Int J Parallel Prog (2018) 46:101–119
https://doi.org/10.1007/s10766-017-0508-z

Multi-dimensional Homomorphisms and Their Implementation in OpenCL

Ari Rasch[1] · **Sergei Gorlatch**[1]

Received: 4 October 2016 / Accepted: 2 May 2017 / Published online: 8 May 2017

Abstract Homomorphisms (traditionally defined on lists) are functions that can be parallelized by the divide-and-conquer paradigm. In this paper, we introduce an extension of the traditional homomorphism concept—*multi-dimensional homomorphisms (MDHs)*—which capture parallelism on multi-dimensional arrays. We propose `md_hom`—a new parallel pattern (a.k.a. algorithmic skeleton), based on the MDH concept, to simplify parallel programming for a broad class of applications. The `md_hom` pattern is general enough to subsume common parallel patterns such as `map` and `reduce`, and also more complex functions built by composing and nesting several patterns. We present a generic implementation schema for `md_hom` in form of an efficient, correct-by-construction OpenCL pseudocode that targets various parallel architectures such as multi-core CPU and graphics processing unit (GPU). We develop our pseudocode schema as parametrized in tuning parameters: these allow to optimize the code for different devices and input sizes by performing an automated search on the parameter space. We evaluate the schematically generated, executable OpenCL code using the example of *general matrix–vector multiplication (GEMV)*—an important linear algebra routine which has gained more attention recently due to its use in the application area of deep learning—on two parallel architectures—Intel CPU and NVIDIA GPU. Our performance results are competitive and in some cases even better than the hand-tuned GEMV implementations provided by the state-of-the-art libraries Intel MKL and NVIDIA cuBLAS, as well as the auto-tunable OpenCL BLAS library CLBlast.

✉ Ari Rasch
a.rasch@uni-muenster.de

Sergei Gorlatch
gorlatch@uni-muenster.de

[1] Department of Mathematics and Computer Science, University of Muenster, Münster, Germany

Keywords Homomorphisms · Algorithmic skeletons · OpenCL · Auto-tuning · Multi-core · GPU computing · Matrix–vector multiplication

1 Introduction

A function h on lists is called a *List Homomorphism (LH)* iff there exists a *combine operator* $\circledast$ such that

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

where $+\!\!+$ denotes list concatenation, i.e., h can be computed by splitting its input in chunks x and y, computing h in parallel on x and y, and then combining the partial results $h(a)$ and $h(b)$ by using $\circledast$. List homomorphisms are a well-studied class of functions for which efficient parallel implementation schemes, e.g., in MPI, have been developed [7].

One major weakness of traditional LHs is that they are defined on lists and thus capture parallelism in only one dimension. For example, in case of *general matrix–vector multiplication (GEMV)* [15] there is a potential of parallelism in two dimensions: row and column. For exploiting parallelism in the row dimension, the input matrix is split horizontally in chunks that are in parallel multiplied with the input vector; the obtained results are combined to the final result by using concatenation. To exploit parallelism in the column dimension, the matrix is split in vertical chunks: each of these chunks is multiplied with the corresponding chunk of the input vector, and the results are finally combined by using vector addition. If we treat GEMV as an LH, then only one dimension of parallelism can be captured. However, modern multi- and many-core devices, such as multi-core CPUs and Graphics Processing Units (GPUs) require massive parallelization in multiple dimensions in order to utilize their full performance potential.

In this paper, we extend the LH concept to *multi-dimensional homomorphisms (MDHs)*, in order to capture parallelism in multiple dimensions. We introduce formally a new parallel pattern (a.k.a. algorithmic skeleton [8]), named `md_hom`, based on the MDH concept, which is suitable to conveniently express a broad class of functions. For example, the `md_hom` pattern subsumes common parallel patterns such as `map` and `reduce`, and also more complex functions built by composing and nesting such patterns. We prove formally that each MDH can be expressed by using `md_hom` and vice versa.

We develop a generic implementation schema for `md_hom` in form of an OpenCL [14] pseudocode that targets modern multi- and many-core devices. For this, we formally decompose its computations in independent parts that together represent a semantically-sound decomposition schema for OpenCL. The formal approach makes our code correct by construction and guarantees, for example, the correctness of the usually error-prone index expressions.

Our implementation schema is developed as parametrized in so-called *tuning parameters*, e.g., the number of threads. Different configurations of tuning parameters induce semantically-equal but differently-optimized code variants. We use these tuning parameters to automatically optimize our code by performing an automatized search (a.k.a. auto-tuning) on the parameter space.

To demonstrate the runtime performance of our executable OpenCL code, we use the example of general matrix–vector multiplication (GEMV)—a linear algebra routine widely used in the area of deep learning [12]. We compare the performance of our GEMV implementation with highly tuned, high-performance libraries on two different processor architectures—Intel MKL in case of the CPU and NVIDIA cuBLAS for the GPU. Moreover, we compare to the auto-tunable OpenCL BLAS library CLBlast [3] which has proven to have competitive performance compared to vendor implementations (e.g., cuBLAS) on different device architectures and for different input sizes [22]. Our experiments demonstrate a competitive and sometimes even better performance of our implementation than provided by the reference implementations.

The structure of the paper is as follows. In Sect. 2, we define the class of MDHs as an extension of LHs and introduce the pattern `md_hom` for which we prove formally that each MDH can be expressed by using this pattern. An OpenCL implementation schema for the `md_hom` pattern, based on a mathematically sound decomposition schema, is presented in Sect. 3 and evaluated by using the example of GEMV in Sect. 4. In Sect. 5, we compare our approach to related work, and we conclude in Sect. 6.

2 Multi-dimensional Homomorphisms and the `md_hom` Pattern

In this section, we introduce multi-dimensional homomorphisms (MDHs)—a broad class of functions that can be efficiently parallelized—and we define the programming pattern `md_hom` to conveniently express MDH functions.

2.1 Multi-dimensional Homomorphisms

To define multi-dimensional homomorphisms (MDHs), we first introduce multi-dimensional arrays (MDAs)—the data type on which MDHs operate.

Definition 1 (*Multi-Dimensional Array*) Let T be an arbitrary type (e.g, `float`), $d \in \mathbb{N}$ a natural number and $(N_1, \dots, N_d) \in \mathbb{N}^d$ a tuple of d natural numbers. A *multi-dimensional array (MDA)* of dimensionality d, size $(N_1, \dots, N_d)$ and with elements in T is a function with the following signature:

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

We use a syntax similar to `C++` for denoting MDAs, i.e., we write

– $T[N_1]\dots[N_d]$ for the set of all MDAs of dimension d, size $(N_1, \dots, N_d)$ and elements in T, and
– $a[i_1]\dots[i_d]$ for the element of the MDA $a \in T[N_1]\dots[N_d]$ that is accessed by the indices $i_1, \dots, i_d$.

Definition 2 (*MDA Concatenation*) The *concatenation* in the k-th dimension of two d-dimensional MDAs, $k \leq d$, which have the same size in all dimensions, except probably in dimension k, is defined as binary function:

$$\begin{aligned} +\!\!+_k : T[N_1]\ldots\underset{\underset{k}{\uparrow}}{[P]}\ldots[N_d] \times T[N_1]\ldots\underset{\underset{k}{\uparrow}}{[Q]}\ldots[N_d] \\ \to T[N_1]\ldots\underset{\underset{k}{\uparrow}}{[P+Q]}\ldots[N_d] \end{aligned}$$

where

$$(a +\!\!+_k b)[i_1]\ldots[i_d] = \begin{cases} a[i_1] \quad \ldots \quad [i_d], & \text{if } i_k \leq P \\ b[i_1]\ldots[i_k - P]\ldots[i_d], & \text{otherwise} \end{cases}$$

Definition 3 (*Multi-Dimensional Homomorphism*) Let T and T' be two arbitrary types. A function $h : T[N_1]\ldots[N_d] \to T'$ on d-dimensional MDAs is called a *multi-dimensional homomorphism (MDH)* of the dimensionality d iff there exist *combine operators* $\circledast_1, \ldots, \circledast_d : T' \times T' \to T'$, such that for each $k \in [1, d]$ and arbitrary, concatenated input MDA $a +\!\!+_k b$:

$$h(a +\!\!+_k b) = h(a) \circledast_k h(b)$$

In words: the value of h on a concatenated MDA in dimension k can be computed by applying h to the MDA's chunks a and b and combining the results afterwards by using the combine operator $\circledast_k$—we say the computation of h can be *decomposed* in two parts in dimension k. Since the computations of $h(a)$ and $h(b)$ are independent of each other, they can be performed in parallel.

We illustrate the definition for the case $d = 2$. For computing h on its complete (concatenated) input MDA, we can

1. split it horizontally (i.e., in dimension 1) in chunks a_1 and b_1, apply h independently to the chunks, and combine the results by operator $\circledast_1$ (Fig. 1a), or
2. split it vertically (i.e., in dimension 2) in chunks a_2 and b_2, apply h to the chunks, and combine the results by operator $\circledast_2$ (Fig. 1b).

In contrast to the traditional LHs, whose computation can be decomposed in only one dimension, MDHs allow for decomposition in several dimensions and thus (as we

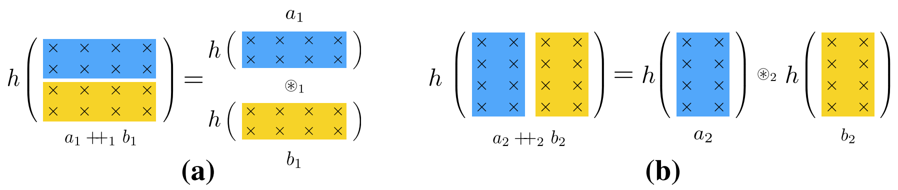


Fig. 1 Illustration of Definition 3 for $d = 2$

Table 1 Examples of views

View	Definition
`pair_vv`	$(v_1,\ldots,v_N),(w_1,\ldots,w_N)\ \mapsto\ ((v_1,w_1),\ldots,(v_N,w_N))$
`pair_mv`	$\begin{pmatrix} m_{11} & \ldots & m_{1N} \\ \vdots & \ddots & \vdots \\ m_{M1} & \ldots & m_{MN}\end{pmatrix},\begin{pmatrix} v_1 \\ \vdots \\ v_N\end{pmatrix} \mapsto \begin{pmatrix} (m_{11},v_1) & \ldots & (m_{1N},v_N) \\ \vdots & \ddots & \vdots \\ (m_{M1},v_1) & \ldots & (m_{MN},v_N)\end{pmatrix}$
`pair_mm`	$\begin{pmatrix} m_{11} & \ldots & m_{1K} \\ \vdots & \ddots & \vdots \\ m_{M1} & \ldots & m_{MK}\end{pmatrix},\begin{pmatrix} m'_{11} & \ldots & m'_{1N} \\ \vdots & \ddots & \vdots \\ m'_{K1} & \ldots & m'_{KN}\end{pmatrix} \mapsto \begin{pmatrix} a_{11} & \ldots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \ldots & a_{MN}\end{pmatrix}$ where $a_{ij}\ =\ ((m_{i1},m'_{1j})\,,\ldots,\ (m_{iK},m'_{Kj}))$

will see at the end of this section) enable a more fine-grained parallelism which is highly required to utilize the full performance potential of modern parallel architectures.

In the following, we present how three important BLAS routines [15] can be expressed as MDHs: (1) `dot` which computes a dot-product, (2) `gemv` for computing (general) matrix–vector multiplication and, (3) `gemm` for computing (general) matrix–matrix multiplication. These routines differ slightly from MDHs in their input types: they operate on one or more vectors and/or matrices (i.e., 1- or 2-dimensional MDAs) instead of a single MDA. For simplicity, we ignore that according to their usual definitions [15], `gemv` and `gemm` also expect two scaling factors and allow to optionally transpose the input matrix.

To adapt the routines' input, we use *views*—functions that fuse several input parameters to one MDA.

Table 1 lists the three views that we use for expressing the BLAS routines as MDHs: (1) `pair_vv` to fuse the two input vectors of `dot`, (2) `pair_mv` for `gemv`, and (3) `pair_mm` for `gemm`. They perform pairing of two vectors, a matrix and a vector or of two matrices, correspondingly.

Table 2 lists three MDHs: `md_dot`, `md_gemv` and `md_gemm`. We use these in combination with the corresponding view to express `dot`, `gemv` and `gemm`:

- $\texttt{dot} \ \ = \texttt{md\_dot} \ \ \circ \texttt{pair\_vv}$,
- $\texttt{gemv} = \texttt{md\_gemv} \circ \texttt{pair\_mv}$,
- $\texttt{gemm} = \texttt{md\_gemm} \circ \texttt{pair\_mm}$.

Here, operator $\circ$ denotes functional composition, i.e., $(f\ \circ\ g)(x) = f(g(x))$.

For example, to express `gemv` as an MDH, we use the view `pair_mv` to fuse the $(M \times N)$ input matrix and the input vector of size N—both of floating point numbers—to one MDA with M pairs of floating point numbers in the dimension 1, and N pairs in the dimension 2. The MDA is afterwards processed to the result vector of size M by using the MDH `md_gemv`:

$$\underbrace{\texttt{float}[M][N]}_{\text{input matrix}},\ \underbrace{\texttt{float}[N]}_{\text{input vector}} \overset{\texttt{pair\_mv}}{\rightarrow} \underbrace{\texttt{float}^2[M][N]}_{\text{MDA}} \overset{\texttt{md\_gemv}}{\rightarrow} \underbrace{\texttt{float}[M]}_{\text{result vector}}$$

Table 2 BLAS routines represented as MDH functions

MDH	Definition
`md_dot`	$((v_1, w_1), \dots, (v_N, w_N)) \mapsto v_1 * w_1 + \dots + v_N * w_N$
`md_gemv`	$\begin{pmatrix} (m_{11}, v_1) & \dots & (m_{1N}, v_N) \\ \vdots & \ddots & \vdots \\ (m_{M1}, v_1) & \dots & (m_{MN}, v_N) \end{pmatrix} \mapsto \begin{pmatrix} m_{11} * v_1 + & \dots & + m_{1N} * v_N \\ & \vdots & \\ m_{M1} * v_1 + & \dots & + m_{MN} * v_N \end{pmatrix}$
`md_gemm`	$\begin{pmatrix} a_{11} & \dots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \dots & a_{MN} \end{pmatrix} \mapsto \begin{pmatrix} b_{11} & \dots & b_{1N} \\ \vdots & \ddots & \vdots \\ b_{M1} & \dots & b_{MN} \end{pmatrix}$ where $a_{ij} = ((m_{i1}, m'_{1j}), \dots, (m_{iK}, m'_{Kj}))$ and $b_{ij} = (m_{i1} * m'_{1j} + \dots + m_{iK} * m'_{Kj})$

Table 3 Combine operators of MDH functions

MDH	$\circledast_1$	$\circledast_2$	$\circledast_3$
`md_dot`	$+$		
`md_gemv`	$+\!\!+_1$	$+_{\text{vec}}$	
`md_gemm`	$+\!\!+_1$	$+\!\!+_2$	$+_{\text{mat}}$

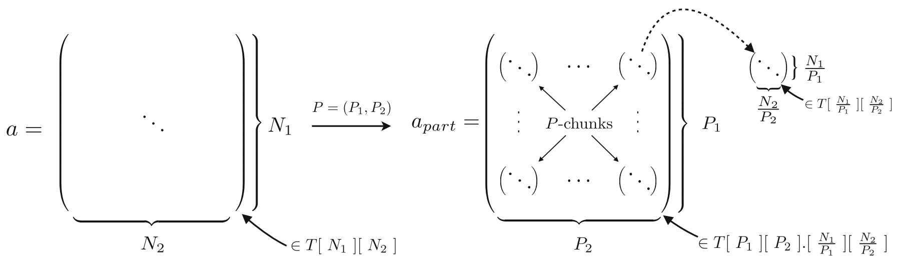
Table 3 shows the combine operators of the three MDHs from Table 2. Here, $+\!\!+_1$ and $+\!\!+_2$ denote MDA concatenation in the first and second dimension as defined in Definition 2, and $+_{vec}$ and $+_{mat}$ are element-wise vector and matrix addition, correspondingly. For example, in case of `md_gemv`, we can decompose its computation in the first dimension by splitting its 2-dimensional input MDA in chunks of rows and combine the results by using concatenation $+\!\!+_1$, or in the second dimension by splitting its input MDA in chunks of columns and use vector addition $+_{\text{vec}}$ to combine the results. Note that `md_dot` is an 1-dimensional MDH with one combine operator, while `md_gemv` and `md_gemm` are 2- or 3-dimensional and thus have 2 or 3 combine operators, correspondingly.

So far, we have defined MDHs as functions whose computation can be decomposed in two independent parts that can be computed in parallel. In general, an MDH's computation can be decomposed in multiple parts by splitting its input MDA in several chunks to which we refer as *partitioning*.

Definition 4 (*MDA Partitioning*) Let $a \in T[N_1]\dots[N_d]$ be an MDA of dimension d and size $(N_1, \dots, N_d)$ and $P = (P_1, \dots, P_d) \in \mathbb{N}^d$ a d-tuple of natural numbers where N_i is divisible by P_i. The *P-partitioning* of a is the d-dimensional MDA

$$a_{\text{part}} \in (T[\tfrac{N_1}{P_1}]\dots[\tfrac{N_d}{P_d}])[P_1]\dots[P_d]$$

of size $(P_1, \dots, P_d)$ which has as its elements d-dimensional MDAs of size $(\frac{N_1}{P_1}, \dots, \frac{N_d}{P_d})$ where for all $p_k \in [1, P_k]$ and $i_k \in [1, \frac{N_k}{P_k}]$:



$a =$
N_1
N_2
$\in T[\ N_1\ ][\ N_2\ ]$
$P = (P_1, P_2)$
$a_{part} =$
P-chunks
P_1
P_2
$\in T[\ P_1\ ][\ P_2\ ].[\ \frac{N_1}{P_1}\ ][\ \frac{N_2}{P_2}\ ]$
$\frac{N_1}{P_1}$
$\frac{N_2}{P_2}$
$\in T[\ \frac{N_1}{P_1}\ ][\ \frac{N_2}{P_2}\ ]$

Fig. 2 Partitioning of a 2-dimensional MDA

$$(a_{\text{part}}[\ p_1] \ldots [\ p_d])[\ i_1] \ldots [\ i_d] = a[\ (p_1 - 1) * \frac{N_1}{P_1} + i_1] \ldots [\ (p_d - 1) * \frac{N_d}{P_d} + i_d]$$

We refer to P as *partitioning schema* for a, and to $a_{\text{part}}[p_1] \ldots [p_d]$ as a *P-chunk of a*.

Notation 1 For a better readability, we write in the following

1. $T[P_1] \ldots [P_d] \cdot [\frac{N_1}{P_1}] \ldots [\frac{N_d}{P_d}]$ instead of $(T[\frac{N_1}{P_1}] \ldots [\frac{N_d}{P_d}])[P_1] \ldots [P_d]$, i.e., we first state the number of chunks in each of the d dimensions $P_1, \ldots, P_d$ and then the number of elements per chunk: $\frac{N_1}{P_1}, \ldots, \frac{N_d}{P_d}$, and
2. $a_{\text{part}}[p_1] \ldots [p_d] \cdot [i_1] \ldots [i_d]$ instead of $(a_{\text{part}}[p_1] \ldots [p_d])[i_1] \ldots [i_d]$ to reduce the number of parentheses.

Figure 2 demonstrates MDA partitioning for a 2-dimensional MDA $a \in T[N_1][N_2]$: a is partitioned according to the partitioning schema $P = (P_1, P_2)$ in $(P_1 * P_2)$ P-chunks where each P-chunk comprises $\frac{N_1}{P_1}$ elements in the dimension 1 and $\frac{N_2}{P_2}$ elements in dimension 2, i.e., it is in $T[\frac{N_1}{P_1}][\frac{N_2}{P_2}]$. The P-chunks are arranged in an MDA $a_{\text{part}} \in (T[\frac{N_1}{P_1}][\frac{N_2}{P_2}])[P_1][P_2]$ for which we write $a_{\text{part}} \in T[P_1][P_2].[\frac{N_1}{P_1}][\frac{N_2}{P_2}]$ according to Notation 1.

Proposition 1 (MDH Decomposition) *If h is a d-dimensional MDH with combine operators $\circledast_1, \ldots, \circledast_d$, a is an input MDA of h and a_{part} a P-partitioning of a for an arbitrary partitioning schema $P = (P_1 \ldots, P_d)$, then*

$$h(a) = \underset{p_1 \in [1, P_1]}{\circledast_1} \ldots \underset{p_d \in [1, P_d]}{\circledast_d} h(\, a_{part}[p_1] \ldots [p_d])$$

Here, $\underset{i \in [1,N]}{\circledast} a[i]$ denotes $a[1] \circledast \ldots \circledast a[N]$ and $\underset{i \in [1,N_1]}{\circledast_1} \underset{j \in [1,N_2]}{\circledast_2} a[i][j]$ denotes $(a[1][1] \circledast_2 \ldots \circledast_2 a[1][N_2]) \circledast_1 \ldots \circledast_1 (a[N_1][1] \circledast_2 \ldots \circledast_2 a[N_1][N_2])$, etc.

Proof Follows by nested induction on d and $P_1, \ldots, P_d$. □

In words: to compute the MDH h on its complete input MDA a, we can apply h independently to the $P_1 * \ldots * P_d$ chunks comprised by a's P-partitioning a_{part} and then combine the results in each of the d dimensions by using the corresponding combine operator. Note that MDHs allow for a decomposition in up to $N_1 * \ldots * N_d$ parts (by using $P = (N_1, \ldots, N_d)$ as partitioning schema), while LHs can be decomposed in only one dimension i and consequently in only N_i parts.

To decompose MDHs for recent device architectures which are usually programmed by using a layered thread hierarchy (such as in the OpenCL approach), we introduce *recursive MDA partitioning* and, accordingly, *recursive MDH decomposition*.

Definition 5 (*Recursive MDA Partitioning*) Let $a \in T[\,N_1]\ldots[\,N_d]$ be a d-dimensional MDA and $(P^1, \ldots, P^n)$ a sequence of partitioning schemas where $P^i = (P^i_1, \ldots, P^i_d)$ and where $\frac{N_j}{P^1_j * \ldots * P^{i-1}_j}$ is divisible by P^i_j, i.e.,

- P^1 is a partitioning schema for a,
- P^2 is a partitioning schema for the P_1-chunks,
- P^3 is a partitioning schema for the P_2-chunks, and so on.

We refer to the sequence $(P^1, \ldots, P^n)$ as *n-fold recursive partitioning schema* for a, and to the MDA

$$a_{\text{part}} \in T[\,P^1_1]\ldots[\,P^1_d].\ \ldots\ .[\,P^n_1]\ldots[\,P^n_d].[\tfrac{N_1}{P^1_1 * \ldots * P^n_1}]\ldots[\tfrac{N_d}{P^1_d * \ldots * P^n_d}]$$

that is obtained by partitioning a according to P_1 and then partition each P_1-chunk further according to P_2 and so on, as *n-fold recursive $(P^1, \ldots, P^n)$-partitioning* of a.

Figure 3 demonstrates a recursive partitioning of a 2-dimensional MDA $a \in T[N_1][N_2]$ using a 2-fold partitioning schema (P^1, P^2). First, the MDA is partitioned according to P^1 in $(P^1_1 * P^1_2)$ P^1-chunks where each chunk comprises $\frac{N_1}{P^1_1}$ elements in its first dimension and $\frac{N_2}{P^1_2}$ in its second. Afterwards, each of the P^1-chunks is further partitioned according to P^2 in $(P^2_1 * P^2_2)$ P^2-chunks where each chunk comprises $\frac{N_1}{P^1_1 * P^2_1}$ elements in its first dimension and $\frac{N_2}{P^1_2 * P^2_2}$ elements in its second dimension.

Proposition 2 (Recursive MDH Decomposition) *If h is a d-dimensional MDH with combine operators $\circledast_1, \ldots, \circledast_d$, a is an input MDA of h and a_{part} an n-fold recursive $(P^1, \ldots, P^n)$-partitioning of a, then*

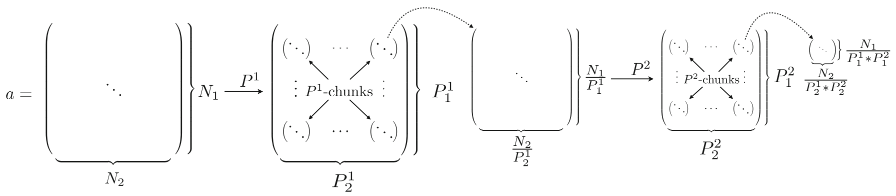


Fig. 3 Partitioning of a 2-dimensional MDA by a 2-fold partitioning schema

$$h(a) = \underset{p_1^1\in[1,P_1^1]}{\circledast_1} \dots \underset{p_d^1\in[1,P_d^1]}{\circledast_d} \ \ddots \ \underset{p_1^n\in[1,P_1^n]}{\circledast_1} \dots \underset{p_d^n\in[1,P_d^n]}{\circledast_d} \; h(\, a_{part}[\,p_1^1] \dots [\,p_d^1]. \;\; \dots \;\; .[\,p_1^n] \dots [\,p_d^n])$$

Proof Follows from Proposition 1 by applying it n times. □

For example, the recursive decomposition of a 2-dimensional MDH h on an input MDA $a \in T[N_1][N_2]$ that is partitioned according to $P = (P^1, P^2)$ (as depicted in Fig. 3) is as follows:

$$h(a) = \underbrace{\underset{p_1^1\in[1,P_1^1]}{\circledast_1}}_{(4)} \underbrace{\underset{p_2^1\in[1,P_2^1]}{\circledast_2}}_{(3)} \underbrace{\underset{p_1^2\in[1,P_1^2]}{\circledast_1}}_{(2)} \underbrace{\underset{p_2^2\in[1,P_2^2]}{\circledast_2}}_{(1)} \underbrace{h(\, a_{\text{part}}[\,p_1^1][\,p_2^1].[\,p_1^2][\,p_2^2])}_{(0)}$$

In this representation, we first compute (0) by applying h to all P^2-chunks. We then combine the results of the chunks that are within the same P_1-chunk in dimension 2 by using $\circledast_2$ and in dimension 1 by using $\circledast_1$, which represent the computations (1) and (2). We obtain one result per P_1-chunk, i.e., $P_1^1 * P_2^1$ results; we combine these analogously in both dimensions by using the corresponding combine operators, i.e., we perform computations (3) and (4).

2.2 The `md_hom` Pattern

A pattern (skeleton) is a higher-order function for which we can generate executable program code. In this section, we introduce formally the new pattern `md_hom` and we show that each MDH function can be conveniently expressed by using this pattern.

Definition 6 Let T and T' be two arbitrary types and $d \in \mathbb{N}$ a natural number. The function `md_hom` with the signature

$$\texttt{md\_hom}: \; (\underbrace{T \to T'}_{f}, \; (\underbrace{T' \times T' \to T'}_{\circledast_i})_{i\in[1,d]}) \;\not\to\; \underbrace{(T[N_1]\dots[N_d] \;\to\; T')}_{\texttt{md\_hom}(f \;,\; (\circledast_1,\dots,\circledast_d))}$$

is a partial function (indicated by the $\not\to$ instead of $\to$) that takes as its input a function $f : T \to T'$ and a tuple of d further functions $\circledast_i : T' \times T' \to T'$, $i \in [1, d]$, and yields a function `md_hom`$(f \;,\; (\circledast_1, \dots, \circledast_d))$ which is defined on d-dimensional MDHs as follows:

$$\texttt{md\_hom}(f \;,\; (\circledast_1, \dots, \circledast_d))(a) \;=\; \underset{i_1\in[1,N_1]}{\circledast_1} \dots \underset{i_d\in[1,N_d]}{\circledast_d} \; f(a[i_1]\dots[i_d])$$

for that we require the homomorphic property, i.e., for each $i \in [1, d]$:

$$\begin{aligned}&\texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))(\ a_1 +\!\!+_i\ a_2) \ = \\ &\quad \texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))(a_1\ )\ \circledast_i\ \ \texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))(a_2\ )\end{aligned}$$

We refer to f as the *scalar function* of $\texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))$ since it operates on elements of T and to $\circledast_1, \ldots, \circledast_d$ as its *combine operators*, correspondingly.

We define md_hom as a partial function since $\texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))$ may not be well defined for some scalar functions f and combine operators $(\circledast_1, \ldots, \circledast_d)$. For example, $\texttt{md\_hom}(\texttt{id}\ ,\ (\texttt{max}_{\text{vec}}, +_{\text{vec}}))$, where id is the identity function and $\texttt{max}_{\text{vec}}$ is the element-wise maximum function on two vectors, is not well defined (the homomorphic property is not met).

We define md_hom so that MDHs can be conveniently expressed by using this pattern; we prove this in the following proposition.

Proposition 3 *Let $h : T[N_1]\ldots[N_d] \to T'$ be a d-dimensional MDH with combine operators $\circledast_1, \ldots, \circledast_d$. Let further be $f : T \to T'$ the function with $h(a) = f(\ a\underbrace{[1]\ldots[1]}_{d\text{-times}})$ for each $a \in T\underbrace{[1]\ldots[1]}_{d\text{-times}}$. Then it holds:*

$$h \ = \ \texttt{md\_hom}(f\ ,\ (\circledast_1, \ldots, \circledast_d))$$

Proof Follows from Proposition 1 for $P = (N_1,\ \ldots\ , N_d)$. □

In words: to express an MDH by using md_hom, we only have to know the MDH's behavior f on scalar values and its combine operators $\circledast_1, \ldots, \circledast_d$. Note that the other direction also holds: each function that is expressed by using md_hom is an MDH; this follows from the homomorphic property in Definition 6.

Table 4 shows the md_hom representation for the MDHs md_dot, md_gemv and md_gemm which we introduced in Table 2. For example, in case of md_gemv, the input MDA consists of pairs of floating point numbers. The two elements of each pair are multiplied by using $*$ and the results are combined in the MDA's two dimensions by using the combine operators $+\!\!+_1$ and $+_{\text{vec}}$. Table 4 also lists md_hom representations for two popular parallel patterns: map(f) which maps a user-defined function f to each element of a vector, and reduce($\oplus$) which combines the elements of a vector by using a binary operator $\oplus$.

Table 4 md_hom representations of MDH functions

MDH	md_hom representation
md_dot	$\texttt{md\_hom}(\ *\ ,\ (+))$
md_gemv	$\texttt{md\_hom}(\ *\ ,\ (+\!\!+_1, +_{\text{vec}}))$
md_gemm	$\texttt{md\_hom}(\ *\ ,\ (+\!\!+_1, +\!\!+_2, +_{\text{mat}}))$
map(f)	$\texttt{md\_hom}(\ \texttt{f}\ \ ,\ (+\!\!+_1))$
reduce($\oplus$)	$\texttt{md\_hom}(\ \texttt{id}\ ,\ (\oplus))$

3 OpenCL Implementation of the `md_hom` Pattern

In this section, we demonstrate how functions that are expressed by using the `md_hom` pattern can be implemented efficiently in OpenCL which is portable across a wide range of parallel architectures, e.g., multi-core CPU and GPU. The execution model of OpenCL has 3 layers: (1) the programmer starts in parallel a user-defined number of thread bundles—so-called *work-groups (WGs)*, (2) each work-group comprises a user-defined number of parallel threads to which OpenCL refers as *work-items (WIs)*, (3) each thread runs sequentially.

Let us consider an arbitrary instantiation of the `md_hom` pattern:

$$h = \texttt{md\_hom}(f\ ,\ (+\!\!+_1, \ldots, +\!\!+_{d_l}, \circledast_1, \ldots, \circledast_{d_r}))$$

whose first d_l combine operators are concatenations and the remaining d_r operators are arbitrary. We consider concatenation as a combine operator specifically, since it occurs in many important applications, e.g., in `md_map`, `md_gemv` and `md_gemm` introduced in Sect. 2.

To parallelize the computation of h in OpenCL, we decompose it in independent parts that we distribute to the different WGs and WIs. OpenCL allows to arrange WGs and WIs in up to three dimensions. We use $d_l + d_r$ dimensions—one for each dimension of our `md_hom` instantiation. In case that d is greater than three, we arrange the higher dimensions in a row-major order in the third dimension. To be able to choose a suitable number of WGs and WIs for a particular target device and input size, we introduce them as parameters:

- $\texttt{NUM\_WG\_L\_1}, \ldots, \texttt{NUM\_WG\_L\_d}_{\texttt{l}}, \texttt{NUM\_WG\_R\_1}, \ldots, \texttt{NUM\_WG\_R\_d}_{\texttt{r}}$;
- $\texttt{NUM\_WI\_L\_1}, \ldots, \texttt{NUM\_WI\_L\_d}_{\texttt{l}}, \texttt{NUM\_WI\_R\_1}, \ldots, \texttt{NUM\_WI\_R\_d}_{\texttt{r}}$;

where the first line represents the number of WGs in the $d_l + d_r$ dimensions, and the second line represents the number of WIs per WG, correspondingly.

We decompose the computation of h according to Proposition 2 recursively in independent parts that are computed by separate WGs and WIs. For this, we define a 3-fold recursive partitioning schema $P_{\texttt{OpenCL}} = (P_{\text{wg}}, P_{\text{sq}}, P_{\text{wi}})$ according to Definition 5. This 3-fold schema directly reflects the 3-layer hierarchy of computations in OpenCL described at the beginning of this section, as follows. We process P_{wg}-chunks in parallel by WGs, P_{sq}-chunks sequentially, and P_{wi}-chunks in parallel by WIs, where

- $P_{\text{wg}} = (\texttt{NUM\_WG\_L\_1}, \ldots, \texttt{NUM\_WG\_L\_d}_{\texttt{l}}, \texttt{NUM\_WG\_R\_1}, \ldots, \texttt{NUM\_WG\_R\_dr})$,
- $P_{\text{sq}} = (\texttt{NUM\_SQ\_L\_1}, \ldots, \texttt{NUM\_SQ\_L\_d}_{\texttt{l}}\ , \texttt{NUM\_SQ\_R\_1}, \ldots, \texttt{NUM\_SQ\_R\_d}_{\texttt{r}})$,
- $P_{\text{wi}} = (\texttt{NUM\_WI\_L\_1}, \ldots, \texttt{NUM\_WI\_L\_d}_{\texttt{l}}\ , \texttt{NUM\_WI\_R\_1}, \ldots, \texttt{NUM\_WI\_R\_d}_{\texttt{r}})$

and

$$\texttt{NUM\_SQ\_L\_i} = \frac{M_i}{\texttt{NUM\_WG\_L\_i} * \texttt{NUM\_WI\_L\_i}}\ ,$$
$$\texttt{NUM\_SQ\_R\_j} = \frac{N_j}{\texttt{NUM\_WG\_R\_j} * \texttt{NUM\_WI\_R\_j}}$$

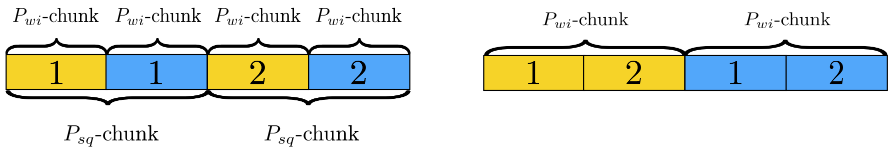


Fig. 4 Memory accesses with and without P_{sq}-chunks

Note that instead of partitioning the P_{wg}-chunks according to P_{wi}, we first partition them according to P_{sq} which we define so that each P_{sq}-chunk comprises the number of elements equal to the number of WIs per WG. This causes the P_{wi}-chunks to comprise a single element which has the following advantage. Modern devices provide high performance when a hardware-defined number of work-items (referred to as *warp size* in NVIDIA GPUs and *SIMD-width* in Intel CPUs) with consecutive IDs access in parallel consecutive memory regions. Such an access pattern enables GPUs to fuse memory accesses (a.k.a. *coalescing*) and CPUs to automatically vectorize computations [10,17].

Figure 4 (left) illustrates our approach: by partitioning according to P_{sq} and then according to P_{wi}, we ensure an advantageous access pattern that is depicted in the figure for the example of a 1-dimensional P_{wg}-chunk of size 4, with 2 WIs per WG. Both WIs iterate over the P_{sq}-chunks and process in each iteration the number of elements equal to the number of WIs in a P_{sq}-chunk (2 in the example). The memory accesses of different WIs are colored differently; the numbers denote in which iteration an element is accessed. For each P_{sq}-chunk (and thus in each iteration), different WIs access consecutive elements in parallel. For comparison, Fig. 4 (right) shows what happens without partitioning according to P_{sq}: each WI processes a range of consecutive elements which usually causes a poor performance on modern devices.

Figure 5 shows the first decomposition schema that we obtain after applying Proposition 2 to `md_hom` according to the partitioning schema $P_{\texttt{OpenCL}}$. The computations are performed from right to left. We start in (0) by applying h to each P_{wi}-chunk. The results for chunks within the same P_{sq}-chunk are combined in the different dimensions—computations (1) and (2)—so that we obtain one result per P_{sq}-chunk. These results are combined analogously, according to (3) and (4), to one result per P_{wg} chunk. The final result is obtained by combining the results of different P_{wg}-chunks according to (5) and (6).

We optimize this decomposition schema two-fold: (1) we utilize the fact that the first d_l combine operators are concatenations, and (2) we use commutativity of the remaining combine operators, as follows.

1. We postpone the concatenations in (4) and (2) in Fig. 5, i.e., we perform the computations in the order (0) $\rightarrow$ (1) $\rightarrow$ (3) $\rightarrow$ (5) $\rightarrow$ (2) $\rightarrow$ (4) $\rightarrow$ (6) for the following reasons: (a) performing concatenations at the end allows a finer-grained parallel combining of the intermediate results in the last d_r dimensions since these combinings can then be done in parallel on each single scalar, rather than on concatenated data, and (b) since combining elements by concatenation means writing them consecutively in memory, we avoid wasting private and local memory: the

$$\texttt{md\_hom}(\ f\ ,\ (+\!\!+_1,\ldots,+\!\!+_{d_l},\circledast_1,\ldots,\circledast_{d_r})\ )(a)$$

$$= \underbrace{\mathop{+\!\!+_1}\limits_{i_1^{wg}\in[1,\texttt{NUM\_WG\_L\_1}]} \cdots \mathop{+\!\!+_{d_l}}\limits_{i_{d_l}^{wg}\in[1,\texttt{NUM\_WG\_L\_d}_\texttt{l}]}}_{(6)} \underbrace{\mathop{\circledast_1}\limits_{j_1^{wg}\in[1,\texttt{NUM\_WG\_R\_1}]} \cdots \mathop{\circledast_{d_r}}\limits_{j_{d_r}^{wg}\in[1,\texttt{NUM\_WG\_R\_d}_\texttt{r}]}}_{(5)}$$

$$\underbrace{\mathop{+\!\!+_1}\limits_{i_1^{sq}\in[1,\texttt{NUM\_SQ\_L\_1}]} \cdots \mathop{+\!\!+_{d_l}}\limits_{i_{d_l}^{sq}\in[1,\texttt{NUM\_SQ\_L\_d}_\texttt{l}]}}_{(4)} \underbrace{\mathop{\circledast_1}\limits_{j_1^{sq}\in[1,\texttt{NUM\_SQ\_R\_1}]} \cdots \mathop{\circledast_{d_r}}\limits_{j_{d_r}^{sq}\in[1,\texttt{NUM\_SQ\_R\_d}_\texttt{r}]}}_{(3)}$$

$$\underbrace{\mathop{+\!\!+_1}\limits_{i_1^{wi}\in[1,\texttt{NUM\_WI\_L\_1}]} \cdots \mathop{+\!\!+_{d_l}}\limits_{i_{d_l}^{wi}\in[1,\texttt{NUM\_WI\_L\_d}_\texttt{l}]}}_{(2)} \underbrace{\mathop{\circledast_1}\limits_{j_1^{wi}\in[1,\texttt{NUM\_WI\_R\_1}]} \cdots \mathop{\circledast_{d_r}}\limits_{j_{d_r}^{wi}\in[1,\texttt{NUM\_WI\_R\_d}_\texttt{r}]}}_{(1)}$$

$$\underbrace{\begin{aligned}\texttt{md\_hom}(\ f\ ,\ (+\!\!+_1,\ldots,+\!\!+_{d_l},\circledast_1,\ldots,\circledast_{d_r})\ )(\ a_{\text{part}}&[i_1^{wg}]\ldots[i_{d_l}^{wg}].[j_1^{wg}]\ldots[j_{d_r}^{wg}].\\ &[i_1^{sq}]\ldots[i_{d_l}^{sq}].[j_1^{sq}]\ldots[j_{d_r}^{sq}].\\ &[i_1^{wi}]\ldots[i_{d_l}^{wi}].[j_1^{wi}]\ldots[j_{d_r}^{wi}]\ )\end{aligned}}_{(0)}$$

Fig. 5 OpenCL decomposition schema for `md_hom`

results of different WIs and WGs are concatenated to the final result in the global memory. The correctness of this transformation is ensured by the following proposition which we formulate for simplicity for the 2-dimensional case—it follows from Proposition 2 and can be generalized to an arbitrary dimensionality:

Proposition 4 *Let* `md_hom`$(f\ ,\ (+\!\!+_1,\circledast))$ *be an arbitrary* 2*-dimensional instantiation of the* `md_hom` *pattern. It holds:*

$$\begin{aligned}&\mathop{+\!\!+_1}\limits_{i_1\in[1,N_1]}\ \mathop{\circledast}\limits_{i_2\in[1,N_2]}\ \mathop{+\!\!+_3}\limits_{i_3\in[1,N_3]}\ \mathop{\circledast}\limits_{i_4\in[1,N_4]}\ \mathop{+\!\!+_5}\limits_{i_5\in[1,N_5]}\ \mathop{\circledast}\limits_{i_6\in[1,N_6]}\ a[\,i_1][\,i_2].[\,i_3][\,i_4].[\,i_5][\,i_6]\\ =\ &\mathop{+\!\!+_1}\limits_{i_1\in[1,N_1]}\ \mathop{+\!\!+_3}\limits_{i_3\in[1,N_3]}\ \mathop{+\!\!+_5}\limits_{i_5\in[1,N_5]}\ \mathop{\circledast}\limits_{i_2\in[1,N_2]}\ \mathop{\circledast}\limits_{i_4\in[1,N_4]}\ \mathop{\circledast}\limits_{i_6\in[1,N_6]}\ a[\,i_1][\,i_2].[\,i_3][\,i_4].[\,i_5][\,i_6]\end{aligned}$$

for an arbitrary $a \in T[\,N_1][\,N_2].[\,N_3][\,N_4].[\,N_5][\,N_6]$.

2. After optimization 1), we perform the computations (1) → (3). Since the combine operators $\circledast_1,\ldots,\circledast_{d_r}$ are usually commutative (such as those in Table 3), we can swap the computations which has the following advantage. In the order (1) → (3), we combine the results of the different WIs in each iteration of (3). In the order (3) → (1), in each iteration, a WI combines its result with that of the iteration before—computation (3)—such that computation (1) is performed only once.

We present our optimized implementation schema for `md_hom` as an OpenCL pseudo-code, based on the schema in Fig. 5. It comprises two kernels executed consecutively: the first kernel performs the computation of (0) → (3) → (1), and the second kernel computes (5) → (2) → (4) → (6).

```
1 __kernel void md_hom( __global T [ M_1        ]...[ M_d_l        ].[ N_1        ]...[ N_d_r        ]       a,
2                       __global T'[ NUM_WG_L_1 ]...[ NUM_WG_L_d_l ].
                                   [ NUM_SQ_L_1 ]...[ NUM_SQ_L_d_l ].
                                   [ NUM_WI_L_1 ]...[ NUM_WI_L_d_l ].[ NUM_WG_R_1 ]...[ NUM_WG_R_d_r ] res_glb,
3                     )
4 {
5   // get WG/WI indices
6   int i_wg_l_1 = get_group_id( d   - 1 ) + 1; ... int i_wg_l_d_l = get_group_id( d - d_l  ) + 1;
7   int i_wg_r_1 = get_group_id( d_r - 1 ) + 1; ... int i_wg_r_d_r = get_group_id( 0        ) + 1;
8
9   int i_wi_l_1 = get_local_id( d   - 1 ) + 1; ... int i_wi_l_d_l = get_local_id( d - d_l  ) + 1;
10  int i_wi_r_1 = get_local_id( d_r - 1 ) + 1; ... int i_wi_r_d_r = get_local_id( 0        ) + 1;
11
12  // private memory for a WI's computation
13  __private T' res_prv[ d_r ];
14
15  // local memory for a WG's computation
16  __local   T' res_lcl[NUM_WI_L_1 ]...[ NUM_WI_L_d_l ][ NUM_WI_R_1 ]...[ NUM_WI_L_d_r ];
17
18  // iteration over P_sq-chunks
19  for( size_t i_sq_l_1 = 1 ; i_sq_l_1 <= NUM_SQ_L_1 ; ++i_sq_l_1 )
     .
       .
         .
20       for( size_t i_sq_l_d_l = 1 ; i_sq_l_d_l <= NUM_SQ_L_d_l ; ++i_sq_l_d_l )
21       {
22        res_prv[ 1 ] = 0;
23        for( size_t i_sq_r_1 = 1 ; i_sq_r_1 <= NUM_SQ_R_1 ; ++i_sq_r_1 ) {
            .
              .
                .
24              res_prv[ d_r ] = 0;
25              for( size_t i_sq_r_d_r = 1 ; i_sq_r_d_r <= NUM_SQ_R_d_r ; ++i_sq_r_d_r )
26              {
27                // computation of a WI
28   (3)          res_prv[ d_r ] (*)_dr = f( a_part[ i_wg_l_1 ]...[ i_wg_l_d_l ].[ i_wg_r_1 ]...[ i_wg_r_d_r ].
                                              [ i_sq_l_1 ]...[ i_sq_l_d_l ].[ i_sq_r_1 ]...[ i_sq_r_d_r ].   (0)
                                              [ i_wi_l_1 ]...[ i_wi_l_d_l ].[ i_wi_r_1 ]...[ i_wi_r_d_r ]
                                         );
29              } // end of for-loop "i_sq_r_d_r"
                .
              .
            .
30          res_prv[ 1 ] (*)_1 res_prv[ 2 ];
31        } // end of for-loop "i_sq_r_1"
32
33        // store WI's result in local memory
34        res_lcl[ i_wi_l_1 ][ i_wi_l_d_l].[ i_wi_r_1 ][ i_wi_r_d_r] = res_prv[ 1 ];
35
36        barrier( CLK_LOCAL_MEM_FENCE );
37
38        // combine the WIs' results in the last d_r dimensions
39        for( size_t dim = d_r ; dim > 0 ; --dim )
40          for( size_t stride = NUM_WI_dim / 2 ; stride > 0 ; stride /= 2 )
41          {
42            if( WI_ID_dim ≤ stride)
43   (1)                  res_lcl[ i_wi_l_1 ]...[  i_wi_l_d_l ].[ i_wi_r_1 ]...[ i_wi_r_dim          ][1]...[1]
                 (*)dim = res_lcl[ i_wi_l_1 ]...[  i_wi_l_d_l ].[ i_wi_r_1 ]...[ i_wi_r_dim + stride ][1]...[1];
44
45            barrier( CLK_LOCAL_MEM_FENCE );
46          }
47
48        // store WG's result in global memory
49        if( i_wi_r_1 == 1 && ... && i_wi_r_d_r == 1 )
50            res_glb[ i_wg_l_1 ]...[ i_wg_l_d_l ].[ i_sq_l_1 ]...[ i_sq_l_d_l ].[ i_wi_l_1 ]...[ i_wi_l_d_l ].
                                                                              [ i_wg_r_1 ]...[ i_wg_r_d_r ]
          = res_lcl[ i_wi_l_1 ]...[ i_wi_l_d_l ][1]...[1];
51
52        barrier( CLK_LOCAL_MEM_FENCE );
53
54      } // end of for-loop "i_sq_l_d_l"
55 } // end of kernel
```

Listing 1: OpenCL implementation schema (pseudocode)

Listing 1 shows the OpenCL pseudo-code of the first kernel. The WGs process in parallel different P_{wg}-chunks, and the WIs process different P_{wi}-chunks. For this, we get in line 6 – 10 the IDs of the corresponding P_{wg}-chunks and P_{wi}-chunks. We consider OpenCL dimensions in reverse order, i.e., OpenCL's dimension d is our dimension 1 and so on, for the following reason. OpenCL applications benefit from accessing consecutive memory regions by WIs with consecutive IDs in the first dimension. However, in `C`-style programming languages, elements of multi-dimensional arrays are stored in the row-major order. The reordering of IDs enables high-performance accesses to the MDA's elements in a particular dimension by WIs with consecutive IDs in that dimension. Note that while OpenCL starts counting indices from 0, we count from 1 for a better readability of our formulas.

In line 19–25, we iterate sequentially over the P_{sq}-chunks and apply in each iteration the scalar function f to the element of the corresponding P_{wi}-chunk (line 28) which represents the computation (0) in Fig. 5. Here, a_part is the $P_{\texttt{OpenCL}}$-partitioning of the input array a. We combine WIs' results for different iterations in line 23–25 (over the P_{sq} chunks in the last d_r dimensions) in private memory due to its high throughput, using the corresponding combine operators $\circledast_{d_r}, \ldots, \circledast_1$ (line 28-30) — this represents computation (3). The combined result is then stored in the local memory (line 34) where it can be accessed by all WIs of the same WG. Afterwards, the computation (1) is performed: the WIs' results are combined step-by-step in the last d_r dimensions (line 39); this is done in parallel by all WIs of a WG cooperatively (line 40–46). In each of the NUM_SQ_L_1 $* \cdots *$ NUM_SQ_L_d$_{\texttt{l}}$ iterations (line 19–20), we obtain one combined result for the WIs with an ID equal to 1 in the last d_r dimensions; these results are stored in the global memory (line 50).

The described computations produce NUM_WG_R_1 $* \ldots *$ NUM_WG_R_d$_{\texttt{r}}$ results that have to be combined according to (5) in Fig. 5. Since OpenCL does not allow synchronization between different WGs, this is performed in a second kernel that we start after the first kernel with only one WG in the last d_r dimensions. The second kernel's code is similar to the first kernel of Listing 1, but the computation of f is not performed. Since we start only one WG in the last d_r dimensions, no synchronization is required between WGs. The second kernel's result is stored consecutively in the global memory, which corresponds to the concatenations in (2), (4) and (6).

Note that the OpenCL code of the implementation schema in Listing 1 is generic in the parameters M$_{\texttt{i}}$, N$_{\texttt{j}}$ and NUM_WG_L_i, NUM_WG_R_j, NUM_WI_L_i NUM_WI_R_j for $i \in [1, d_l]$ and $j \in [1, d_r]$. The parameters M$_{\texttt{i}}$ and N$_{\texttt{j}}$ represent the input size which is prescribed by the input and thus cannot be changed. However, the other parameters are *tuning parameters*, i.e., different parameter values lead to semantically-equal but differently optimized variants of code. The tuning parameters enable customizing the parallelism granularity of our implementation for the specific characteristics of the target device and input size by choosing appropriate numbers of WGs and WIs.

To determine the most suitable tuning parameter values, we use the OpenTuner framework [2]—a popular auto-tuning tool. We apply OpenTuner to our implementation with the given input size (i.e., M_i, N_j) and the corresponding range of values for each tuning parameter. For md_gemv (matrix–vector multiplication), which we use for evaluation in Sect. 4, auto-tuning takes a time of 2 seconds for the CPU and 7 seconds for the GPU. In comparison, CLTune [16] has a longer search time for tuning CLBlast's routine xgemv_fast: on average 17 seconds for the CPU and 7 seconds for the GPU. This is arguably due to the efficacy of the OpenTuner which combines various search techniques to reduce tuning time.

Auto-tuning causes an additional one-time overhead for each combination of device and input size, but it improves kernels' performance. It is especially beneficial to auto-tune if an application calls a kernel multiple times for the same device and input size. For example, in the application field of deep learning, the kernel for computing GEMV is called many times—10^6 and more [12]—on the same input size, making auto-tuning time negligible.

To achieve the input type compatibility between our `md_hom` and particular applications as presented in the literature, we use views (see Sect. 2) implemented as `C` preprocessor macros. For example, our `md_gemv` (matrix–vector multiplication) operates on a single MDA while the popular implementations of matrix–vector multiplication usually operate on a matrix and a vector. The compatibility is ensured by using the view `pair_mv` from Table 1. Let `mat` be an arbitrary $M \times N$ input matrix, `vec` an arbitrary input vector of size N and $a \in \texttt{float}^2[M][N]$ the view's output MDA of pairs of floating point numbers, then we implement `pair_mv` using the following macro:

```
#define a(i, j, pi) ((pi == 1) ? mat[(i - 1) * N + j - 1]
                                : vec[j - 1])
```

4 Experimental Results

We use the example of MDH `md_gemv` (matrix–vector multiplication) for evaluating the performance of the auto-tuned OpenCL code that is implemented according to our generic schema in Listing 1. We compare our implementation with the currently fastest (as shown in [21,22]), hand-tuned vendor implementations of the BLAS routine `gemv`, each optimized for a particular device architecture:

1. the `cblas_sgemv` function of the Intel MKL library [11] on an Intel Xeon E5-1620 CPU;
2. the `cublasSgemv` function of the NVIDIA cuBLAS library [18] on an NVIDIA Tesla K20c GPU.

In addition, we compare our implementation on both architectures — CPU and GPU— to the routine `xgemv_fast` of the auto-tunable state-of-the-art OpenCL BLAS library CLBlast [3]; it has competitive performance compared to vendor implementations (e.g., cuBLAS) on different device architectures and for different input sizes [22]. We use CLBlast's auto-tuner CLTune to tune `xgemv_fast` for the target device and input size.

In all experiments, we measure the pure computation time, using the OpenCL profiling API to take the runtime of our OpenCL application. For the reference implementations by Intel and NVIDIA, we use the C++ Chrono library and NVIDIA `nvprof` profiling tool, correspondingly; for CLBlast we use its profiling API. In each experiment, we report the median time of 100 runs.

Figure 6 shows our experimental results on the CPU (left) and GPU (right) for three different sizes of the input matrix: $2^{14} \times 2^{14}$ (square matrix), $2^{10} \times 2^{18}$ (wide matrix) and $2^{18} \times 2^{10}$ (tall matrix). For each input size, we show the runtime of our OpenCL `md_gemv` implementation (white), the vendor implementations MKL for the CPU (light grey) and cuBLAS for the GPU (dark grey) respectively, and the runtime for CLBLast (black). We observe a slightly better performance of our implementation

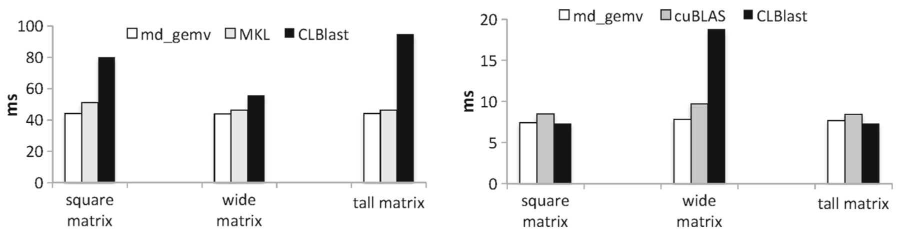


Fig. 6 Runtime of matrix–vector multiplication (GEMV) on Intel CPU (*left*) and NVIDIA GPU (*right*) for different matrix shapes

compared to the hand-tuned code by Intel and NVIDIA which are developed to yield a good performance on average for various input sizes, while we strive for highest performance for each individual input size.

In comparison to CLBlast, we achieve in most cases a significantly better performance. This is due to the fact that we parallelize in both dimensions of the input MDA while CLBlast parallelizes in only one dimension.

5 Related Work

Our work has been inspired by, and it enhances the previous research in the area. In [4], *abide-tree homomorphisms* are studied on two-dimensional arrays. In comparison to them, we extend the homomorphism theory for arrays to arbitrary dimensionality, which enables a more fine-grained parallelism as required by modern parallel devices. Moreover, we develop an executable implementation schema in OpenCL, rather than a composition of functional building blocks such as `map` and `reduce` that would need a further effort on efficient parallelization. The approaches [9,13] study computations on arbitrary dimensional MDAs with focus on CPU architectures while we target all OpenCL-capable devices, i.e., also GPUs.

In [21], starting from a high-level pattern expression, a search space of different, semantically-equal implementations is generated and explored for a specific hardware by using a search engine. We follow an alternative approach in order to avoid large search spaces and, consequently, time-intensive search space exploration: we generate a parametrized implementation which is then optimized for specific devices and input sizes by tuning a small set of parameters. Moreover, we provide an arguably simpler high-level abstraction: e.g., we express matrix multiplication by using the `md_hom` pattern and a corresponding view, while the related work uses four different patterns in combination with lambda expressions, pattern composition and nesting [22].

There are several parallel pattern libraries such as SkelCL [20], Muesli [6], SkePU [5] and FastFlow [1] for simplifying parallel programming: they generate low-level OpenCL or CUDA code from high-level expressions. While we consider optimizations for different devices, SkelCL and Muesli are optimized towards particular architectures. The optimizations in SkePU are, to the best of our knowledge, not applicable for their `MapArray` pattern which is required, for example, to express matrix–vector multiplication. FastFlow provides a general parallel pattern—*Loop-of-*

Stencil-Reduce—which is optimized for stencil computations that are performed in a loop; however, linear algebra routines such as matrix–vector multiplication are not considered.

While we express the three BLAS routines `dot`, `gemv` and `gemm` by using solely the `md_hom` pattern and a corresponding view, the aforementioned related work requires for that various specific patterns. For example, in [5], pattern `MapReduce`$(f, \oplus)$—an optimized implementation of `map`(f) composed with `reduce`$(\oplus)$—is required to express the routine `dot`, and pattern `MapArray`—a variant of the `map` pattern which produces a result from two input vectors—is used to express `gemv`. In [20], `gemm` is expressed by using `allPairs`$(\oplus)$—a pattern that applies the binary function $\oplus$ to each combination of rows and columns of two input matrices. For this, the composition of two further patterns—`zip`$(*)$ and `reduce`$(+)$—is nested in `allPairs`. All these patterns in the related work have to be implemented and optimized specifically.

Sorenson [19] and Xu et al. [23] present CUDA implementations for matrix–vector multiplication that are optimized by using auto-tuning. In contrast to our work, both implementations are restricted to NVIDIA GPUs.

6 Conclusion

In this paper, we present a multi-dimensional extension of the homomorphism concept and propose a new parallel pattern `md_hom` based on this extension. We show that `md_hom` is convenient for expressing a broad class of applications, and we develop a correct-by-construction OpenCL implementation schema for `md_hom` that targets various modern multi- and many-core devices. The schema is optimized for the target hardware and input size by using auto-tuning. To evaluate the runtime performance of the OpenCL code generated according to our schema, we implement the example of general matrix–vector multiplication (GEMV)—a BLAS routine widely used, e.g., for deep learning. We achieve a competitive or better performance than hand-tuned, platform-specific BLAS libraries and the auto-tunable OpenCL BLAS library CLBlast on two different parallel architectures—Intel CPU and NVIDIA GPU.

References

1. Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Pezzi, G.P., Torquati, M.: The loop-of-stencil-reduce paradigm. In: Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 3, pp. 172–177. IEEE (2015)
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: OpenTuner: an extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 303–316. ACM (2014)
3. Cedric Nugteren: CLBlast. https://github.com/CNugteren/CLBlast (2017)
4. Emoto, K., Matsuzaki, K., Hu, Z., Takeichi, M.: Surrounding theorem: developing parallel programs for matrix-convolutions. In: Euro-Par 2006 Parallel Processing, pp. 605–614. Springer (2006)
5. Enmyren, J., Kessler, C.W.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, pp. 5–14. ACM (2010)
6. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. **7**(2), 129–138 (2012)
7. Gorlatch, S.: Extracting and implementing list homomorphisms in parallel program development. Sci. Comput. Program. **33**(1), 1–27 (1999)

8. Gorlatch, S., Cole, M.: Parallel skeletons. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1417–1422. Springer (2011)
9. Grelck, C., Scholz, S.B.: SAC—a functional array language for efficient multi-threaded execution. Int. J. Parallel Program. **34**(4), 383–427 (2006)
10. Intel: OpenCL Optimization Guide (2011)
11. Intel: Intel MKL. https://software.intel.com/en-us/intel-mkl (2016)
12. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
13. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, shape-polymorphic, parallel arrays in Haskell. In: ACM Sigplan Notices, vol. 45, pp. 261–272. ACM (2010)
14. Khronos OpenCL Working Group: The OpenCL Specification. https://www.khronos.org/opencl/ (2017)
15. Netlib: BLAS. http://www.netlib.org/blas/ (2016)
16. Nugteren, C., Codreanu, V.: CLTune: a generic auto-tuner for OpenCL kernels. In: Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), pp. 195–202. IEEE (2015)
17. NVIDIA: NVIDIA OpenCL Best Practices Guide (2015)
18. NVIDIA: NVIDIA cuBLAS. https://developer.nvidia.com/cublas (2016)
19. Sørensen, H.H.B.: High-performance matrix-vector multiplication on the GPU. In: Alexander, M. (ed.) Euro-Par 2011: Parallel Processing Workshops, pp. 377–386. Springer (2011)
20. Steuwer, M., Gorlatch, S.: SkelCL: a high-level extension of OpenCL for multi-GPU systems. J. Supercomput. **69**(1), 25–33 (2014)
21. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance Opencl code. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 205–217. ACM (2015)
22. Steuwer, M., Remmelg, T., Dubach, C.: Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, p. 15. ACM (2016)
23. Xu, W., Liu, Z., Wu, J., Ye, X., Jiao, S., Wang, D., Song, F., Fan, D.: Auto-tuning GEMV on many-core GPU. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS), pp. 30–36. IEEE (2012)