Universität Münster

EuroLLVM 2024

Linalg vs MDH: A Comparison of two MLIR Dialects

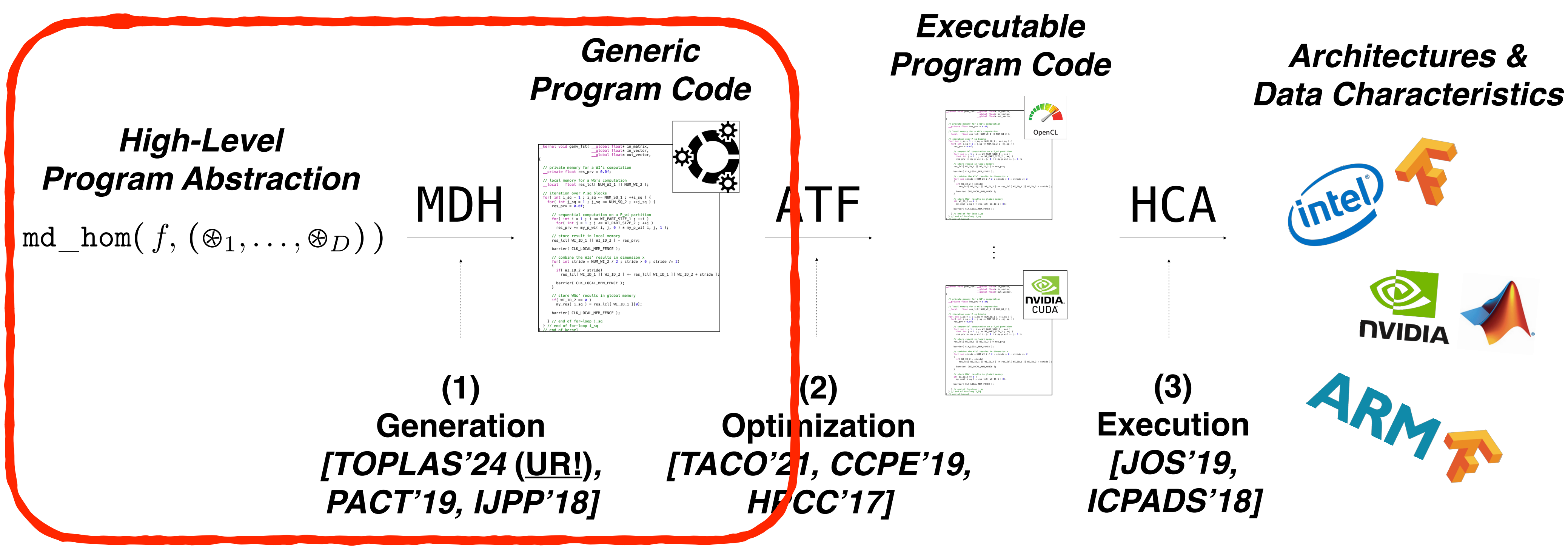
Jens Hunloh, Lars Hunloh, Richard Schulze
Ari Rasch, Tobias Grosser

UNIVERSITY OF CAMBRIDGE

Who are we?

Lars & Jens Hunloh

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



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 <u>Ari</u>**

   → Brief overview of what MDH formalism is

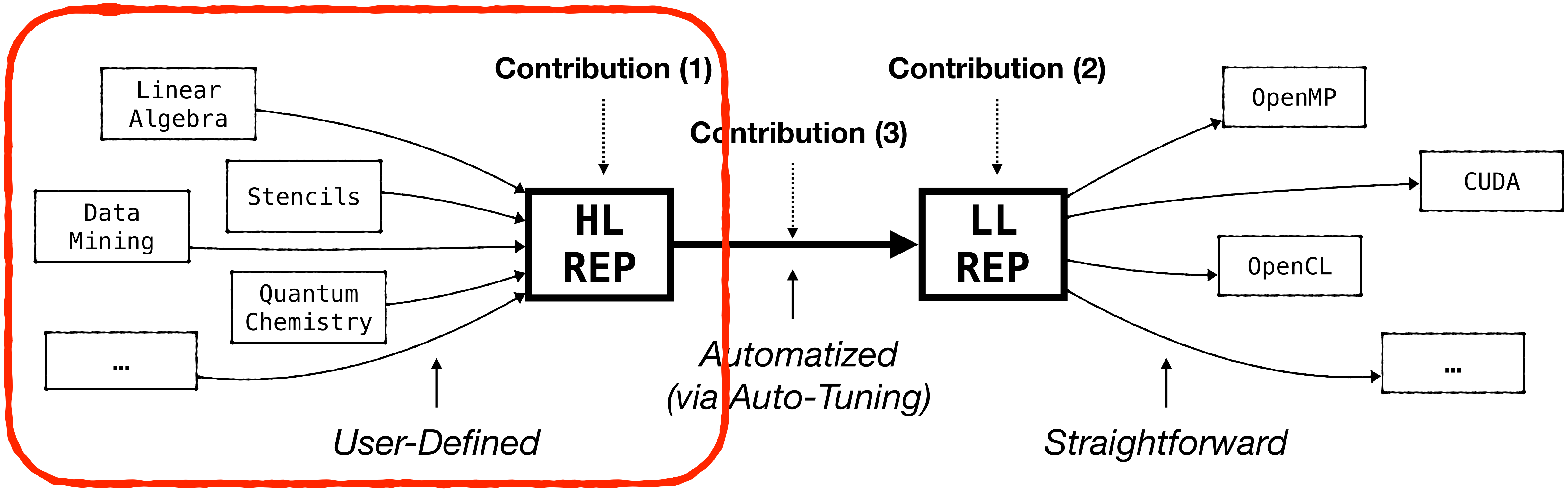
2. **MDH in MLIR (~5min), by <u>Jens</u>**

   → The MDH formalism implemented as MLIR dialect

3. **Linalg vs MDH (~5min), by <u>Lars</u>**

   → Comparison of Linalg with MDH's MLIR dialect

The MDH Approach



Focus Today

Note: We have implemented all three contributions into MLIR

The MDH approach [1] (formally) introduces:

(1) High-Level Program Representation for conveniently expressing data-parallel computations, agnostic from hardware and optimization details

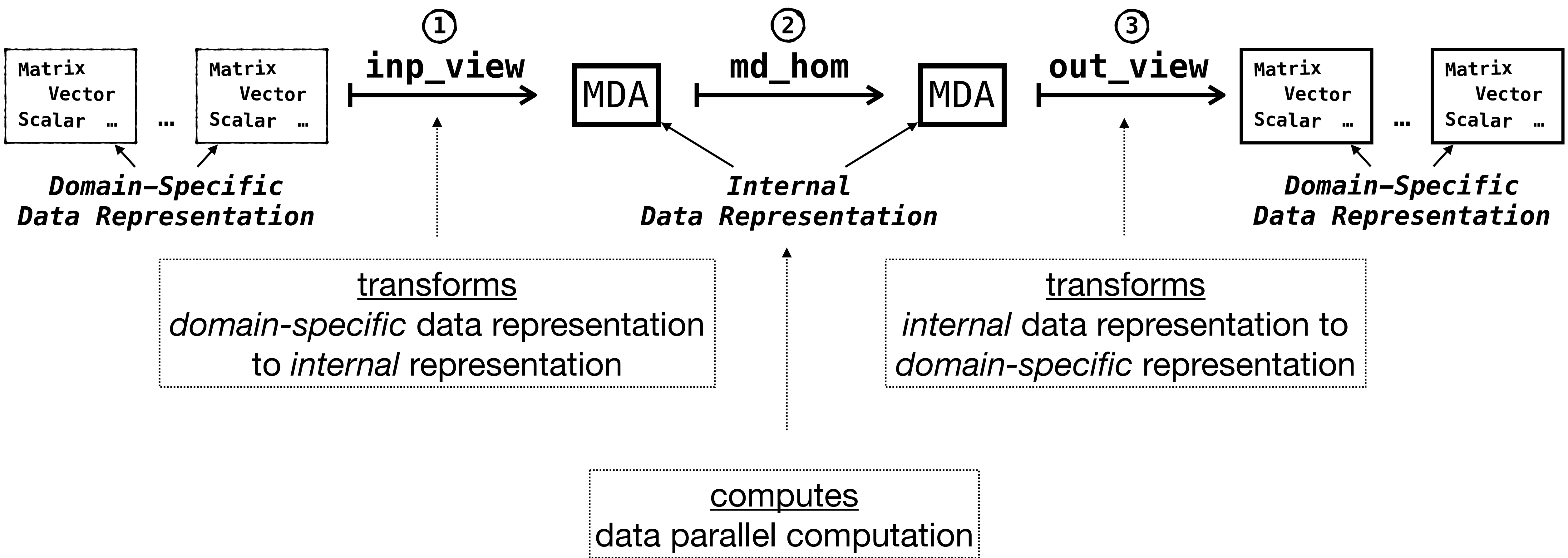
(2) Low-Level Program Representation that expresses device- and data-optimized de- and re-composition strategies of computations & straightforwardly transformable to executable program code

(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:

$$\texttt{MatVec}^{<T\in\texttt{TYPE}\,|\,I,K\in\mathbb{N}>} := \texttt{out\_view}<T>(\ w:(i,k)\mapsto(i)\ ) \circ \texttt{md\_hom}<I,K>(\ *,\ (+\!\!+,+)\ ) \circ \texttt{inp\_view}<T,T>(\ M:(i,k)\mapsto(i,k)\ ,\ v:(i,k)\mapsto(k)\ )$$

MDH High-Level Representation[1] 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

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

High-Level Representation

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$
Dot	*	+	/	/	/
MatVec	*	++	+	/	/
MatMul	*	++	++	+	/
MatMul^T	*	++	++	+	/
bMatMul	*	++	++	++	+

Views	inp_view		out_view
	A	B	C
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^T	(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)

1) Linear Algebra Routines

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$	$\circledast_5$	$\circledast_6$	$\circledast_7$	$\circledast_8$	$\circledast_9$	$\circledast_{10}$
Conv2D	*	++	++	+	+	/	/	/	/	/	/
MCC	*	++	++	++	++	+	+	+	/	/	/
MCC_Capsule	*	++	++	++	++	+	+	+	++	++	+

Views	inp_view		out_view
	I	F	O
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)

2) Convolution Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
Jacobi1D	J_{1D}	++	/
Jacobi2D	J_{2D}	++	++

Views	inp_view	out_view
	I	O
Jacobi1D	(i) $\mapsto$ (i+0) , (i) $\mapsto$ (i+1) , (i) $\mapsto$ (i+2)	(i) $\mapsto$ (i)
Jacobi2D	(i,j) $\mapsto$ (i,j+1) , (i,j) $\mapsto$ (i+1,j) , ...	(i,j) $\mapsto$ (i,j)

3) Jacobi Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
PRL	wght	++	max_{PRL}

Views	inp_view		out_view
	N	E	M
PRL	(i,j) $\mapsto$ (i)	(i,j) $\mapsto$ (j)	(i,j) $\mapsto$ (i)

4) Probabilistic Record Linkage

md_hom	f	$\circledast_1$	$\circledast_2$
Histo	f_{Histo}	++	+

Views	inp_view		out_view
	Bins	Elems	Out
Histo	(b,e) $\mapsto$ (b)	(b,e) $\mapsto$ (e)	(b,e) $\mapsto$ (b)

5) Histogram

md_hom	f	$\circledast_1$
map(f)	f	++
reduce($\oplus$)	id	$\oplus$
reduce($\oplus$,$\otimes$)	(x) $\mapsto$ (x,x)	($\oplus$,$\otimes$)

Views	inp_view	out_view	
	I	O_1	O_2
map(f)	(i) $\mapsto$ (i)	(i) $\mapsto$ (i)	/
reduce($\oplus$)	(i) $\mapsto$ (i)	(i) $\mapsto$ ()	/
reduce($\oplus$,$\otimes$)	(i) $\mapsto$ (i)	(i) $\mapsto$ ()	(i) $\mapsto$ ()

6) Map/Reduce Patterns

md_hom	f	$\circledast_1$
scan($\oplus$)	id	$++_{\text{prefix-sum}}(\oplus)$

Views	inp_view	out_view
	I	O
scan($\oplus$)	(i) $\mapsto$ (i)	(i) $\mapsto$ (i)

7) Scan Pattern

md_hom	f	$\circledast_1$	$\circledast_2$
MBBS	id	$++_{\text{prefix-sum}}(+)$	+

Views	inp_view	out_view
	A	Out
MBBS	(i,j) $\mapsto$ (i,j)	(i) $\mapsto$ (i)

8) Maximum Bottom Box Sum

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

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

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	–	–

NVIDIA

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	–	–

intel®

MDH achieves encouraging experimental results [1]

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

MDH in MLIR

WIP

`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)

Implemented by Lars & Jens Hunloh

(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]

We aim to contribute to MLIR!

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

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) )
```

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) )
```

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) )
```

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:

MDH

```
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:

Linalg

```
#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 expressing MatVec

Quick reminder "`Linalg`", before comparison "`Linalg` vs. `MDH`"

Quick Reminder: Linalg

Linalg MLIR Dialect:

Linalg

```
#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 expressing MatVec

Accesses to Input/Output Data

Quick Reminder: Linalg

Linalg MLIR Dialect:

Linalg

```
#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 expressing MatVec

Iteration Space Specification

Quick Reminder: Linalg

Linalg MLIR Dialect:

Linalg

```
#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 expressing MatVec

MatVec Computation (mul and add)

Comparison: Linalg vs MDH

Linalg

```
#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
  }
}
```

MDH

```
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
}
```

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 2nd dimension ($\circledast_2$)

$$\begin{pmatrix} M_{1,1} & \dots & M_{1,K} \\ \vdots & \ddots & \vdots \\ M_{I,1} & \dots & M_{I,K} \end{pmatrix}, \begin{pmatrix} v_1 \\ \vdots \\ v_K \end{pmatrix} \overset{\texttt{MatVec}}{\mapsto} \overset{\xrightarrow{\qquad\qquad\qquad\circledast_2}}{\begin{array}{ccc} f(M_{1,1}, v_1) & \dots & f(M_{1,K}, v_K) \\ \vdots & \ddots & \vdots \\ f(M_{I,1}, v_1) & \dots & f(M_{I,K}, v_K) \end{array}} \Big\downarrow \circledast_1 = \begin{pmatrix} M_{1,1} * v_1 + \ldots + M_{1,K} * v_K \\ \vdots \\ M_{I,1} * v_1 + \ldots + M_{I,K} * v_K \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_I \end{pmatrix}$$

`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

Linalg

```
// ...
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
  }
}
```

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?**

<u>Comparison:</u> Linalg vs MDH

<u>Summary — Questions to Linalg community:</u>

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?

<u>Further Questions:</u>

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

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

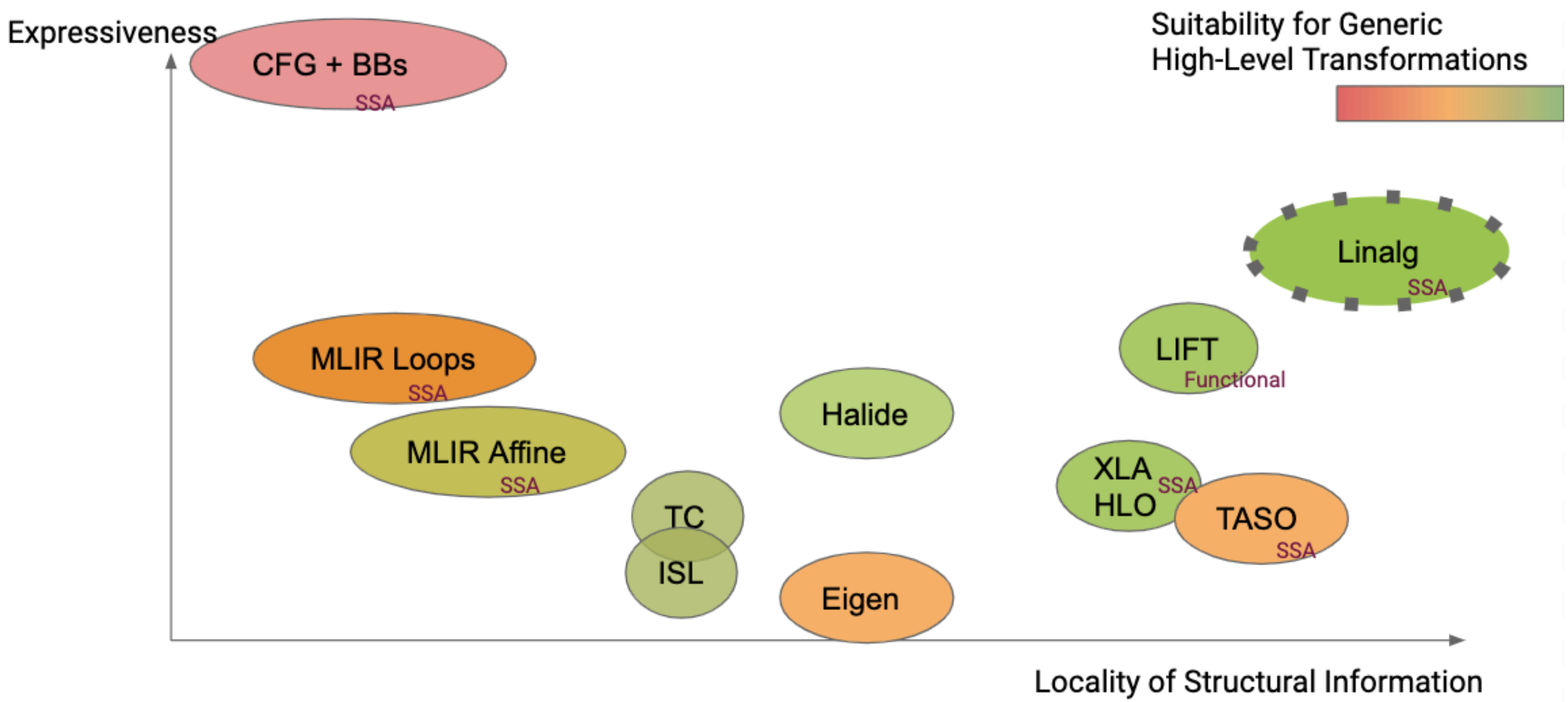
[1] "(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms" (*under review at ACM TOPLAS*)

<u>Comparison:</u> 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?

WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER


Questions?

Lars & Jens
Hunloh

Ari Rasch

We are grateful for any kind of feedback

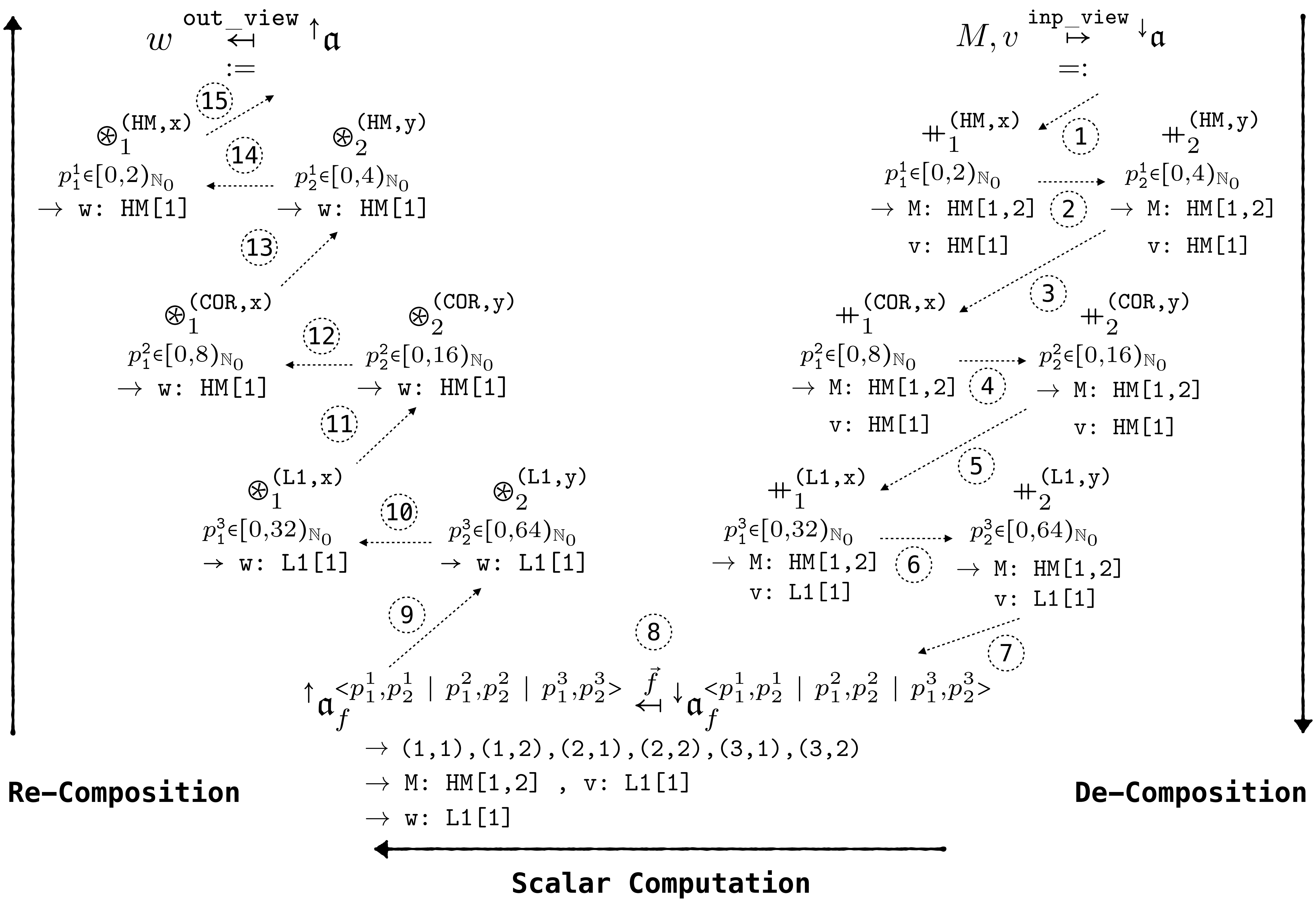
https://mdh-lang.org

https://atf-tuner.org

https://hca-project.org

<u>Excursion:</u> MDH Low-Level Representation

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



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

<u>Excursion:</u> MDH Low-Level Representation

<u>MDH Low-Level Representation in MLIR:</u>

```
#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", ...)
}
```

Input View

De-Composition Phase

Scalar Phase

Re-Composition Phase

Output View

MDH–LL–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
}
```

MDH-HL-MLIR

Lowering

HL → LL

using
Auto-Tuning [2]

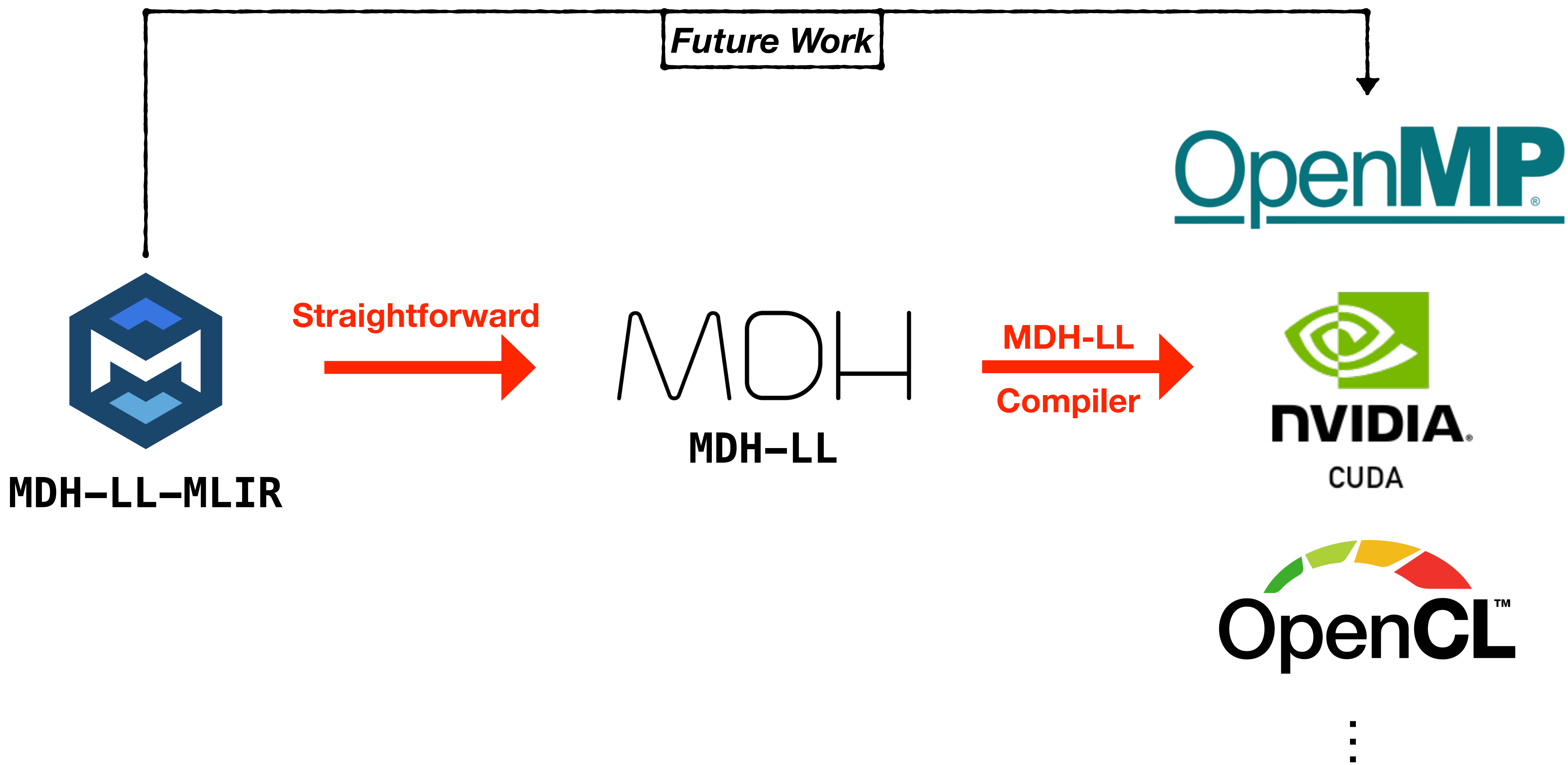
```
#map  = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0, d1) -> (d1)>
#map2 = affine_map<(d0, d1) -> (d0)>
module {
  func.func @main()
  {
    %alloc   = memref.alloc() : memref<128x64xf32>
    %alloc_0 = memref.alloc() : memref<64xf32>
    %0 = call @mdh_matvec(%alloc, %alloc_0) : (memref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

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

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, …)