Westfälische
Wilhelms-Universität
Münster

Multi-Dimensional Homomorphisms and Their Implementation in OpenCL

Ari Rasch and Sergei Gorlatch

University of Münster, Germany

Goals

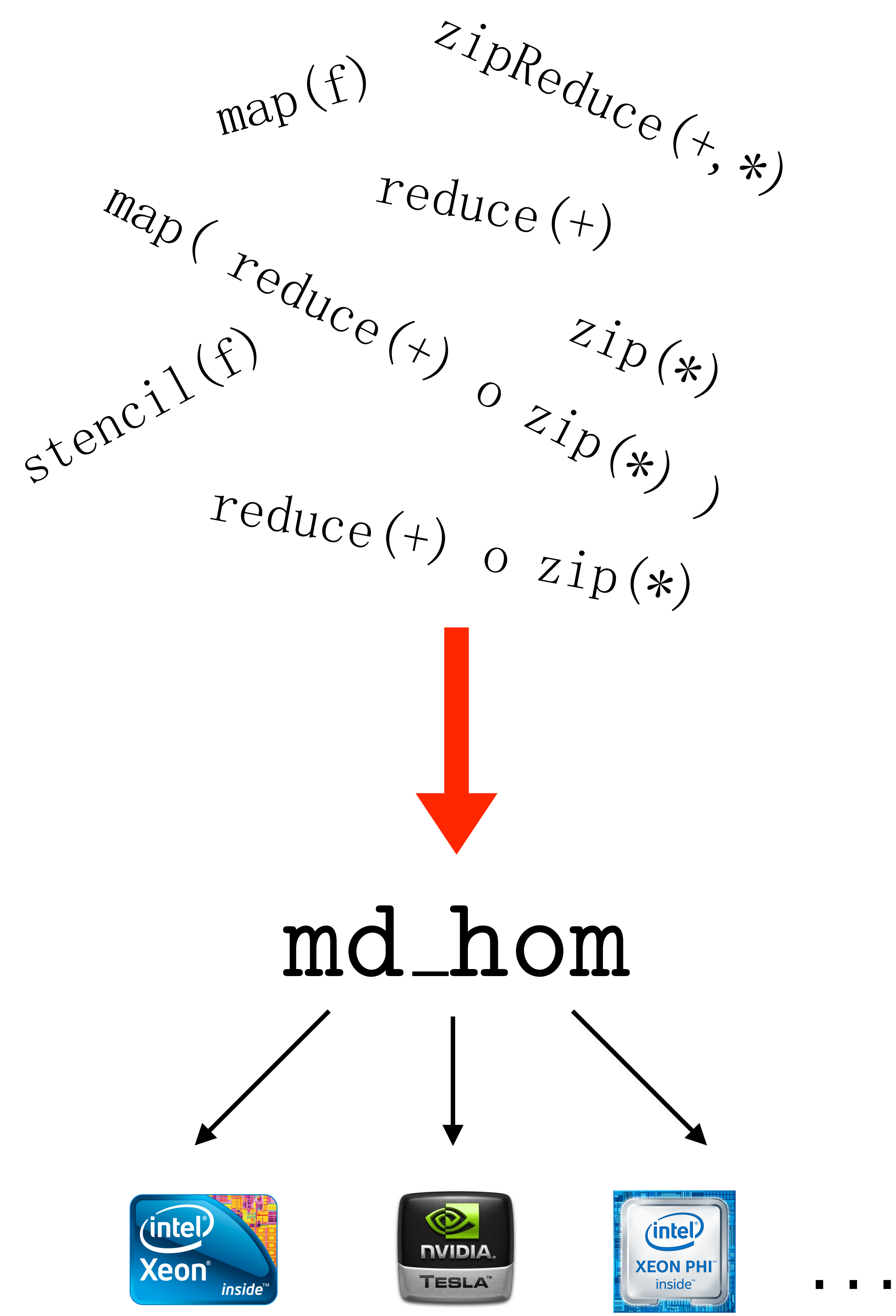
1. Definition of a data-parallel pattern (a.k.a. *algorithmic skeleton*) **`md_hom`** that

   - has high performance on different processor architectures and for different input sizes.

   - *is expressive*, i.e., covers typical parallel patterns and functions generated by composition and nesting of several patterns.

2.

3.



Goals

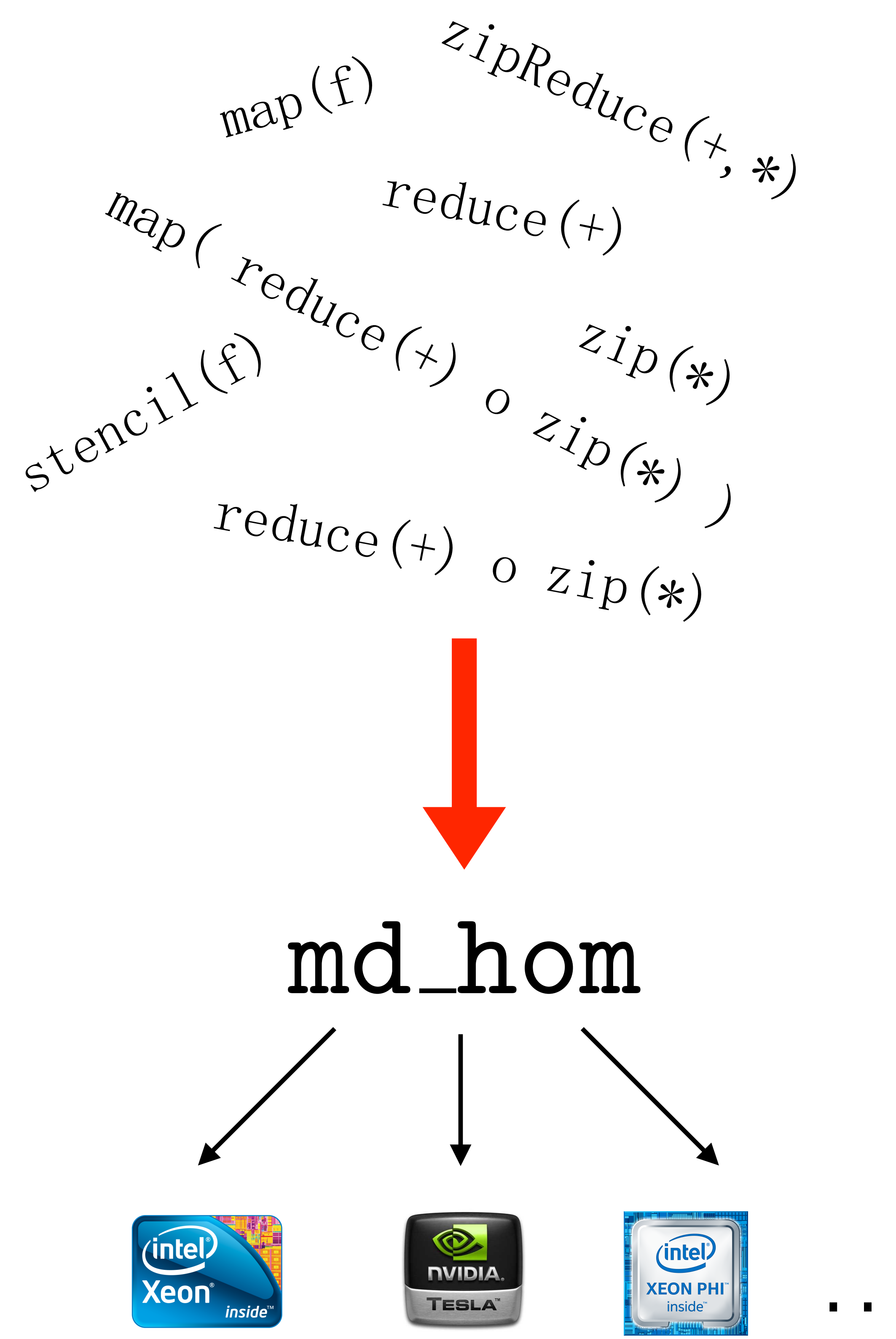
1. Definition of a data-parallel pattern (a.k.a. *algorithmic skeleton*) **`md_hom`** that

   - has high performance on different processor architectures and for different input sizes.

   - *is expressive*, i.e., covers typical parallel patterns and functions generated by composition and nesting of several patterns.

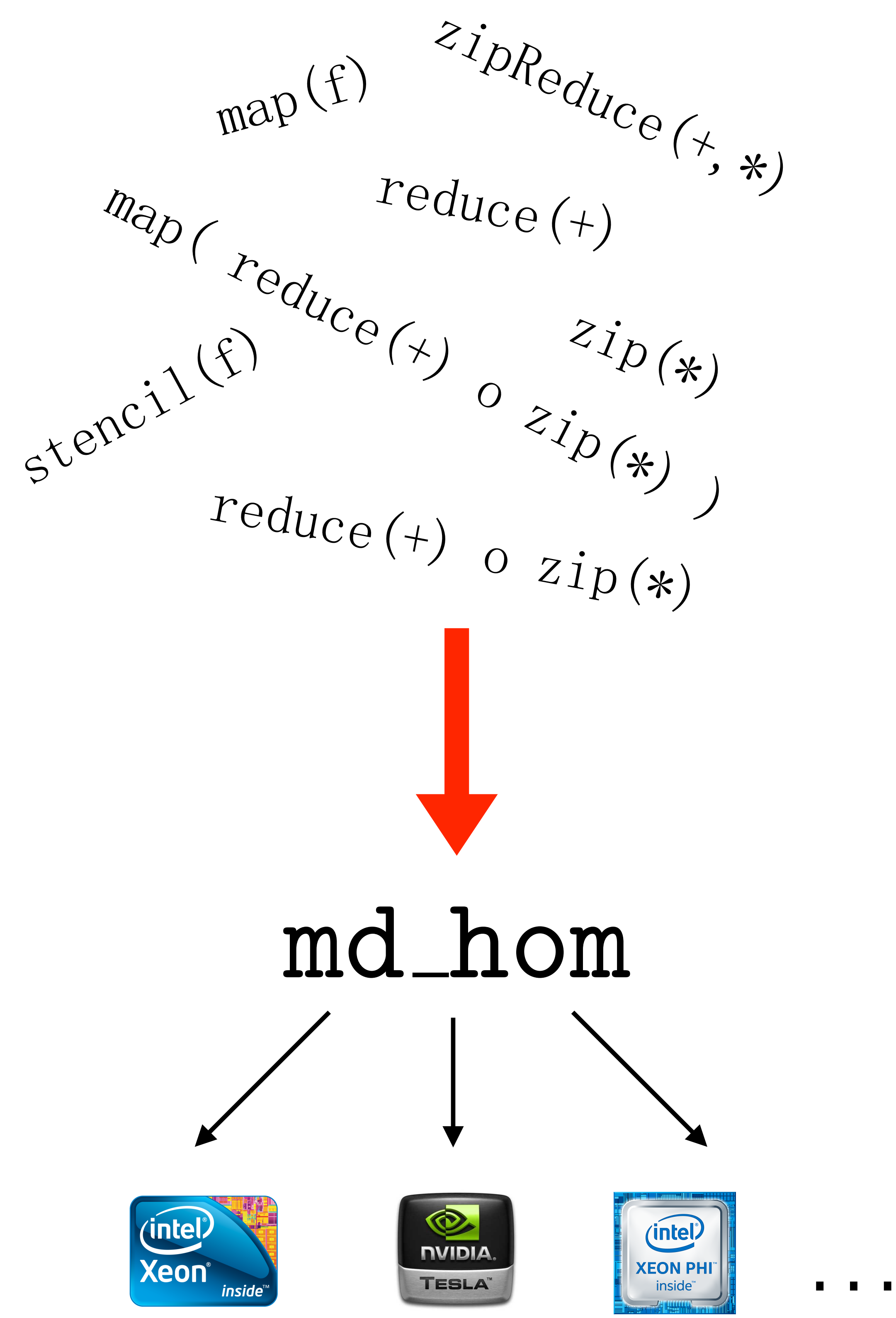
2. An extension of the theory of *list homomorphisms* (*LHs*) to *multi-dimensional homomorphisms* (*MDHs*) → **`md_hom`** is a pattern to conveniently express MDH functions.

3.



Goals

1. Definition of a data-parallel pattern (a.k.a. *algorithmic skeleton*) `md_hom` that
   - has high performance on different processor architectures and for different input sizes.
   - *is expressive*, i.e., covers typical parallel patterns and functions generated by composition and nesting of several patterns.
2. An extension of the theory of *list homomorphisms* (*LHs*) to *multi-dimensional homomorphisms* (*MDHs*) → `md_hom` is a pattern to conveniently express MDH functions.
3. An OpenCL implementation schema for `md_hom`.



Weakness of LHs

- Traditional definition of a LH:

A function h on lists is called a LH iff there exists a *combine operator* $\circledast$, such that h can be applied to parts of its input and the intermediate results be combined by using $\circledast$:

$$h(x \mathbin{+\!\!+} y) = h(x) \circledast h(y)$$

where $\mathbin{+\!\!+}$ denotes list concatenation.

Weakness of LHs

- Traditional definition of a LH:

A function h on lists is called a LH iff there exists a *combine operator* $\circledast$, such that h can be applied to parts of its input and the intermediate results be combined by using $\circledast$:

$$h(x +\!\!+\, y) = h(x) \circledast h(y)$$

where $+\!\!+$ denotes list concatenation.

- Weakness of LHs:
  - Parallelism is captured in only one dimension.
  - Functions on multi-dimensional data types (in general) allow to exploit parallelism in multiple dimensions.
  - Parallelizing such functions in only one dimension limits the degree of parallelism.

  → **Modern systems' hardware, which provides a high degree of parallelism, is not utilized at its full performance potential!**

Weakness of LHs

Example:

- *map(f)*: applies a function *f* to the elements of an matrix
- Two cases for map(f) as LH:
  1.
  2.

$$\underbrace{map(f)\left(\begin{pmatrix} m_{(1,1)} & \cdots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \cdots & m_{(m,n)} \end{pmatrix}\right) = \begin{matrix} map(f)(\,(m_{(1,1)} \;\ldots\; m_{(1,n)})\,) \\ +\!\!+ \\ \vdots \\ +\!\!+ \\ map(f)(\,(m_{(m,1)} \;\ldots\; m_{(m,n)})\,) \end{matrix}}_{\text{1-st case}}$$

$$\underbrace{map(f)\left(\begin{pmatrix} m_{(1,1)} & \cdots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \cdots & m_{(m,n)} \end{pmatrix}\right) = map(f)\left(\begin{pmatrix} m_{(1,1)} \\ \vdots \\ m_{(m,1)} \end{pmatrix}\right) +\!\!+ \; map(f)\left(\begin{pmatrix} m_{(1,n)} \\ \vdots \\ m_{(m,n)} \end{pmatrix}\right)}_{\text{2-nd case}}$$

Weakness of LHs

Example:

- *map(f)*: applies a function *f* to the elements of an matrix
- Two cases for map(f) as LH:
  1. the matrix is a list of row vectors
  2. the matrix is a list of column vectors
- <u>Drawback:</u> Degree of parallelism is restricted in both cases.

$$\underbrace{map(f)\left(\begin{pmatrix} m_{(1,1)} & \cdots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \cdots & m_{(m,n)}\end{pmatrix}\right) = \begin{matrix} map(f)(\,(m_{(1,1)} \ \ldots \ m_{(1,n)})\,) \\ +\!\!+ \\ \vdots \\ +\!\!+ \\ map(f)(\,(m_{(m,1)} \ \ldots \ m_{(m,n)})\,)\end{matrix}}_{\text{1-st case}}$$

$$\underbrace{map(f)\left(\begin{pmatrix} m_{(1,1)} & \cdots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \cdots & m_{(m,n)}\end{pmatrix}\right) = map(f)\left(\begin{pmatrix} m_{(1,1)} \\ \vdots \\ m_{(m,1)}\end{pmatrix}\right) +\!\!+ \; map(f)\left(\begin{pmatrix} m_{(1,n)} \\ \vdots \\ m_{(m,n)}\end{pmatrix}\right)}_{\text{2-nd case}}$$

$$\underbrace{map(f)\left(\begin{pmatrix} m_{(1,1)} & \cdots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \cdots & m_{(m,n)}\end{pmatrix}\right) = \begin{matrix} map(f)(\,(m_{(1,1)})\,) \ +\!\!+ \ \ldots \ +\!\!+ \ map(f)(\,(m_{(1,n)})\,) \\ +\!\!+ \\ \vdots \\ +\!\!+ \\ map(f)(\,(m_{(m,1)})\,) \ +\!\!+ \ \ldots \ +\!\!+ \ map(f)(\,(m_{(m,n)})\,)\end{matrix}}_{\textit{combined}}$$

MDHs

- *MDHs* extend LHs in order to capture parallelism in multiple dimensions.
- MDHs use *multi-dimensional arrays (MDAs)* as their input type.
- MDAs can be *concatenated* in different dimensions.

Remark: The following definitions and propositions are simplified for the presentation. More precise, formal definitions/propositions can be found in the paper.

Definition of MDA

Definition: [MDA]

Let T be an arbitrary set, $D \in \mathbb{N}$ and $(N_1, \ldots, N_D) \in \mathbb{N}^D$. A *$D$-dimensional T-array of size* $(N_1, \ldots, N_D)$ is a function that maps D-many indices to an element of T:

$$[1, N_1] \times \cdots \times [1, N_D] \to T$$

Examples:

$$\underbrace{(1, 2, 3)}_{\text{1-dim array of size (3)}}$$

$$\underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}}_{\text{2-dim array of size (2,3)}}$$

Definition of Concatenation

Definition: [MDA concatenation]

The *concatenation of two D-dimensional arrays in the i-th dimension*, whose sizes differ only in dimension i, is defined as binary function $+\!\!+_d$.

Example:

$$\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \end{pmatrix} \in T[2][3] \;+\!\!+_1\; \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \in T[4][3] \;=\; \begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} \in T[6][3]$$

Definition of MDH

Definition: [MDH]

A function h on D-dimensional arrays is called a *D-dimensional homomorphism* iff there exist *combine operators* $\circledast_1, \ldots, \circledast_D$ such that h can be applied to parts of its input and the intermediate results be combined in dimension i by using $\circledast_i$:

$$h(a +\!\!+_i\, b) \;=\; h(a) \circledast_i h(b)$$

Definition of MDH

Definition: [MDH]

A function h on D-dimensional arrays is called a *D-dimensional homomorphism* iff there exist *combine operators* $\circledast_1, \ldots, \circledast_D$ such that h can be applied to parts of its input and the intermediate results be combined in dimension i by using $\circledast_i$:

$$h(a \mathbin{+\!\!+}_i b) \;=\; h(a) \circledast_i h(b)$$

Example:

The function $map(f)$ is a 2-dimensional homomorphism with concatenation as combine operators: $\circledast_1 = \mathbin{+\!\!+}_1$ and $\circledast_2 = \mathbin{+\!\!+}_2$.

Both dimensions of parallelism are captured and not only one of them as for the traditional LH concept!

MDH Decomposition

- We aim to decompose MDHs in independent computations that can be carried out in parallel.
- For this, we first define MDA partitioning:

Let a be an MDA of dimension D. We refer to the MDA that is obtained by splitting a in the i-th dimension in P_i parts as $(P_1, \ldots, P_D)$-*partitioning of* a.

Example:

$$a = \underbrace{\begin{pmatrix} a_{(1,1)} & \cdots & a_{(1,n)} \\ \vdots & \ddots & \vdots \\ a_{(m,1)} & \cdots & a_{(m,n)} \end{pmatrix}}_{\in T[m][n]} \xrightarrow{\mathbf{P = (m,n)}} \begin{pmatrix} \underbrace{(a_{(1,1)})}_{T[1][1]} & \cdots & \underbrace{(a_{(1,n)})}_{T[1][1]} \\ \vdots & \ddots & \vdots \\ \underbrace{(a_{(m,1)})}_{T[1][1]} & \cdots & \underbrace{(a_{(m,n)})}_{T[1][1]} \end{pmatrix} \in T[m][n].[1][1]$$

The (m,n) partitioning of a is a 2-dim array of 2-dim arrays of size (m,n) and (1,1).

MDH Decomposition

Proposition:

Each D-dimensional homomorphism applied to an MDA a can be decomposed in $P_1 * \ldots * P_D$ independent computations by using the $(P_1, \ldots, P_D)$-partitioning of a.

Example:

$$\underbrace{h\left(\begin{pmatrix} a_{(1,1)} & \cdots & a_{(1,n)} \\ \vdots & \ddots & \vdots \\ a_{(m,1)} & \cdots & a_{(m,n)} \end{pmatrix} \right)}_{\in T[m][n]} \overset{\text{Prop.}}{=} \begin{array}{c} \circledast_2 \longrightarrow \\ \begin{pmatrix} h(\underbrace{(a_{(1,1)})}_{T[1][1]}) & \cdots & h(\underbrace{(a_{(1,n)})}_{T[1][1]}) \\ \vdots & \ddots & \vdots \\ h(\underbrace{(a_{(m,1)})}_{T[1][1]}) & \cdots & h(\underbrace{(a_{(m,n)})}_{T[1][1]}) \end{pmatrix} \end{array} \Big\downarrow \circledast_1$$

- h is independently applied to each singleton array
- Intermediate results are combined in dimension i by using $\circledast_i$

Definition `md_hom`

Proposition:

Every MDH is completely determined by its combine operators and its actions on singleton arrays.

Definition: [`md_hom`]

We write

$$\texttt{md\_hom}(\ f,\ (\circledast_1, \ldots, \circledast_D)\ )$$

for the unique D-dimensional homomorphism with combine operators $\circledast_1, \ldots, \circledast_D$ and action f on singleton arrays: $h(a) = f(a[0] \ldots [0])$.

Examples for `md_hom`

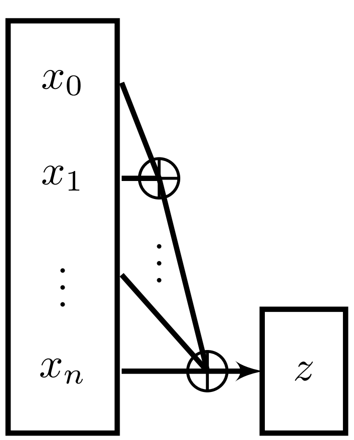
Examples:

Function	`md_hom` representation
`map`(`f`)	`md_hom`(`f` , $(+\!\!+_1, +\!\!+_2)$)

Examples for `md_hom`

Examples:

Function	`md_hom` representation
`map`(`f`)	`md_hom`(`f` , $(+\!\!+_1, +\!\!+_2)$)
`reduce`($\oplus$)	`md_hom`(`id` , $(\oplus)$)



reduce

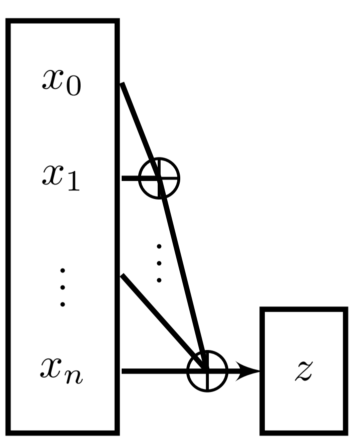
Examples for `md_hom`

Examples:

Function	`md_hom` representation
`map`$(\mathtt{f})$	`md_hom`$(\ \mathtt{f}\ ,\ (+\!\!+_1, +\!\!+_2)\ )$
`reduce`$(\oplus)$	`md_hom`$(\ \mathtt{id}\ ,\ (\oplus)\ )$
`dot_product`	`md_hom`$(\ *\ ,\ (+)\ )\ \circ$ `pair`

$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}, \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \overset{\mathtt{pair}}{\mapsto} \begin{pmatrix} (v_1, w_1) \\ \vdots \\ (v_n, w_n) \end{pmatrix}$$

Fusing two vectors by pairing their components.



reduce

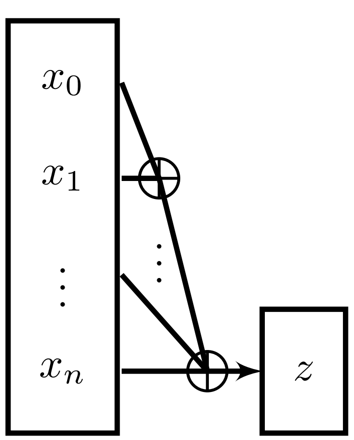
Examples for `md_hom`

Examples:

Function	`md_hom` representation
`map`(`f`)	`md_hom`(`f` , $(\mathbin{+\!\!+}_1, \mathbin{+\!\!+}_2)$)
`reduce`($\oplus$)	`md_hom`(`id` , $(\oplus)$)
`dot_product`	`md_hom`($*$, $(+)$) $\circ$ `pair`
`gemv`	`md_hom`($*$, $(\mathbin{+\!\!+}_1, +_{vec})$) $\circ$ `allPairs`

$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}, \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} \overset{\texttt{pair}}{\mapsto} \begin{pmatrix} (v_1, w_1) \\ \vdots \\ (v_n, w_n) \end{pmatrix}$$

Fusing two vectors by pairing their components.



reduce

$$\begin{pmatrix} m_{(1,1)} & \dots & m_{(1,n)} \\ \vdots & \ddots & \vdots \\ m_{(m,1)} & \dots & m_{(m,n)} \end{pmatrix}, \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \overset{\texttt{allPairs}}{\mapsto} \begin{pmatrix} (m_{(1,1)}, v_1) & \dots & (m_{(1,n)}, v_n) \\ \vdots & \ddots & \vdots \\ (m_{(m,1)}, v_1) & \dots & (m_{(m,n)}, v_n) \end{pmatrix}$$

Fusing a matrix and a vector by applying *pair* to the vector and each row

Implementing MDHs in OpenCL

- In OpenCL, the number of work-groups (WGs) and work-items (WIs) must be chosen by the programmer.
- Both have a great impact on performance:
  - a too low number causes that the hardware is not utilized optimally,
  - a too high number may cause high overhead.
- An appropriate number of WGs and WIs must be chosen specifically for the hardware and input size.

→ Our implementation schema will be generic in these parameters in order to provide high performance for arbitrary hardware and input sizes.

Implementing MDHs in OpenCL

- We illustrate our $\texttt{md\_hom}$ implementation schema by using the example of GEMV:

$$\texttt{md\_hom}(\; * \;,\; (+\!\!+_1, +)\;) \circ \texttt{allPairs}$$

- For this, we

  1. demonstrate our decomposition schema,

  2. present the corresponding OpenCL implementation.

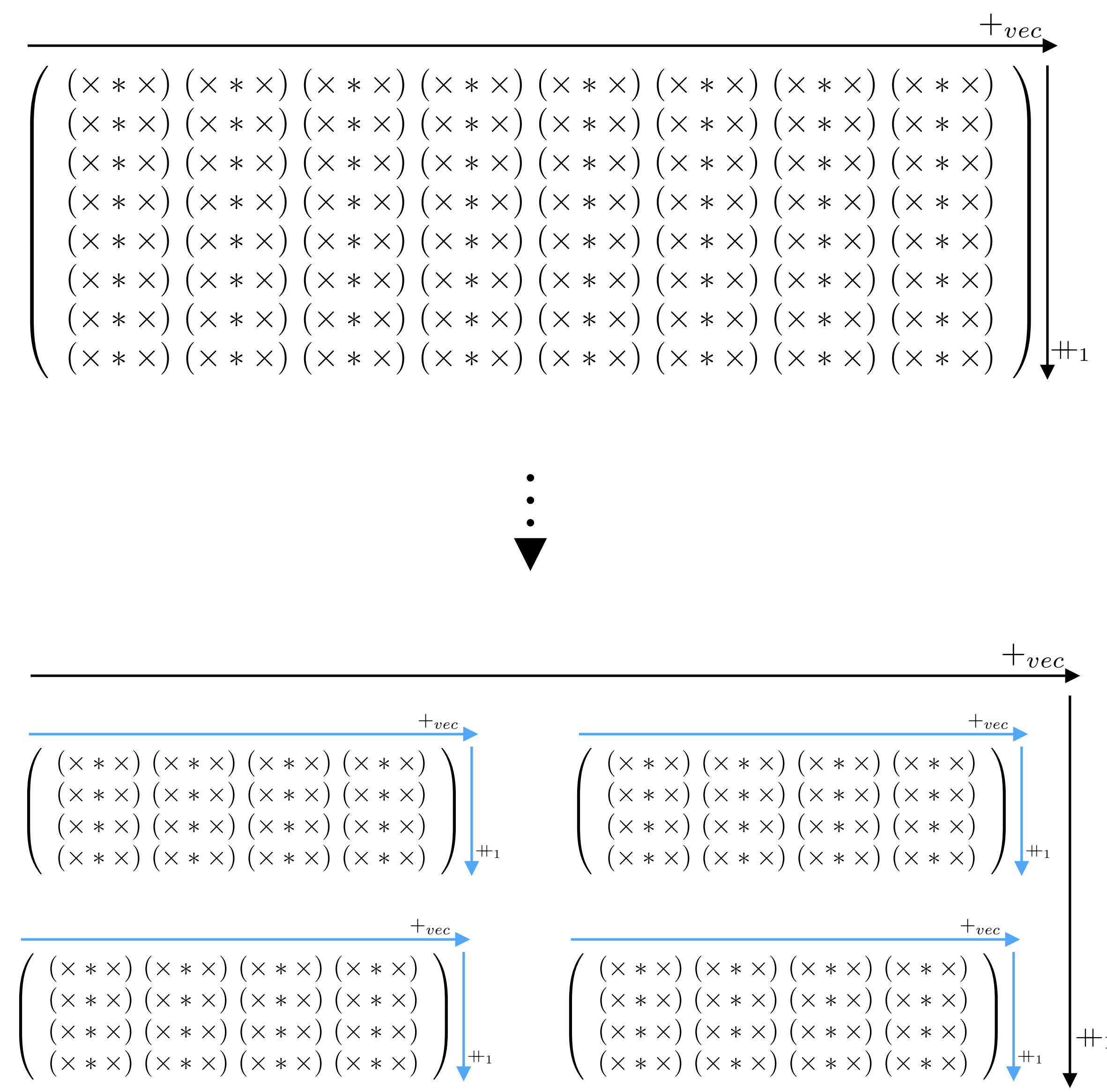
Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:

$$\begin{array}{cl} \xrightarrow{\hspace{20em} +_{vec}} & \\ \begin{pmatrix} (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \\ (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) & (\times * \times) \end{pmatrix} & \Big\downarrow +\!\!+_1 \end{array}$$

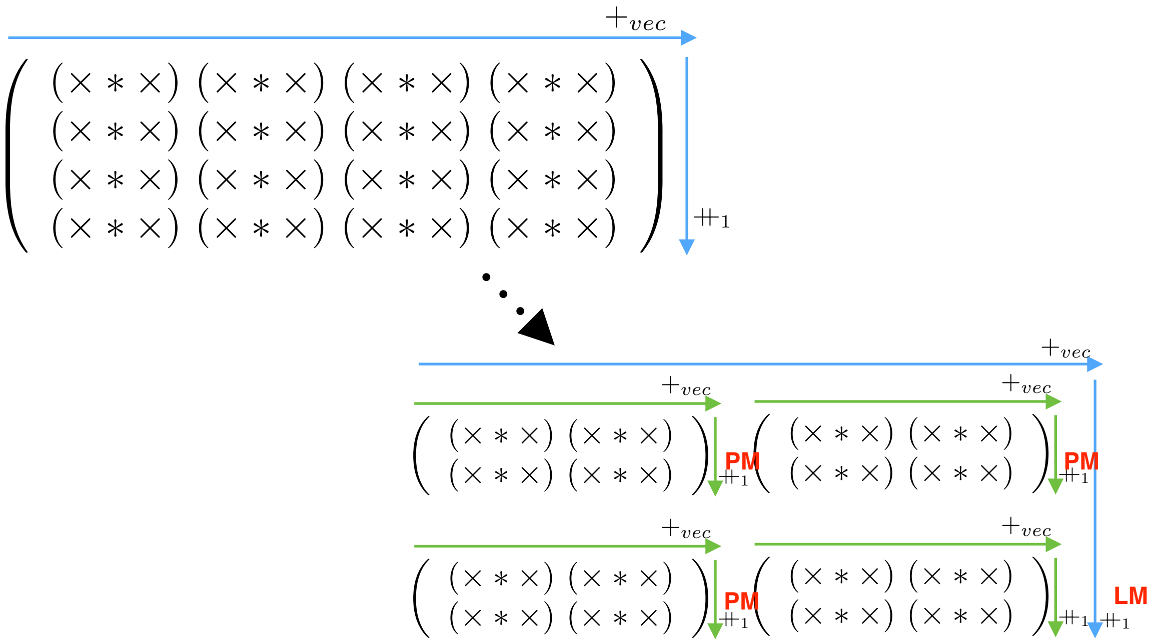
Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:
  - **P_{wg} = (NUM_WG_Y, NUM_WG_X) → handled by the WGs**



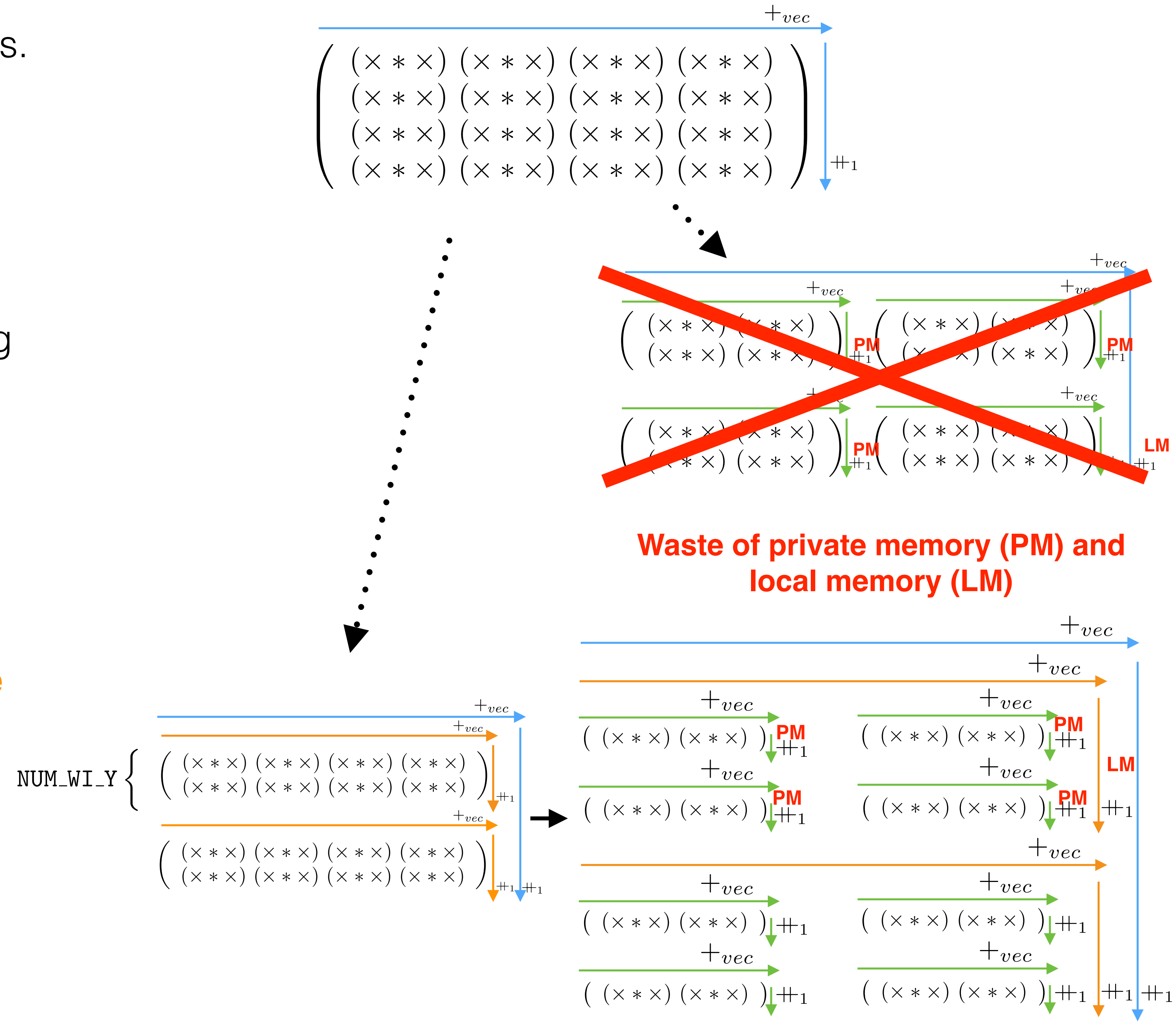
Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:
    - **P_{wg} = (NUM_WG_Y, NUM_WG_X) → handled by the WGs**
    -
    - **P_{wi} = (NUM_WI_Y, NUM_WI_X) → handled by the WIs**



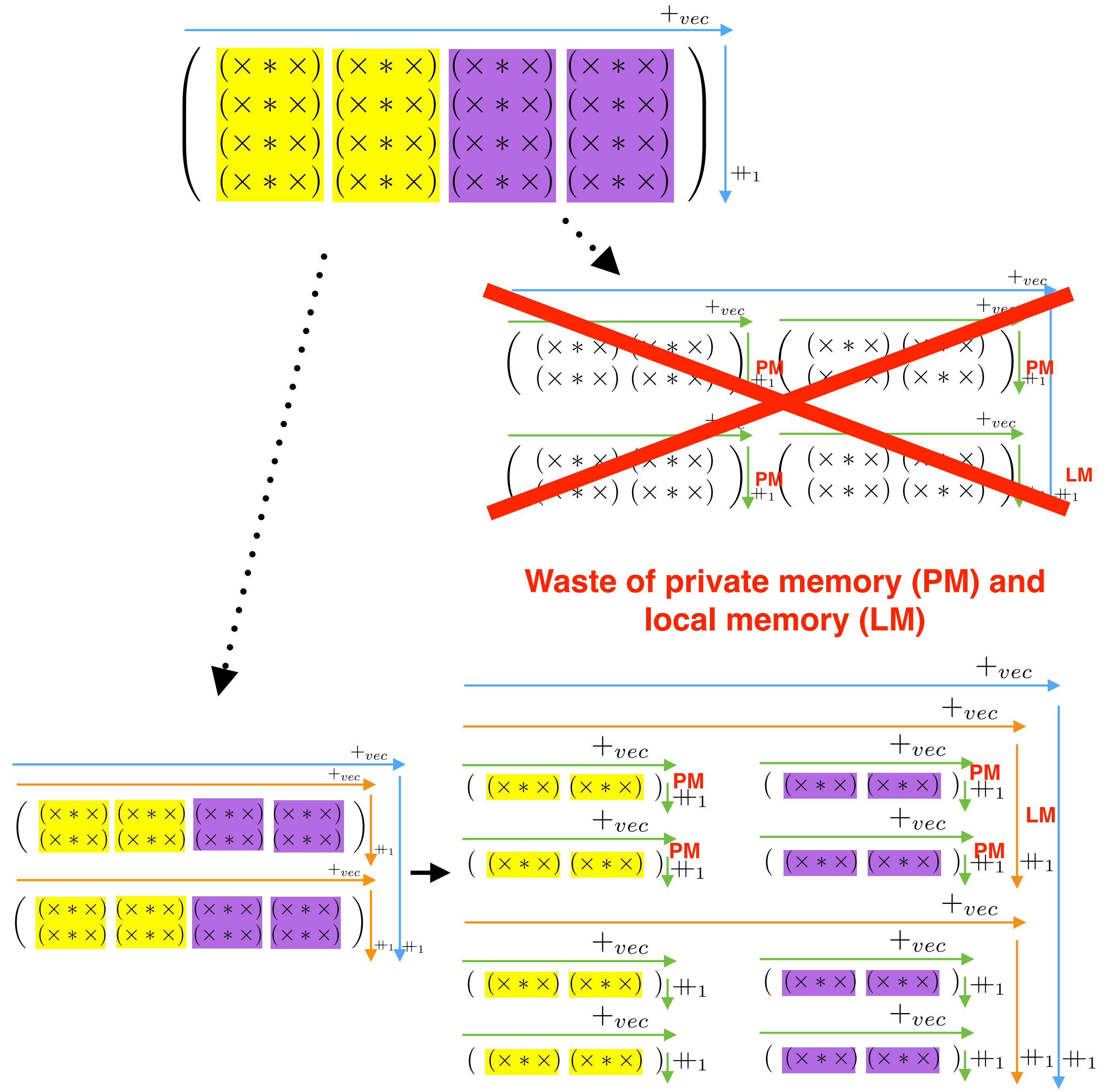
Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:
  - **P_{wg} = (NUM_WG_Y, NUM_WG_X) → handled by the WGs**
  - **P_{sq} = ($\frac{\texttt{M}}{\texttt{NUM\_WG\_Y*NUM\_WI\_Y}}$,1) → WIs iterate sequentially over these blocks**
  - **P_{wi} = (NUM_WI_Y, NUM_WI_X) → handled by the WIs**



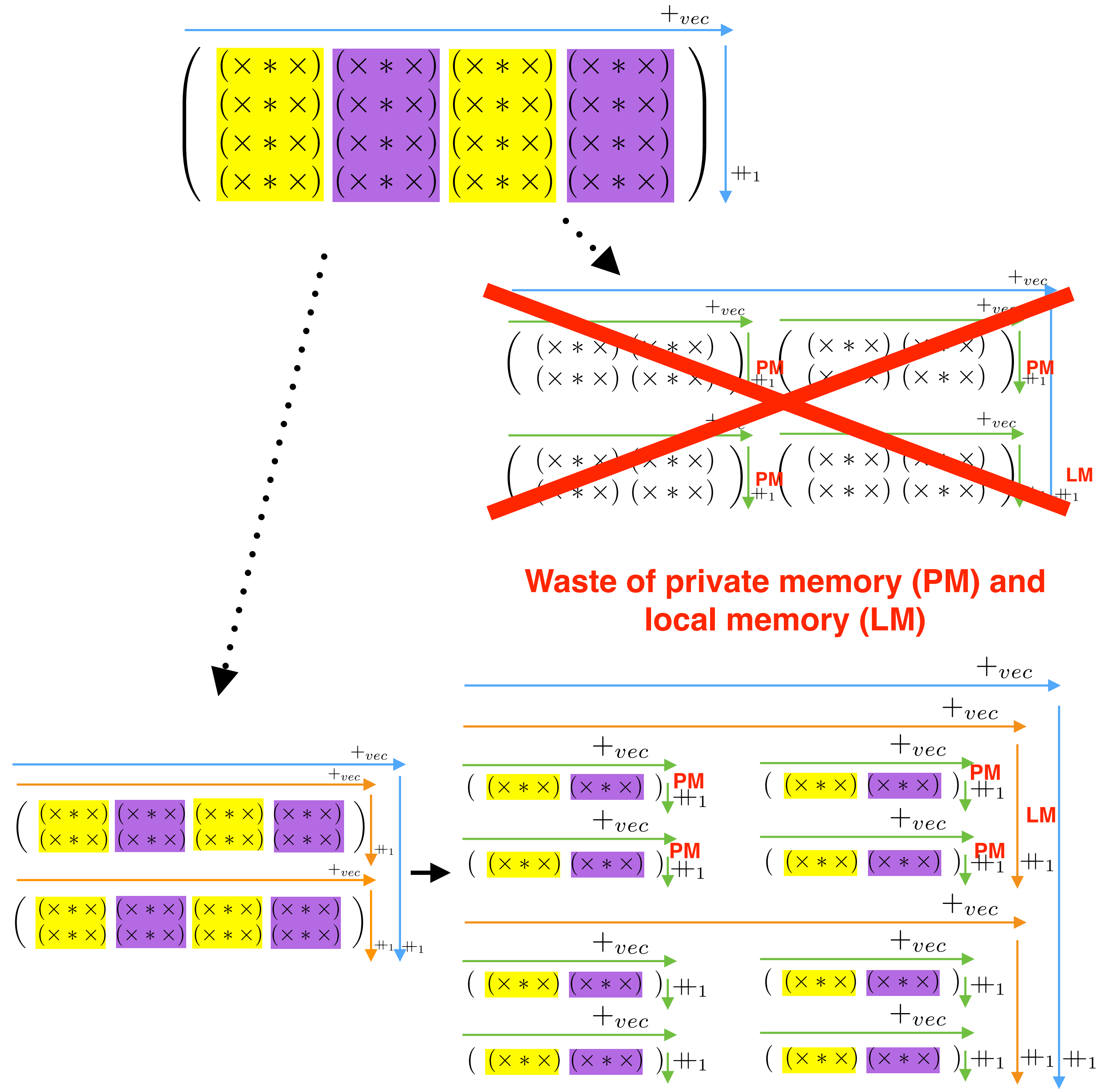
Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:
  - **P_{wg} = (NUM_WG_Y, NUM_WG_X) → handled by the WGs**
  - **P_{sq} = ($\frac{\texttt{M}}{\texttt{NUM\_WG\_Y*NUM\_WI\_Y}}$,1) → WIs iterate sequentially over these blocks**
  - **P_{wi} = (NUM_WI_Y, NUM_WI_X) → handled by the WIs**
- Reordering the P_{wg} partitions enable optimized memory accesses.



Decomposition Schema

- We arrange WGs and WIs in D=2 dimensions.
- For demonstration, we use an (M,N) = (8,8) input MDA and start 2x2 WGs where each comprises 2x2 WIs.
- We decompose the computations according to the following partitionings to distribute them evenly to the WGs and WIs:
  - **P_{wg} = (NUM_WG_Y, NUM_WG_X) → handled by the WGs**
  - **P_{sq} = ($\frac{\texttt{M}}{\texttt{NUM\_WG\_Y*NUM\_WI\_Y}}$,1) → WIs iterate sequentially over these blocks**
  - **P_{wi} = (NUM_WI_Y, NUM_WI_X) → handled by the WIs**
- Reordering the P_{wg} partitions enable optimized memory accesses.



OpenCL Implementation according to the Decomposition Schema

- Synchronization between WGs requires the start of two kernels in OpenCL.
- The P_{wg} partitions are computed in the first kernel.
- The obtained results (one per WG) are combined in the second kernel.
- We focus on the first kernel since the second kernel is analogous (but performed in only one WG in the second dimension in order to avoid further synchronization).

Implementation of `allPairs` and the Partitionings

- We implement the *allPairs* function and the partitionings in code by using preprocessor macros:

```
// view macro
#define allPairs( i, j, c ) ( ( (c) == 0 ) ? in_matrix[ (i-1) * N + (j-1) ] : in_vector[ (j-1) ] )

// reordering
#define reorder(j) (  ( ((j)-1) / WI_PART_SIZE_2 ) + ( ((j)-1) % WI_PART_SIZE_2 ) * NUM_WI_2 + 1  )

// P_wg partitioning
#define my_p_wg( i, j, c ) view( WG_OFFSET_1 + (i) , WG_OFFSET_2 + reorder(j) , (c) )

// P_sq partitioning
#define my_p_sq( i, j, c ) my_p_wg( SQ_OFFSET_1(i_sq) + (i) , SQ_OFFSET_2(j_sq) + (j) , (c) )

// P_wi partitioning
#define my_p_wi( i, j, c ) my_p_sq( WI_OFFSET_1 + (i) , WI_OFFSET_2 + (j) , (c) )

// results
#define my_res( i ) out_vector[ ( WG_OFFSET_1 + WI_OFFSET_1 + (i-1) * SQ_PART_SIZE_1 ) * NUM_WG_2 + WG_ID_2 ]
```

- The allPairs macro takes three integers, i,j and c, and returns either the first component (c==0) or the second component (c==1) of the input MDH's element at the position i,j.

- reorder takes an integer and returns the integer that represents the new position of the calling WI.

- The next three macros represent a WI's P_{wg}, P_{sq} and P_{wi} partition.

- *my_res* indicates where a WI has to store its result.

Implementation of `allPairs` and the Partitionings

- We implement the *allPairs* function and the partitionings in code by using preprocessor macros:

```
// view macro
#define allPairs( i, j, c ) ( ( (c) == 0 ) ? in_matrix[ (i-1) * N + (j-1) ] : in_vector[ (j-1) ] )

// reordering
#define reorder(j) (  ( ((j)-1) / WI_PART_SIZE_2 ) + ( ((j)-1) % WI_PART_SIZE_2 ) * NUM_WI_2 + 1  )

// P_wg partitioning
#define my_p_wg( i, j, c ) view( WG_OFFSET_1 + (i) , WG_OFFSET_2 + reorder(j) , (c) )

// P_sq partitioning
#define my_p_sq( i, j, c ) my_p_wg( SQ_OFFSET_1(i_sq) + (i) , SQ_OFFSET_2(j_sq) + (j) , (c) )

// P_wi partitioning
#define my_p_wi( i, j, c ) my_p_sq( WI_OFFSET_1 + (i) , WI_OFFSET_2 + (j) , (c) )

// results
#define my_res( i ) out_vector[ ( WG_OFFSET_1 + WI_OFFSET_1 + (i-1) * SQ_PART_SIZE_1 ) * NUM_WG_2 + WG_ID_2 ]
```

- The allPairs macro takes three integers, i,j and c, and returns either the first component (c==0) or the second component (c==1) of the input MDH's element at the position i,j.

- reorder takes an integer and returns the integer that represents the new position of the calling WI.

- The next three macros represent a WI's P_{wg}, P_{sq} and P_{wi} partition.

- *my_res* indicates where a WI has to store its result.

Implementation of `allPairs` and the Partitionings

- We implement the *allPairs* function and the partitionings in code by using preprocessor macros:

```
// view macro
#define allPairs( i, j, c ) ( ( (c) == 0 ) ? in_matrix[ (i-1) * N + (j-1) ] : in_vector[ (j-1) ] )

// reordering
#define reorder(j) (  ( ((j)-1) / WI_PART_SIZE_2 ) + ( ((j)-1) % WI_PART_SIZE_2 ) * NUM_WI_2 + 1  )

// P_wg partitioning
#define my_p_wg( i, j, c ) view( WG_OFFSET_1 + (i) , WG_OFFSET_2 + reorder(j) , (c) )

// P_sq partitioning
#define my_p_sq( i, j, c ) my_p_wg( SQ_OFFSET_1(i_sq) + (i) , SQ_OFFSET_2(j_sq) + (j) , (c) )

// P_wi partitioning
#define my_p_wi( i, j, c ) my_p_sq( WI_OFFSET_1 + (i) , WI_OFFSET_2 + (j) , (c) )

// results
#define my_res( i ) out_vector[ ( WG_OFFSET_1 + WI_OFFSET_1 + (i-1) * SQ_PART_SIZE_1 ) * NUM_WG_2 + WG_ID_2 ]
```

- The allPairs macro takes three integers, i,j and c, and returns either the first component (c==0) or the second component (c==1) of the input MDH's element at the position i,j.

- reorder takes an integer and returns the integer that represents the new position of the calling WI.

- The next three macros represent a WI's P_{wg}, P_{sq} and P_{wi} partition.

- *my_res* indicates where a WI has to store its result.

Implementation of `allPairs` and the Partitionings

- We implement the *allPairs* function and the partitionings in code by using preprocessor macros:

```
// view macro
#define allPairs( i, j, c ) ( ( (c) == 0 ) ? in_matrix[ (i-1) * N + (j-1) ] : in_vector[ (j-1) ] )

// reordering
#define reorder(j) (  ( ((j)-1) / WI_PART_SIZE_2 ) + ( ((j)-1) % WI_PART_SIZE_2 ) * NUM_WI_2 + 1  )

// P_wg partitioning
#define my_p_wg( i, j, c ) view( WG_OFFSET_1 + (i) , WG_OFFSET_2 + reorder(j) , (c) )

// P_sq partitioning
#define my_p_sq( i, j, c ) my_p_wg( SQ_OFFSET_1(i_sq) + (i) , SQ_OFFSET_2(j_sq) + (j) , (c) )

// P_wi partitioning
#define my_p_wi( i, j, c ) my_p_sq( WI_OFFSET_1 + (i) , WI_OFFSET_2 + (j) , (c) )

// results
#define my_res( i ) out_vector[ ( WG_OFFSET_1 + WI_OFFSET_1 + (i-1) * SQ_PART_SIZE_1 ) * NUM_WG_2 + WG_ID_2 ]
```

- The allPairs macro takes three integers, i,j and c, and returns either the first component (c==0) or the second component (c==1) of the input MDH's element at the position i,j.

- reorder takes an integer and returns the integer that represents the new position of the calling WI.

- The next three macros represent a WI's P_{wg}, P_{sq} and P_{wi} partition.

- *my_res* indicates where a WI has to store its result.

Implementation of `allPairs` and the Partitionings

- We implement the *allPairs* function and the partitionings in code by using preprocessor macros:

```
// view macro
#define allPairs( i, j, c ) ( ( (c) == 0 ) ? in_matrix[ (i-1) * N + (j-1) ] : in_vector[ (j-1) ] )

// reordering
#define reorder(j) (  ( ((j)-1) / WI_PART_SIZE_2 ) + ( ((j)-1) % WI_PART_SIZE_2 ) * NUM_WI_2 + 1  )

// P_wg partitioning
#define my_p_wg( i, j, c ) view( WG_OFFSET_1 + (i) , WG_OFFSET_2 + reorder(j) , (c) )

// P_sq partitioning
#define my_p_sq( i, j, c ) my_p_wg( SQ_OFFSET_1(i_sq) + (i) , SQ_OFFSET_2(j_sq) + (j) , (c) )

// P_wi partitioning
#define my_p_wi( i, j, c ) my_p_sq( WI_OFFSET_1 + (i) , WI_OFFSET_2 + (j) , (c) )

// results
#define my_res( i ) out_vector[ ( WG_OFFSET_1 + WI_OFFSET_1 + (i-1) * SQ_PART_SIZE_1 ) * NUM_WG_2 + WG_ID_2 ]
```

- The allPairs macro takes three integers, i,j and c, and returns either the first component (c==0) or the second component (c==1) of the input MDH's element at the position i,j.

- reorder takes an integer and returns the integer that represents the new position of the calling WI.

- The next three macros represent a WI's P_{wg}, P_{sq} and P_{wi} partition.

- *my_res* indicates where a WI has to store its result.

OpenCL Sourcecode

- Iteration over the P_{sq} partitions.
- Sequential computations on a P_{wi} partition in private memory.
- Summing up the WIs' results in local memory.
- Storing the WG's result in global memory.

```
__kernel void gemv_fst( __global float* in_matrix,
                        __global float* in_vector,
                        __global float* out_vector,
{

  // private memory for a WI's computation
  __private float res_prv = 0.0f;

  // local memory for a WG's computation
  __local   float res_lcl[ NUM_WI_1 ][ NUM_WI_2 ];

  // iteration over P_sq blocks
  for( int i_sq = 1 ; i_sq <= NUM_SQ_1 ; ++i_sq ) {
    for( int j_sq = 1 ; j_sq <= NUM_SQ_2 ; ++j_sq ) {
      res_prv = 0.0f;

      // sequential computation on a P_wi partition
      for( int i = 1 ; i <= WI_PART_SIZE_1 ; ++i )
        for( int j = 1 ; j <= WI_PART_SIZE_2 ; ++j )
        res_prv += my_p_wi( i, j, 0 ) * my_p_wi( i, j, 1 );

      // store result in local memory
      res_lcl[ WI_ID_1 ][ WI_ID_2 ] = res_prv;

      barrier( CLK_LOCAL_MEM_FENCE );

      // combine the WIs' results in dimension x
      for( int stride = NUM_WI_2 / 2 ; stride > 0 ; stride /= 2)
      {
        if( WI_ID_2 < stride)
          res_lcl[ WI_ID_1 ][ WI_ID_2 ] += res_lcl[ WI_ID_1 ][ WI_ID_2 + stride ];

        barrier( CLK_LOCAL_MEM_FENCE );
      }

      // store WGs' results in global memory
      if( WI_ID_2 == 0 )
        my_res( i_sq ) = res_lcl[ WI_ID_1 ][0];

      barrier( CLK_LOCAL_MEM_FENCE );

    } // end of for-loop j_sq
  } // end of for-loop i_sq
} // end of kernel
```

OpenCL Sourcecode

- Iteration over the P_{sq} partitions.
- Sequential computations on a P_{wi} partition in private memory.
- Summing up the WIs' results in local memory.
- Storing the WG's result in global memory.

```
__kernel void gemv_fst( __global float* in_matrix,
                        __global float* in_vector,
                        __global float* out_vector,
{

  // private memory for a WI's computation
  __private float res_prv = 0.0f;

  // local memory for a WG's computation
  __local   float res_lcl[ NUM_WI_1 ][ NUM_WI_2 ];

  // iteration over P_sq blocks
  for( int i_sq = 1 ; i_sq <= NUM_SQ_1 ; ++i_sq ) {
    for( int j_sq = 1 ; j_sq <= NUM_SQ_2 ; ++j_sq ) {
      res_prv = 0.0f;

      // sequential computation on a P_wi partition
      for( int i = 1 ; i <= WI_PART_SIZE_1 ; ++i )
        for( int j = 1 ; j <= WI_PART_SIZE_2 ; ++j )
        res_prv += my_p_wi( i, j, 0 ) * my_p_wi( i, j, 1 );

      // store result in local memory
      res_lcl[ WI_ID_1 ][ WI_ID_2 ] = res_prv;

      barrier( CLK_LOCAL_MEM_FENCE );

      // combine the WIs' results in dimension x
      for( int stride = NUM_WI_2 / 2 ; stride > 0 ; stride /= 2)
      {
        if( WI_ID_2 < stride)
          res_lcl[ WI_ID_1 ][ WI_ID_2 ] += res_lcl[ WI_ID_1 ][ WI_ID_2 + stride ];

        barrier( CLK_LOCAL_MEM_FENCE );
      }

      // store WGs' results in global memory
      if( WI_ID_2 == 0 )
        my_res( i_sq ) = res_lcl[ WI_ID_1 ][0];

      barrier( CLK_LOCAL_MEM_FENCE );

    } // end of for-loop j_sq
  } // end of for-loop i_sq
} // end of kernel
```

OpenCL Sourcecode

- Iteration over the P_{sq} partitions.
- Sequential computations on a P_{wi} partition in private memory.
- Summing up the WIs' results in local memory.
- Storing the WG's result in global memory.

```
__kernel void gemv_fst( __global float* in_matrix,
                        __global float* in_vector,
                        __global float* out_vector,
{

  // private memory for a WI's computation
  __private float res_prv = 0.0f;

  // local memory for a WG's computation
  __local   float res_lcl[ NUM_WI_1 ][ NUM_WI_2 ];

  // iteration over P_sq blocks
  for( int i_sq = 1 ; i_sq <= NUM_SQ_1 ; ++i_sq ) {
    for( int j_sq = 1 ; j_sq <= NUM_SQ_2 ; ++j_sq ) {
      res_prv = 0.0f;

      // sequential computation on a P_wi partition
      for( int i = 1 ; i <= WI_PART_SIZE_1 ; ++i )
        for( int j = 1 ; j <= WI_PART_SIZE_2 ; ++j )
        res_prv += my_p_wi( i, j, 0 ) * my_p_wi( i, j, 1 );

      // store result in local memory
      res_lcl[ WI_ID_1 ][ WI_ID_2 ] = res_prv;

      barrier( CLK_LOCAL_MEM_FENCE );

      // combine the WIs' results in dimension x
      for( int stride = NUM_WI_2 / 2 ; stride > 0 ; stride /= 2)
      {
        if( WI_ID_2 < stride)
          res_lcl[ WI_ID_1 ][ WI_ID_2 ] += res_lcl[ WI_ID_1 ][ WI_ID_2 + stride ];

        barrier( CLK_LOCAL_MEM_FENCE );
      }

      // store WGs' results in global memory
      if( WI_ID_2 == 0 )
        my_res( i_sq ) = res_lcl[ WI_ID_1 ][0];

      barrier( CLK_LOCAL_MEM_FENCE );

    } // end of for-loop j_sq
  } // end of for-loop i_sq
} // end of kernel
```

OpenCL Sourcecode

- Iteration over the P_{sq} partitions.
- Sequential computations on a P_{wi} partition in private memory.
- Summing up the WIs' results in local memory.
- Storing the WG's result in global memory.

```
__kernel void gemv_fst( __global float* in_matrix,
                        __global float* in_vector,
                        __global float* out_vector,
{

  // private memory for a WI's computation
  __private float res_prv = 0.0f;

  // local memory for a WG's computation
  __local   float res_lcl[ NUM_WI_1 ][ NUM_WI_2 ];

  // iteration over P_sq blocks
  for( int i_sq = 1 ; i_sq <= NUM_SQ_1 ; ++i_sq ) {
    for( int j_sq = 1 ; j_sq <= NUM_SQ_2 ; ++j_sq ) {
      res_prv = 0.0f;

      // sequential computation on a P_wi partition
      for( int i = 1 ; i <= WI_PART_SIZE_1 ; ++i )
        for( int j = 1 ; j <= WI_PART_SIZE_2 ; ++j )
        res_prv += my_p_wi( i, j, 0 ) * my_p_wi( i, j, 1 );

      // store result in local memory
      res_lcl[ WI_ID_1 ][ WI_ID_2 ] = res_prv;

      barrier( CLK_LOCAL_MEM_FENCE );

      // combine the WIs' results in dimension x
      for( int stride = NUM_WI_2 / 2 ; stride > 0 ; stride /= 2)
      {
        if( WI_ID_2 < stride)
          res_lcl[ WI_ID_1 ][ WI_ID_2 ] += res_lcl[ WI_ID_1 ][ WI_ID_2 + stride ];

        barrier( CLK_LOCAL_MEM_FENCE );
      }

      // store WGs' results in global memory
      if( WI_ID_2 == 0 )
        my_res( i_sq ) = res_lcl[ WI_ID_1 ][0];

      barrier( CLK_LOCAL_MEM_FENCE );

    } // end of for-loop j_sq
  } // end of for-loop i_sq
} // end of kernel
```

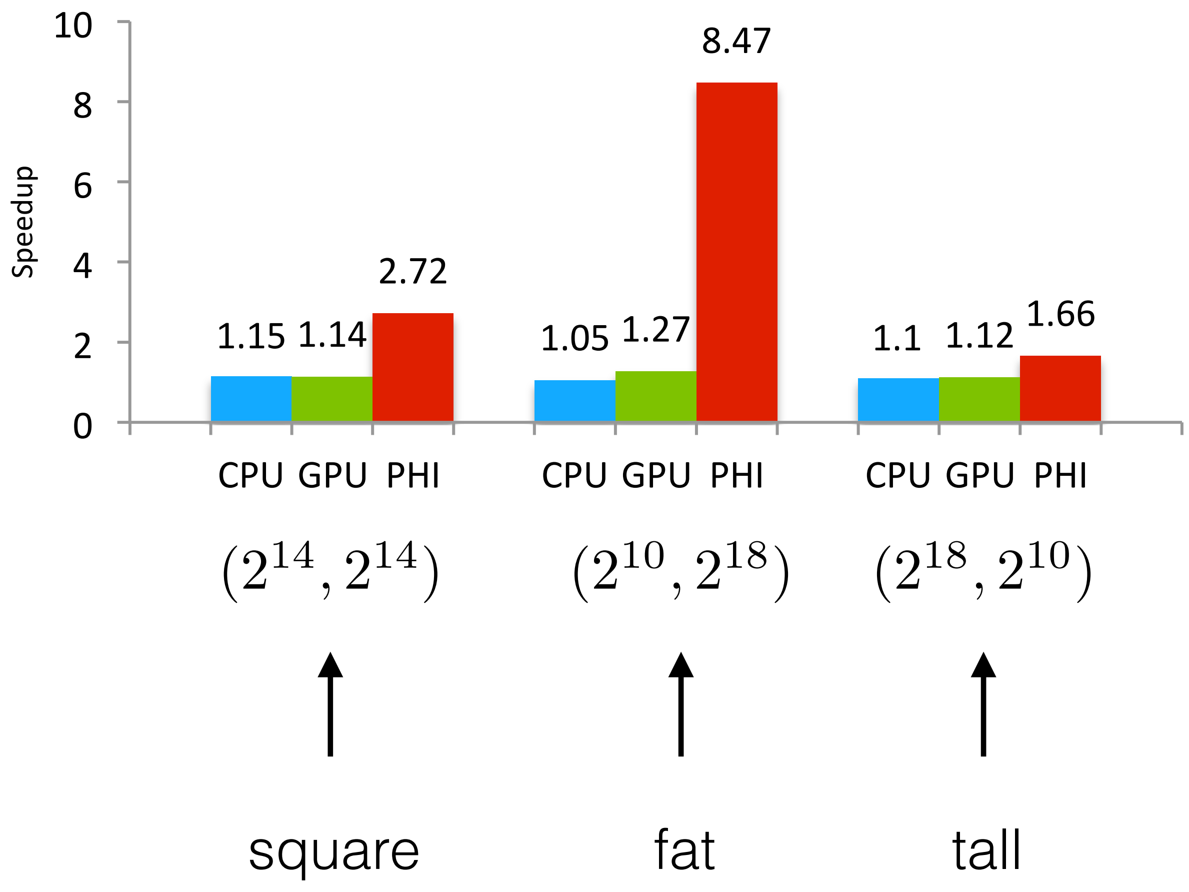
OpenCL Sourcecode

- Iteration over the P_{sq} partitions.
- Sequential computations on a P_{wi} partition in private memory.
- Summing up the WIs' results in local memory.
- Storing the WG's result in global memory.

```
__kernel void gemv_fst( __global float* in_matrix,
                        __global float* in_vector,
                        __global float* out_vector,
{

  // private memory for a WI's computation
  __private float res_prv = 0.0f;

  // local memory for a WG's computation
  __local   float res_lcl[ NUM_WI_1 ][ NUM_WI_2 ];

  // iteration over P_sq blocks
  for( int i_sq = 1 ; i_sq <= NUM_SQ_1 ; ++i_sq ) {
    for( int j_sq = 1 ; j_sq <= NUM_SQ_2 ; ++j_sq ) {
      res_prv = 0.0f;

      // sequential computation on a P_wi partition
      for( int i = 1 ; i <= WI_PART_SIZE_1 ; ++i )
        for( int j = 1 ; j <= WI_PART_SIZE_2 ; ++j )
        res_prv += my_p_wi( i, j, 0 ) * my_p_wi( i, j, 1 );

      // store result in local memory
      res_lcl[ WI_ID_1 ][ WI_ID_2 ] = res_prv;

      barrier( CLK_LOCAL_MEM_FENCE );

      // combine the WIs' results in dimension x
      for( int stride = NUM_WI_2 / 2 ; stride > 0 ; stride /= 2)
      {
        if( WI_ID_2 < stride)
          res_lcl[ WI_ID_1 ][ WI_ID_2 ] += res_lcl[ WI_ID_1 ][ WI_ID_2 + stride ];

        barrier( CLK_LOCAL_MEM_FENCE );
      }

      // store WGs' results in global memory
      if( WI_ID_2 == 0 )
        my_res( i_sq ) = res_lcl[ WI_ID_1 ][0];

      barrier( CLK_LOCAL_MEM_FENCE );

    } // end of for-loop j_sq
  } // end of for-loop i_sq
} // end of kernel
```

Evaluation

- We compared our implementation with
  1. the Intel MKL library on an Intel CPU,
  2. the NVIDIA cuBLAS library on an NVIDIA GPU and
  3. the MAGMA MIC library on an Intel Xeon Phi co-processor.
- We used the OpenTuner framework to determine optimized numbers of WGs and WIs.
- Search time was on average:
  - 3.25min on the CPU (13 configs → 15sec/conf)
  - 9.53min on the GPU (44 configs → 13sec/conf)
  - 24.57min on the PHI (67 configs → 22sec/conf)



Conclusion

- MDHs are an extension of LHs to capture parallelism in multiple dimensions.
- `md_hom` is a pattern to conveniently express MDH functions.
- `md_hom` can be implemented as generic OpenCL code that can automatically be optimized for different hardware and input sizes.
- Performance for GEMV better than hand-tuned code (speedups up to 8x) on three different device architectures and for different input sizes.

Questions?