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

### Accesses to Input Data

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k ) > ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128,64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
    ( memref<128x64xf32>,memref<64xf32> )  
        -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
    md_hom<128,64>( *, (#+,+) )  
        inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

### Accesses to Output Data

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k )> ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128, 64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
( memref<128x64xf32>,memref<64xf32> )  
                    -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

### Scalar Function

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k )> ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128, 64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
    ( memref<128x64xf32>,memref<64xf32> )  
        -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
    md_hom<128,64>( *, (#+,+))  
        inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```



### Combine Operators

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k ) > ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128,64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
    ( memref<128x64xf32>,memref<64xf32> )  
        -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```

MDH

### Iteration Space

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k ) > ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128, 64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
( memref<128x64xf32>,memref<64xf32> )  
                    -> memref<128xf32>
```



MDH-HL-MLIR

# MDH in MLIR

## Introductory Example – MatVec:

```
out_view<f32>( w:(i,k)↔(i) )  
  md_hom<128,64>( *, (#+,+) )  
    inp_view<f32,f32>( M:(i,k)↔(i,k) , v:(i,k)↔(k) )
```



### Data Types

```
void MatVec( float[] M, float[] v, float[] w)  
{  
    for( int i=0 ; i < 128 ; ++i )  
        for( int k=0 ; k < 64 ; ++k )  
            w[i] += M[i][k] * v[k];  
}
```



```
func.func @main()  
{  
    %M = memref.alloc() : memref<128x64xf32>  
    %v = memref.alloc() : memref<64xf32>  
  
    %w = mdh.compute "mdh_matvec"  
    {  
        inp_view =  
        [  
            [ affine_map<( i,k ) -> ( i,k )> ],  
            [ affine_map<( i,k ) -> ( k ) > ]  
        ],  
  
        md_hom =  
        {  
            scalar_func = @mul,  
            combine_ops = [ "cc", ["pw", @add] ]  
        },  
  
        out_view =  
        [  
            [ affine_map<( i,k ) -> ( i )> ]  
        ]  
    }  
    {  
        inp_types = [ f32, f32 ],  
        mda_size = [ 128,64 ],  
        out_types = [ f32 ]  
    }(%M,%v):  
    ( memref<128x64xf32>,memref<64xf32> )  
        -> memref<128xf32>
```



MDH-HL-MLIR

# Quick Reminder: LinAlg



## LinAlg MLIR Dialect:

```
#map1 = affine_map<(d0, d1) -> (d0, d1)> LinAlg
#map2 = affine_map<(d0, d1) -> (d1)      >
#map3 = affine_map<(d0, d1) -> (d0)      >
module {
  func.func @main() {
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>
    %w = memref.alloc() : memref<128xf32>
    linalg.generic
    {
      indexing_maps = [#map1, #map2, #map3],
      iterator_types = ["parallel", "reduction"]
    } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
      outs(%w:memref<128xf32>) {
        ^bb0(%in_1: f32, %in_2: f32, %out: f32):
          %0 = arith.mulf %in_1, %in_2 : f32
          %1 = arith.addf %out, %0 : f32
          linalg.yield %1 : f32
      }
      return
    }
}
```

**Quick reminder “LinAlg”,  
before comparison  
“LinAlg vs. MDH”**

# Quick Reminder: Linalg



## Linalg MLIR Dialect:

```
#map1 = affine_map<(d0, d1) -> (d0, d1)>
#map2 = affine_map<(d0, d1) -> (d1)      >
#map3 = affine_map<(d0, d1) -> (d0)      >

module {
  func.func @main() {
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>
    %w = memref.alloc() : memref<128xf32>
    linalg.generic
    {
      indexing_maps = [#map1, #map2, #map3],
      iterator_types = ["parallel", "reduction"]
    } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
      outs(%w:memref<128xf32>) {
        ^bb0(%in_1: f32, %in_2: f32, %out: f32):
          %0 = arith.mulf %in_1, %in_2 : f32
          %1 = arith.addf %out, %0 : f32
          linalg.yield %1 : f32
      }
      return
    }
}
```

**Linalg**

**Accesses to  
Input/Output Data**

# Quick Reminder: LinAlg



## LinAlg MLIR Dialect:

```
#map1 = affine_map<(d0, d1) -> (d0, d1)> LinAlg
#map2 = affine_map<(d0, d1) -> (d1)      >
#map3 = affine_map<(d0, d1) -> (d0)      >
module {
  func.func @main() {
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>
    %w = memref.alloc() : memref<128xf32>
    linalg.generic
    {
      indexing_maps = [#map1, #map2, #map3],
      iterator_types = ["parallel", "reduction"]
    } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
      outs(%w:memref<128xf32>)
      ^bb0(%in_1: f32, %in_2: f32, %out: f32):
        %0 = arith.mulf %in_1, %in_2 : f32
        %1 = arith.addf %out, %0 : f32
        linalg.yield %1 : f32
    }
    return
  }
}
```

## Iteration Space Specification

# Quick Reminder: Linalg



## Linalg MLIR Dialect:

```
#map1 = affine_map<(d0, d1) -> (d0, d1)> Linalg
#map2 = affine_map<(d0, d1) -> (d1)      >
#map3 = affine_map<(d0, d1) -> (d0)      >
module {
  func.func @main() {
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>
    %w = memref.alloc() : memref<128xf32>
    linalg.generic
    {
      indexing_maps = [#map1, #map2, #map3],
      iterator_types = ["parallel", "reduction"]
    } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
      outs(%w:memref<128xf32>) {
        ^bb0(%in_1: f32, %in_2: f32, %out: f32):
          %0 = arith.mulf %in_1, %in_2 : f32
          %1 = arith.addf %out, %0 : f32
          linalg.yield %1 : f32
      }
      return
    }
}
```

**MatVec Computation  
(mul and add)**

# Comparison: LinAlg vs MDH

```
#map1 = affine_map<(d0, d1) -> (d0, d1)>
#map2 = affine_map<(d0, d1) -> (d1)      >
#map3 = affine_map<(d0, d1) -> (d0)      >
module {
    func.func @main() {
        %M = memref.alloc() : memref<128x64xf32>
        %v = memref.alloc() : memref<64xf32>
        %w = memref.alloc() : memref<128xf32>
        linalg.generic
        {
            indexing_maps = [#map1, #map2, #map3],
            iterator_types = ["parallel", "reduction"]
        } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
        outs(%w:memref<128xf32>) {
            ^bb0(%in_1: f32, %in_2: f32, %out: f32):
                %0 = arith.mulf %in_1, %in_2 : f32
                %1 = arith.addf %out, %0 : f32
                linalg.yield %1 : f32
        }
        return
    }
}
```

**LinAlg**

```
func.func @main()
{
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>

    %w = mdh.compute "mdh_matvec"
    {
        inp_view =
        [
            [ affine_map<( i,k ) -> ( i,k )> ],
            [ affine_map<( i,k ) -> ( k ) > ]
        ],
        md_hom =
        {
            scalar_func = @mul,
            combine_ops = [ "cc", ["pw",@add] ]
        },
        out_view =
        [
            [ affine_map<( i,k ) -> ( i )> ]
        ]
    }
    inp_types = [ f32, f32 ],
    mda_size = [ 128,64 ],
    out_types = [ f32 ]
}(%M,%v):( memref<128x64xf32>,memref<64xf32> )
-> memref<128xf32>

return
}
```

**MDH**

**Significant design difference:**

MDH **separates** the **scalar operation** (e.g., mul)  
from the **operations for combining intermediate  
results** (e.g., add)

Note: MatVec is a simple example!

# Comparison: Linalg vs MDH

## Advantages we see in MDH Design:

1. **Performance:** parallelizing & optimizing also reduction-like parts within the computation

**MDH can parallelize & optimize also 2<sup>nd</sup> dimension ( $\otimes_2$ )**

The diagram illustrates the computation of a matrix-vector product  $M \cdot v$ . On the left, a matrix  $M$  (with dimensions  $I \times K$ ) and a vector  $v$  (with dimension  $K$ ) are shown. An arrow labeled "MatVec" points to the right, where the multiplication is performed. The result is a vector  $w$  (with dimension  $I$ ). The computation is visualized as follows: the matrix  $M$  is transformed into a row vector of column vectors, each being a function  $f(M_{i,:}, v)$  applied to the vector  $v$ . This row vector is then multiplied by the vector  $v$  using a reduction operation  $\otimes_1$  along the second dimension. A separate reduction operation  $\otimes_2$  is shown above, indicating that the computation can also be parallelized across the second dimension.

$$\begin{pmatrix} M_{1,1} & \dots & M_{1,K} \\ \vdots & \ddots & \vdots \\ M_{I,1} & \dots & M_{I,K} \end{pmatrix}, \begin{pmatrix} v_1 \\ \vdots \\ v_K \end{pmatrix} \xrightarrow{\text{MatVec}} \begin{pmatrix} f(M_{1,1}, v_1) & \dots & f(M_{1,K}, v_K) \\ \vdots & \ddots & \vdots \\ f(M_{I,1}, v_1) & \dots & f(M_{I,K}, v_K) \end{pmatrix} = \begin{pmatrix} M_{1,1} * v_1 + \dots + M_{1,K} * v_K \\ \vdots \\ M_{I,1} * v_1 + \dots + M_{I,K} * v_K \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_I \end{pmatrix}$$

Matrix-Vector Multiplication

Q1

How does Linalg parallelize and optimize  
reduction-heavy computations?

# Comparison: Linalg vs MDH

## Advantages we see in MDH Design:

2. **Naturality:** i) avoiding *unnecessary memory accesses* (e.g., Linalg requires 0-initialized output vector for MatVec) and ii) *not requiring neutral elements* for combine ops

```
// ...
module {
    func.func @main() {
        %M = memref.alloc() : memref<128x64xf32>
        %v = memref.alloc() : memref<64xf32>
        %w = memref.alloc() : memref<128xf32>
        linalg.generic
        { /* ... */
            ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
            outs(%w:memref<128xf32>)
        { /* ... */
        return
    }
}
```

Linalg

Needs existence and  
initialization with neutral  
element of combine op “+”  
(which is “0”)

Q2

Can Linalg express  $w=M*v$ ,  
instead of  $w+=M*v$ ?

# Comparison: Linalg vs MDH

## Advantages we see in MDH Design:

3. Expressivity: expressing also more advanced computations (whose reduction dimensions rely on different kinds of operators)

```
#parallel for reduce ⊕₁  
for( ... ) {  
    #parallel for reduce ⊕₂  
    for( ... ) {  
        // ...  
        out_2 ⊕₂= foo(...)  
    }  
    out_1 ⊕₁= out_2;  
}
```

Intermediate results of loops are combined using different combine operators (e.g., MBBS example [3])

[3] Farzan, Nicolet, "Modular Divide-and-Conquer Parallelization of Nested Loops", PLDI19

Q3

Can Linalg express loops relying on different combine operators?

# Comparison: Linalg vs MDH

## Summary – Questions to Linalg community:

1. How does Linalg parallelize and optimize reduction-heavy computations?
2. Can Linalg express  $w = M * v$ , instead of  $w += M * v$ ?
3. Can Linalg express loops relying on different combine operators?

## Further Questions:

1. Why not explicitly request iteration space size?  
→ *Convenient, but cannot be computed from buffer sizes in the general case*

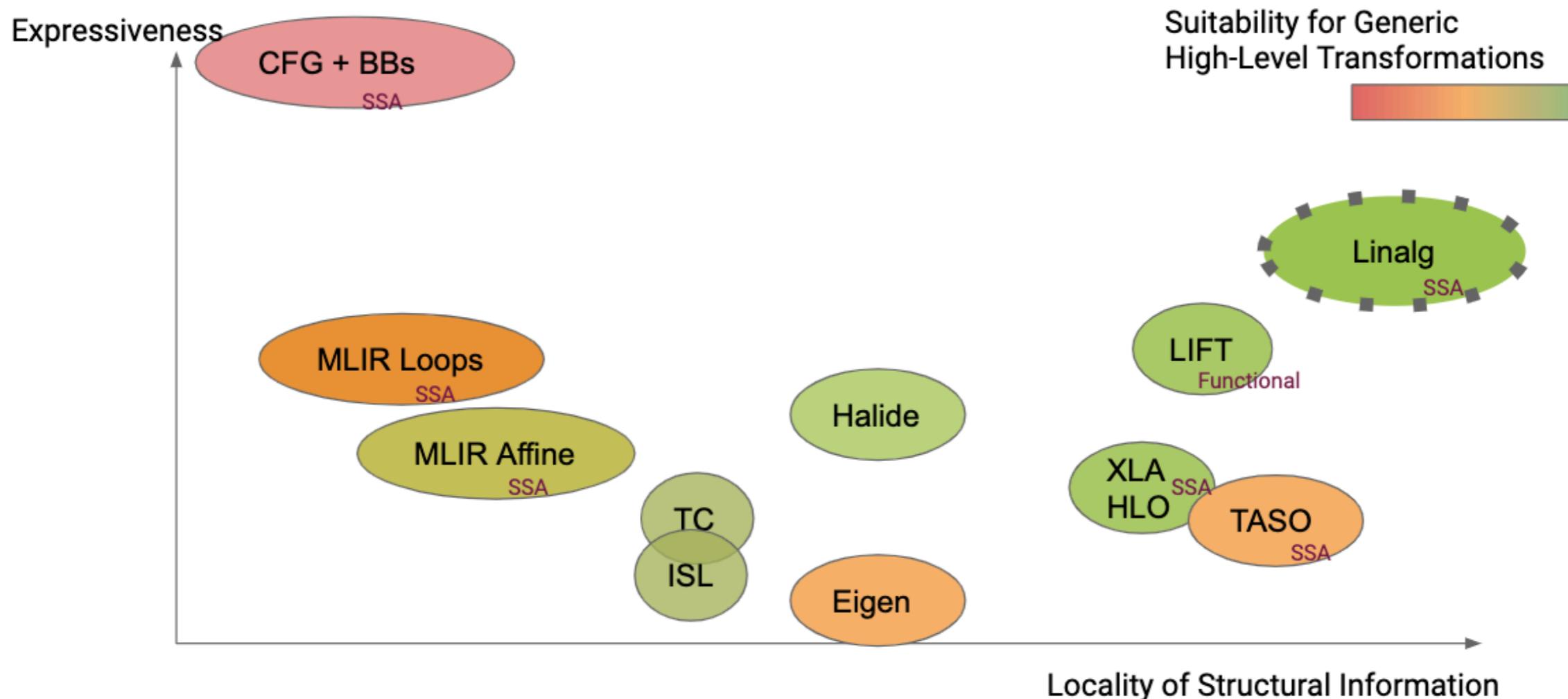
**Note: Exploiting MDH design for code generation is complex,  
but elaborated and (formally) explained in [1]**

# Comparison: Linalg vs MDH

## Questions to Linalg community:

### **Summary of Existing Alternatives a Picture**

Lastly, we summarize our observations of lessons from [Prior Art](#)—when viewed under the lense of our [Core Guiding Principles](#) — with the following picture.



This figure is not meant to be perfectly accurate but a rough map of how we view the distribution of structural information in existing systems, from a codegen-friendly angle. Unsurprisingly, the [Linalg Dialect](#) and its future evolutions aspire to a position in the top-right of this map.

***"Linalg Dialect Rationale: The Case For Compiler-Friendly Custom Operations"***

**Where would you place MDH?**

# Questions?



Ari Rasch



Lars & Jens  
Hunloh



We are grateful for  
any kind of feedback



<https://mdh-lang.org>



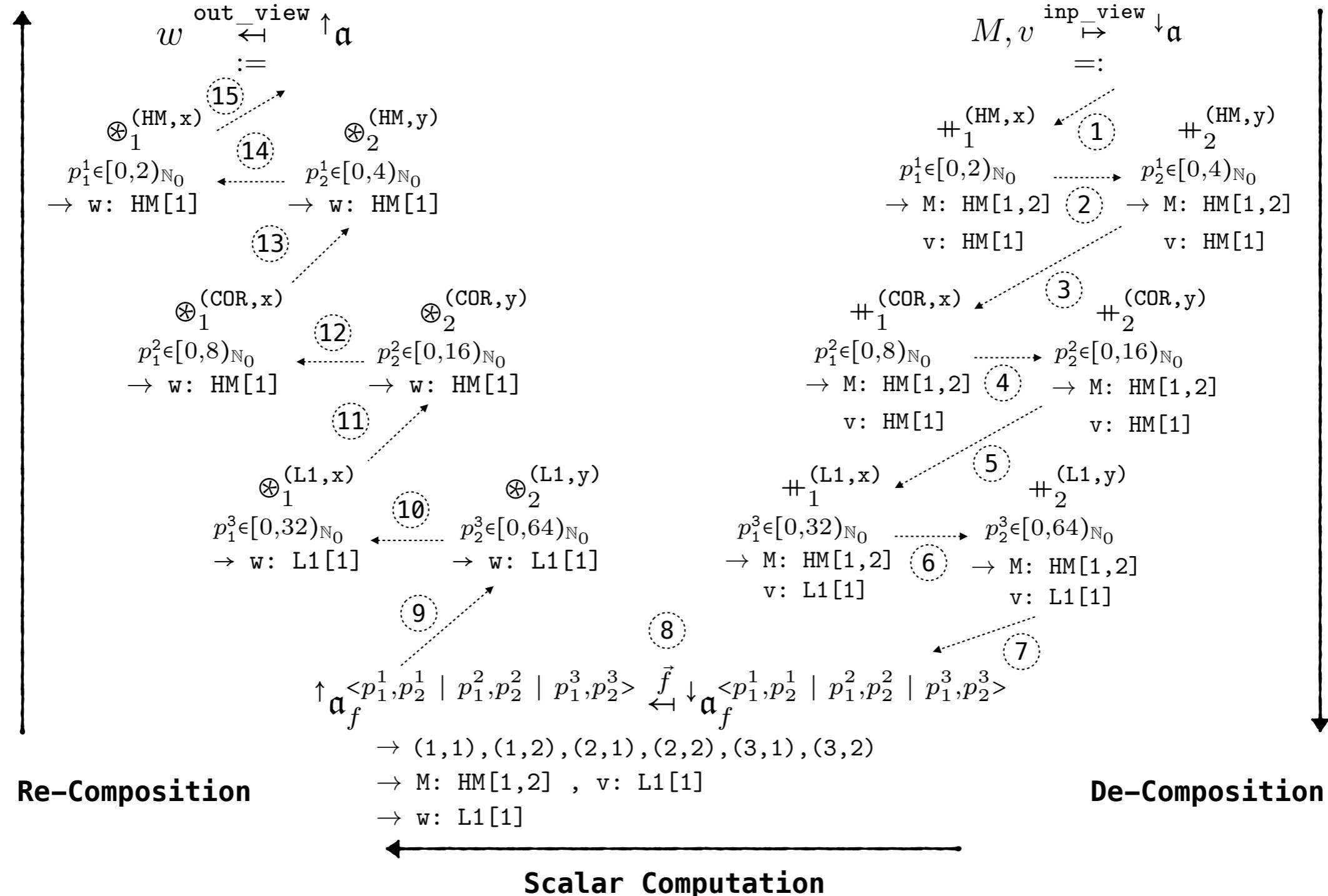
<https://atf-tuner.org>



<https://hca-project.org>

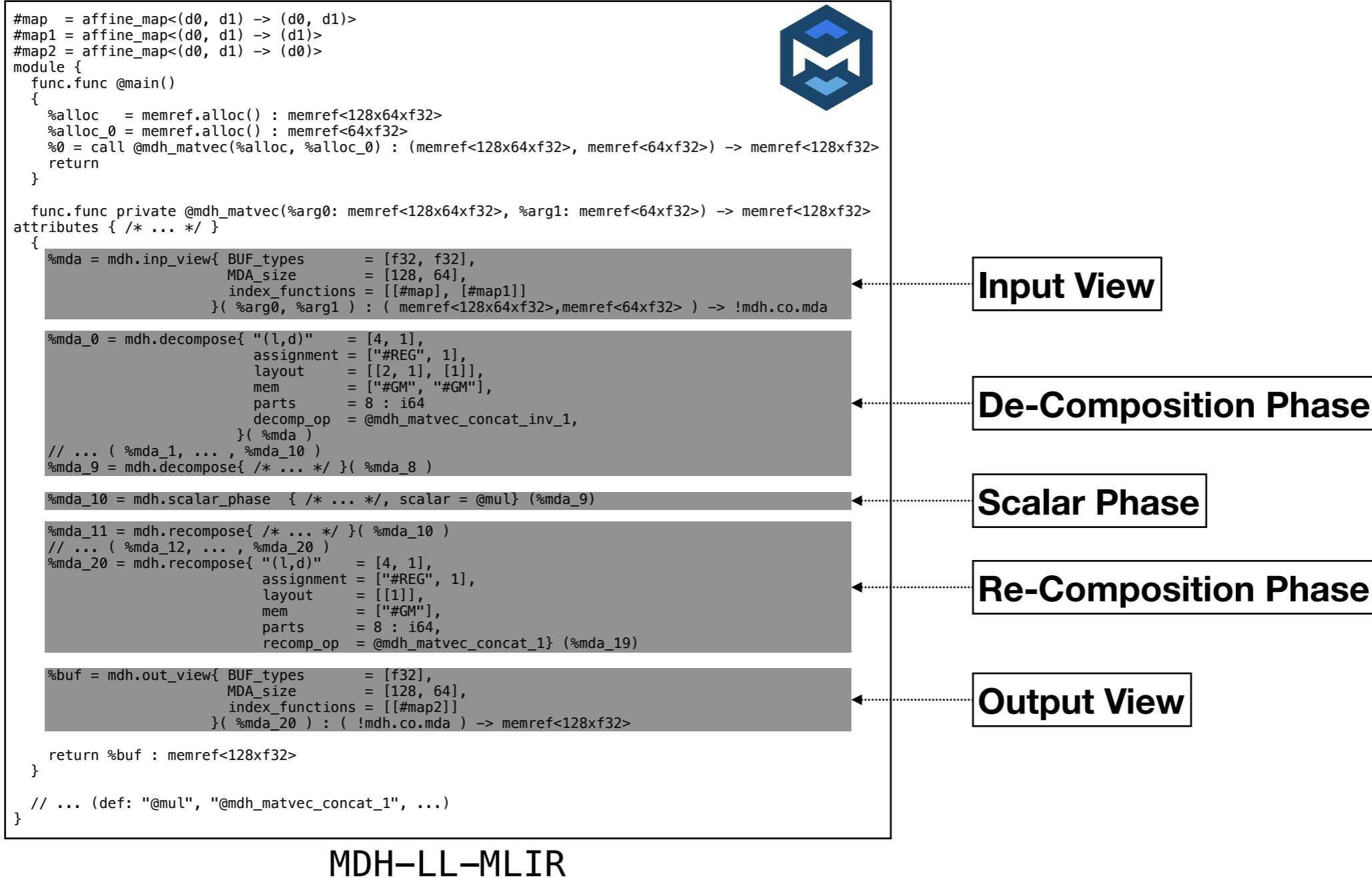
# Excursion: MDH Low-Level Representation

MDH's Low-Level Representation expresses a (de/re)-composition of a computation [1]:



# Excursion: MDH Low-Level Representation

## MDH Low-Level Representation in MLIR:



The MDH-LL-MLIR *Dialect*  
implements MDH's formal low-level Representation

# Excursion: MDH Lowering

## Lowering: MDH-HL-MLIR → MDH-LL-MLIR

```
func.func @main()
{
  %M = memref.alloc() : memref<128x64xf32>
  %v = memref.alloc() : memref<64xf32>

  %w = mdh.compute "mdh_matvec"
  {
    inp_view =
    [
      [ affine_map<( i,k ) -> ( i,k )> ],
      [ affine_map<( i,k ) -> ( k ) > ]
    ],
    md_hom =
    {
      scalar_func = @mul,
      combine_ops = [ "cc", ["pw",@add] ]
    },
    out_view =
    [
      [ affine_map<( i,k ) -> ( i )> ]
    }
    {
      inp_types = [ f32, f32 ],
      mda_size = [ 128, 64 ],
      out_types = [ f32 ]
    }(%M,%v):( memref<128x64xf32>,memref<64xf32> )
      -> memref<128xf32>
  }
  return
}
```



Lowering  
→  
HL → LL  
↑  
using  
Auto-Tuning [2]

```
#map  = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
#map2 = affine_map<(d0, d1) -> (d0)>
module {
  func.func @main()
  {
    %alloc  = memref.alloc() : memref<128x64xf32>
    %alloc_0 = memref.alloc() : memref<64xf32>
    %0 = call @mdh_matvec(%alloc, %alloc_0) : (memref<128x64xf32>, memref<64xf32>) -> memref<128xf32>
    return
  }

  func.func private @mdh_matvec(%arg0: memref<128x64xf32>, %arg1: memref<64xf32>) -> memref<128xf32> attributes { /* ... */ }
  {
    %mda = mdh.inp_view{ BUF_types      = [f32, f32],
                         MDA_size       = [128, 64],
                         index_functions = [[#map], [#map1]] }
    (%arg0, %arg1) : ( memref<128x64xf32>,memref<64xf32> ) -> !mdh.co.mda

    %mda_0 = mdh.decompose{ "(l,d)"      = [4, 1],
                           assignment     = ["#REG", 1],
                           layout         = [[2, 1], [1]],
                           mem            = ["#GM", "#GM"],
                           parts          = 8 : i64
                           decomp_op     = @mdh_matvec_concat_inv_1,
                           (%mda) }
    // ... ( %mda_1, ... , %mda_10 )
    %mda_9 = mdh.decompose{ /* ... */ }(%mda_8)

    %mda_10 = mdh.scalar_phase { /* ... */, scalar = @mul} (%mda_9)
    %mda_11 = mdh.recompose{ /* ... */ }(%mda_10)
    // ... ( %mda_12, ... , %mda_20 )
    %mda_20 = mdh.recompose{ "(l,d)"      = [4, 1],
                           assignment     = ["#REG", 1],
                           layout         = [[1]],
                           mem            = ["#GM"],
                           parts          = 8 : i64,
                           recomp_op     = @mdh_matvec_concat_1} (%mda_19)

    %buf = mdh.out_view{ BUF_types      = [f32],
                         MDA_size       = [128, 64],
                         index_functions = [[#map2]] }
    (%mda_20) : ( !mdh.co.mda ) -> memref<128xf32>
    return %buf : memref<128xf32>
  }
  // ... (def: "@mul", "@mdh_matvec_concat_1", ...)
}
```

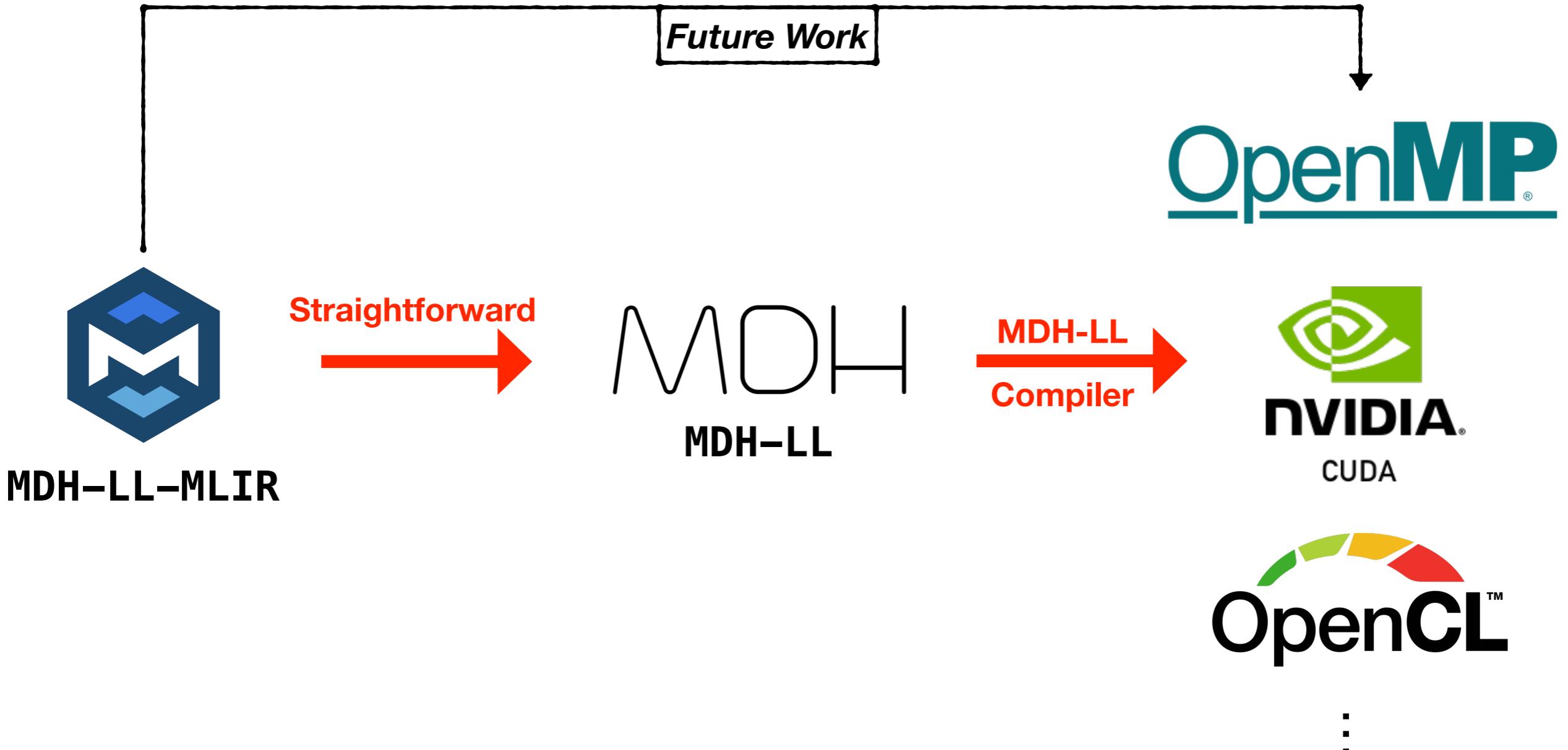


## MDH-HL-MLIR

## MDH-LL-MLIR

# Excursion: MDH Lowering

## Code Generation:



Our future work aims to implement our Code Generation also in MLIR (e.g., to be independent of MDH compiler, benefit from assembly level optimizations, ...)