Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms

Ari Rasch
University of Muenster, Germany
a.rasch@wwu.de

Richard Schulze
University of Muenster, Germany
r.schulze@wwu.de

Sergei Gorlatch
University of Muenster, Germany
gorlatch@wwu.de

***Abstract*—We address a key challenge in programming high-performance applications – achieving *portable performance*, i.e., the same source code achieves a consistent, high level of performance over the variety of modern parallel processors, including multi-core CPU and many-core Graphics Processing Unit (GPU), and over the variety of input sizes. Our approach relies on the algebraic formalism of *Multi-Dimensional Homomorphisms (MDH)*, which enables expressing data-parallel computations uniformly via a higher-order function (a.k.a. parallel pattern). For MDHs, we develop a novel code generation approach based on a generic OpenCL implementation. Our implementation efficiently exploits the OpenCL's abstract platform and memory model, generically for arbitrary MDH functions, by incorporating a parameterized parallelization and tiling strategy – on both layers of the OpenCL's two models and in all dimensions of the multi-dimensional input. We achieve performance portability for MDHs by auto-tuning the parameters of our two strategies, thereby enabling fully automatically optimizing our code for any combination of an MDH function, target device, and input size. We demonstrate for computations from four popular domains – dense linear algebra (BLAS), stencil computations, data mining, and tensor contractions – how we express them in the MDH formalism, and we experimentally show that our automatically generated and auto-tuned code for them achieves competitive and often significantly better performance than several state-of-practice approaches on both Intel multi-core CPU and NVIDIA many-core GPU – speedups of up to $5\times$ over the state-of-the-art performance-portable approaches, and competitive or even better performance as compared to hand-optimized approaches such as Intel MKL and NVIDIA cuBLAS on real-world input data as used in deep learning.**

I. Motivation

A key challenge in programming high-performance applications is to achieve portable performance over the variety of parallel devices and input sizes.

Popular approaches to parallel programming are often either restricted to the hardware of a particular vendor (like CUDA for NVIDIA [1]) or, even if they provide code portability (like OpenCL [2]), the portability of performance is usually not available over the variety of modern parallel processors, because devices may differ significantly in their characteristics (e.g., the number of cores, cache sizes, etc.). Also devices from different vendors (e.g., NVIDIA GPU vs. AMD GPU) pose different or even contradicting requirements on the code for achieving the full performance potential of the corresponding device. Performance differs also across input sizes [3]. For example, a high-performance implementation of *GEneral Matrix-Matrix multiplication (GEMM)* targeting big, square input matrices [4] differs significantly from a GEMM implementation optimized for small, irregularly shaped matrices, e.g., as currently occurring in deep learning [3].

In this paper, we develop a novel approach to code generation that enables high, portable performance for *Multi-dimensional Homomorphisms (MDH)* [5] – a class of formally-defined functions that cover important data-parallel computations. Our approach relies on the uniform algebraic representation of MDHs – via the `md_hom` higher-order function (a.k.a. *parallel pattern* [6]) – which allows to efficiently exploit the OpenCL's abstract device models [2]: 1) the *platform model* by incorporating in our generated code a multi-layered, multi-dimensional parallelization strategy over the models' hierarchy of cores, and 2) the *memory model* by incorporating a multi-layered, multi-dimensional tiling strategy over the models' memory hierarchy. Our two strategies are parameterized in the performance-critical parameters of the two OpenCL models; for example, in the number of threads and size of tiles – on both core/memory layers of the models, and in all dimensions of the MDHs' multi-dimensional input. By targeting the OpenCL's abstract device models (rather than a particular device, e.g., an Intel CPU or NVIDIA GPU) and by parameterizing our parallelization and tiling strategies in performance-critical parameters, we enable performance portability: we can fully automatically optimize our generated code – for any combination of an MDH function, target device, and input size – by automatically choosing (auto-tuning) optimized values of parameters.

We demonstrate the efficiency of our automatically generated and auto-tuned code by experiments with computations from four popular application domains: 1) dense linear algebra (BLAS), 2) stencil computations, 3) data mining, and 4) tensor contractions. For each computation, we show how it can be expressed in the MDH formalism, and our experimental results demonstrate competitive and often even significantly better performance of our approach as compared to several state-of-practice approaches on both CPU and GPU: we achieve speedups of up to $5\times$ over the state-of-the-art performance-portable approaches, and competitive or even better performance as compared to hand-optimized approaches (such as Intel MKL [7] and NVIDIA cuBLAS [8]) on important real-world input sizes, e.g., as used in the popular deep learning framework Caffe [9].

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00035

II. Related Work

Achieving high, portable performance on multi- and many-core devices has attracted much attention recently. Approaches such as Tiramisu [10], AlphaZ [11], CHiLL [12], URUK [13], Halide [14], Transformation Recipes [15], and POET [16] achieve high performance on different devices, but they involve the user into optimization, thus requiring from her significant expert knowledge. Futhark [17], PENCIL [18], Pluto [19], Polly [20], Tensor Comprehensions [21], COGENT [22], and PolyMage [23] automatically generate optimized parallel code; however, they often miss high performance, e.g., because of inefficiently exploiting devices' fast memory resources. Moreover, the related work is often tied to specific application classes; for example, Halide only to image processing and COGENT only to tensor contractions. Similarly, approaches [24]–[27] are applicable to only linear algebra routines, SPIRAL [28], [29] only to signal processing, TVM [30] and Boda [31] to machine learning algorithms, and TACO [32] to sparse tensor computations. Domain-specific and target-specific approaches, including for (iterative) stencils [23], [33]–[40] or for small-scale linear algebra routines [41], may achieve higher performance than our proposed framework, by developing optimizations that are specialized to a specific computation pattern or a single architecture, but usually at the expense of performance portability.

Approaches such as REPA [42], OBSIDIAN [43], Operator Language [44], and HTA [45] generate efficient code (e.g., in CUDA) by relying on functional high-level abstractions which are similar to our MDH formalism. However, the problem of performance portability is not addressed.

Our work is inspired by [5], where the algebraic formalism of Multi-Dimensional Homomorphisms (MDH) is introduced, and a first, auto-tunable OpenCL implementation for MDHs is provided. Compared to [5], our code generation approach is substantially extended in order to improve performance and portability: while [5] optimizes only for OpenCL's platform model, our generated code is also optimized for OpenCL's memory model by incorporating a parameterized tiling strategy, leading to significantly higher performance (as we demonstrate experimentally in Section VII). Furthermore, we exploit the OpenCL's platform model more efficiently than [5]: we process in parallel a flexible number of arbitrarily-sized tiles, such that we can choose the number of threads and size of tiles independently of each other, allowing to auto-tune our implementation for both OpenCL models individually (as discussed in Section V), as usually required for high performance. In addition, we significantly extend the experimental evaluation of [5]: we present results for BLAS routine GEMM (GEneral Matrix-Matrix multiplication) which is significantly more complex to optimize than GEMV in [5]. Besides BLAS, we demonstrate for computations from three further important domains – stencil computations, data mining, and tensor contractions – that they can be expressed in the MDH formalism (in Section IV), and we present experimental results for them (in Section VII).

Lift [46] is closely related to our work – it also aims at automatically generating optimized OpenCL code targeting different devices and input sizes; it has proven to provide high, portable performance for important computations, e.g., linear algebra routines [47] and stencil computations [48]. Lift's approach is based on a vast search space of semantically-equal, differently-optimized OpenCL programs, where space's efficient exploration relies on hand-pruning by the user. In contrast to Lift, we enable fully automatically optimizing our code, without incorporating the user, by taking a completely different approach to optimization: we generate a generic OpenCL program that is parameterized in performance-critical parameters, thereby enabling optimizing our code via classical auto-tuning. Moreover, we demonstrate experimentally in Section VII that our approach provides higher performance than Lift – for different applications, devices, and input sizes – thus contributing to a more efficient performance portability.

III. Multi-Dimensional Homomorphisms and the md_hom Parallel Pattern

In this section, we briefly recapitulate the definitions of MDHs and the md_hom higher-order function (a.k.a. parallel pattern) which is used to uniformly express MDHs [5].

Definition 1 (Multi-Dimensional Homomorphism)**.** Let T and T' be two arbitrary data types. A function $h : T[\,N_1\,]\ldots[\,N_d\,] \rightarrow T'$ on d-dimensional arrays of size $N_1 \times \ldots \times N_d$ and with elements in T is called a *Multi-Dimensional Homomorphism (MDH)* iff there exist *combine operators* $\circledast_1, \ldots, \circledast_d : T' \times T' \rightarrow T'$, such that for each integer $k \in [1, d]$ and arbitrary, concatenated input array $a +\!\!+_k b$ in dimension k, the homomorphic property is satisfied:

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

In words: the value of h on a concatenated array in dimension k can be computed by applying h to array's parts a and b, and then combining the results by combine operator $\circledast_k$.

MDHs have a uniform representation via the md_hom parallel pattern [5], as follows. Every MDH h is uniquely determined by its combine operators $\circledast_1, \ldots, \circledast_d$ and its behavior f on scalar values (i.e., $f(\,a[0]\ldots[0]\,) = h(a)$ for every $a \in T[1]\ldots[1]$). This enables expressing h using the md_hom higher-order function (a.k.a. pattern) which takes these functions as parameters:

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

IV. Data-Parallel Computations Expressed in the MDH Formalism

A. Linear Algebra Routines (BLAS)

Basic Linear Algebra Subprograms (BLAS) are one of the most essential application classes in high-performance computing; over decades until now, elaborating high-performance implementations of BLAS has been the focus of many industrial (e.g., [49], [50]) and research projects (e.g, [3], [51]–[55]).

We demonstrate the usage of md_hom using the example of the most popular BLAS routines: DOT (dot product), GEMV (GEneral Matrix-Vector multiplication), and GEMM (GEneral Matrix-Matrix multiplication):

$$\begin{aligned} \texttt{DOT} &= \texttt{md\_hom}(\ *,\ (+)\ ) \circ \texttt{view\_dot} \\ \texttt{GEMV} &= \texttt{md\_hom}(\ *,\ (+\!\!+,\ +)\ ) \circ \texttt{view\_gemv} \\ \texttt{GEMM} &= \texttt{md\_hom}(\ *,\ (+\!\!+,\ +\!\!+,\ +)\ ) \circ \texttt{view\_gemm} \end{aligned}$$

Here, $\circ$ denotes function composition, i.e., applying functions after each other from right to left.

For expressing DOT via md_hom, we first fuse its domain-specific input – two vectors $v, w \in T[N]$ of size N and type T (e.g., $T =$ float or double) – straightforwardly to a 1-dimensional array comprising pairs of type T^2. For this, we use a so-called *(input) view* function [5] (discussed in detail in Section V-C) which the MDH formalism provides to uniformly prepare a domain-specific input for md_hom. For function DOT, its view is defined as: $\texttt{view\_dot}(\ v, w\ ) = (\ (v_1, w_1), \ldots, (v_n, w_n)\ )$. MDH's can optionally also have an *(output) view*, e.g., to store the result of an MDH to different output buffers (we do not discuss output views in this paper). In our generated code (presented in the next section), we implement views as preprocessor macros that performs straightforward index computations, rather than costly memory operations.

After fusing DOT's input vectors v and w via function view_dot, we compute DOT expressed as an instance of the md_hom parallel pattern. For this, we use as the pattern's arguments the combine operator of DOT, which is summation (denoted by $+$), and the function that expresses DOT's behavior on scalar values – the scalar values are pairs (e.g., consisting of two floats or doubles), and DOT's behavior on these pairs is multiplication (denoted by $*$).

The md_hom expression for GEMV is analogous. We compute multiple dot products – one per row of the input matrix A – and combine the obtained results by concatenation. Thus, for expressing GEMV via md_hom, we take the md_hom expression of DOT and use the concatenation operator $+\!\!+$ as a further combine operator.

Similarly, for computing GEMM, we have to perform multiple times GEMV – one GEMV computation per column of GEMM's second input matrix – and combine the results by concatenation. Consequently, we express GEMM conveniently via md_hom by passing to the GEMV's md_hom expression only concatenation as a further combine operator.

Note that to keep the discussion clear, we use slightly simplified versions of BLAS routines that focus on the actual computations. For example, in the original definition of GEMM, the output matrix can be scaled by a factor and optionally be transposed.

B. Stencil Computations

Stencil computations are very relevant in high-performance computing; they are used in important application domains, such as image processing and deep learning. Stencil computations apply a user-defined *stencil function* f to equally-sized chunks of a multi-dimensional array.

A stencil computation is expressed as MDH as follows:

$$S_f = \texttt{md\_hom}(\ f,\ (+\!\!+_1, \ldots, +\!\!+_d)\ ) \circ \texttt{view\_stencil}_{<P>}$$

Here, $\texttt{view\_stencil}_{<P>}$ is the stencils' view function; it takes a d-dimensional array A as input, and it yields an array of the same dimensionality. The view's output array comprises at position $(i_1, \ldots, i_d)$ neighbors $NB_{(i_1,\ldots,i_d)}$ of element $A[\,i_1\,]\ldots[\,i_d\,]$ in the input array; the neighbors are specified by the user-defined set $P = \{\ (c_1^k, \ldots, c_d^k)\ \mid\ 1 \leq k \leq K\ \}$ which contains the coordinates (i.e., positive integers) of the neighbors – K many – relative to position $(i_1, \ldots, i_d)$, i.e., $NB_{(i_1,\ldots,i_d)} = \{\ (A[i_1 + c_1^k], \ldots, A[i_d + c_d^k])\ \mid\ 1 \leq k \leq K\ \}$.

The stencil's md_hom expression applies f – the user-defined stencil function – to each of the neighbor sets within the view's output array, and it combines the results in the different dimensions using concatenation $+\!\!+$.

C. Data Mining

As an example in the area of data mining, we consider Probabilistic Record Linkage (PRL) [56] – it identifies data records (e.g. person data) referring to the same real-world entity, e.g., Mary Smith vs. Marie Smith. PRL is increasingly used in epidemiology centers, intelligence agencies, and universities. PRL is expressed as MDH as follows:

$$\texttt{PRL} = \texttt{md\_hom}(\ \texttt{weight},\ (+\!\!+, \texttt{max})\ ) \circ \texttt{cart}$$

Function cart takes two 1-dimensional arrays of records (e.g., person data: name, surname, birthday, etc.) in which the duplicate records have to be identified; it yields a 2-dimensional array of pairs, which comprises all possible combinations of records (a.k.a. cartesian product) within the two input arrays. Function weight applies the so-called PRL-specific *weight function* [56] to the pairs, and the obtained results are combined by function max in the second dimension (i.e., we take the maximum weight for each entry) and by concatenation in the first dimension.

D. Tensor Contractions

Tensor contractions are the basic building blocks of deep learning computations. An example of a tensor contraction, on two 4D input tensors A and B (tensors can be interpreted as multi-dimensional arrays), is as follows:

$$C[a, b, c, d] = \sum_{e,f} A[a, e, b, f] * B[d, f, c, e]$$

This contraction example can be expressed as MDH:

$$\texttt{TC} = \texttt{md\_hom}(\ \star,\ (+\!\!+, +\!\!+, +\!\!+, +\!\!+, +, +)\ ) \circ \texttt{view\_tc}$$

Here, function $\texttt{view\_tc}(\ A,\ B\ )(\ a,\ b,\ c,\ d,\ e,\ f\ ) = (\ A[\,a,\ e,\ b,\ f\,],\ B[\,d,\ f,\ c,\ e\,])$ prepares the inputs A and B as a 6-dimensional array of pairs, and the md_hom expression applies multiplication to each pair and combines the obtained results by concatenation in the first four dimensions a, b, c, d and by addition in the two remaining dimensions e and f.

V. OpenCL Code Generation for MDHs

We present our code generation approach for MDHs, which is based on a generic OpenCL implementation. An important advantage of our approach is that we generate our code as targeted to the OpenCL's abstract platform and memory model (rather than to a particular device, e.g., an Intel CPU or NVIDIA GPU) by incorporating a parameterized parallelization and tiling strategy in our code for these abstract models; thereby, we enable a subsequent auto-tuning of our generated code for arbitrary combinations of an MDH function, target device, and input size.

In the following, before discussing the pseudocode of our implementation in Section V-C, we present in Section V-A an illustrative example of our generated OpenCL code, and we discuss in Section V-B our implementation's auto-tunable parameters.

A. Illustrative Example of our Generated OpenCL Code

Figure 1 depicts the schema of our OpenCL implementation for the illustrative example of a 2-dimensional MDH:

$$\texttt{md\_hom}(\ f,\ (\circledast_1, \circledast_2)\ )$$

In this example, the input – an 8×8 input matrix – is processed in four consecutive steps ①→②→③→④, from left to right; in each step, we show memory's content on all OpenCL layers – *global*, *local*, *private* – from top to bottom.

In OpenCL, the input is processed in parallel by threads – a.k.a. *Work-Items* (WI) in OpenCL terminology – which are arranged in thread groups – a.k.a. *Work-Groups* (WG). The optimal numbers of WGs and WIs depend on the concrete application, target device, and input size, and as such, these numbers are very critical for performance. In our parallelization strategy, the number of WGs and WIs are flexible parameters so that we can automatically determine optimized values for them using classical auto-tuning (discussed in Section VI). For demonstration, we use in the figure $2*2$ WGs (differently colored in the top and mid parts of the figure, and connected by dashed lines) each consisting of $2 * 2$ WIs (differently colored in the mid and bottom parts) to process an 8×8 input matrix. The input matrix resides in global memory in OpenCL; we process it in parallel by all WIs of the WGs in 4 consecutive steps ①-④, from left to right. A WG operates on local layer; in each step, it processes, according to the $2*2$ WIs per WG, a 2×2-sized part of its *local tile* – a chunk of the input array which we copy from slow global memory (top part of the figure) to fast local memory (mid part of the figure). Analogously, a WI operates on private layer and processes in each individual step a part of its *private tile* – a chunk of the local tile (mid part of the figure) which we copy to even faster private memory (bottom part of the figure). In this example, for illustration purpose, we use a local tile size of 4×4 (mid part of the figure), and we use a private tile size of 2×2 (bottom part). Our tiling strategy (presented in Section V-C) uses parameterized sizes of tiles, so that the sizes can be auto-tuned, because optimal values of them depend on the physical local and private memory resources of the specific target device.

A WI processes its 2×2-sized private tile (bottom part of the figure) by applying the MDH's scalar function f to each of tile's elements and combining the results in the MDH's two dimensions using the MDH's combine operators $\circledast_1$ and $\circledast_2$.

Note that in each step ①-④, we access with WGs and WIs always consecutive memory regions – on all three memory layers (global, local, private) – thereby, we efficiently exploit locality which is a fundamental hardware principle for high performance. For example, our access pattern enables OpenCL to automatically SIMD-parallelize our code for CPU-like architectures [57], and it enables coalesced global memory accesses and avoiding local memory bank conflicts on GPUs [58] – these optimizations are very important for high performance.

Note further that not in all devices, local memory is faster than global memory and private memory faster than local memory (e.g., in CPUs). For such cases, we allow disabling caching in local/private memory, which we discuss in the next section.

B. Tuning Parameters

Table I lists the performance-critical parameters (a.k.a. *tuning parameters*) of our OpenCL implementation (presented in Section V-C), for which we automatically determine optimized values via auto-tuning (in Section VI) – different parameter values lead to semantically-equal but differently-optimized variants of our code.

With parameters 1-2 in the table, we optimize our code for OpenCL's platform model, with parameters 3-6 for the memory model, and parameters 7-10 are for both models.

No.	Name	Description
1	$\texttt{NUM\_WG}^{\texttt{<i>}}$	number of Work-Groups
2	$\texttt{NUM\_WI}^{\texttt{<i>}}$	number of Work-Items
3	$\texttt{LT\_SIZE}^{\texttt{<i>}}$	local tile size
4	$\texttt{PT\_SIZE}^{\texttt{<i>}}$	private tile size
5	$\texttt{MEM\_INP}^{\texttt{<LYR,b,i>}}$	memory regions for caching input
6	$\texttt{MEM\_RES}^{\texttt{<LYR,b,i>}}$	memory regions for comp. results
7	$\sigma^{\texttt{<LYR>}}_{\texttt{arr}\rightarrow\texttt{ocl}}$	mapping array to OpenCL dimensions
8	$\sigma^{\texttt{<LYR,b>}}_{\texttt{buff-do}}$	buffer dimension order
9	$\sigma^{\texttt{<LYR>}}_{\texttt{mdh-do}}$	MDH dimension order
10	CMB_RES	layer to combine results on

TABLE I: Tuning parameters our OpenCL code for MDHs.

a) Parallelization Parameters: We use parameters 1 and 2 to arbitrarily set the granularity of parallelism in our implementation by flexibly choosing the numbers of work-groups ($\texttt{NUM\_WG}^{\texttt{<i>}}$) and work-items ($\texttt{NUM\_WI}^{\texttt{<i>}}$) – we parallelize our computations in all dimensions of the target

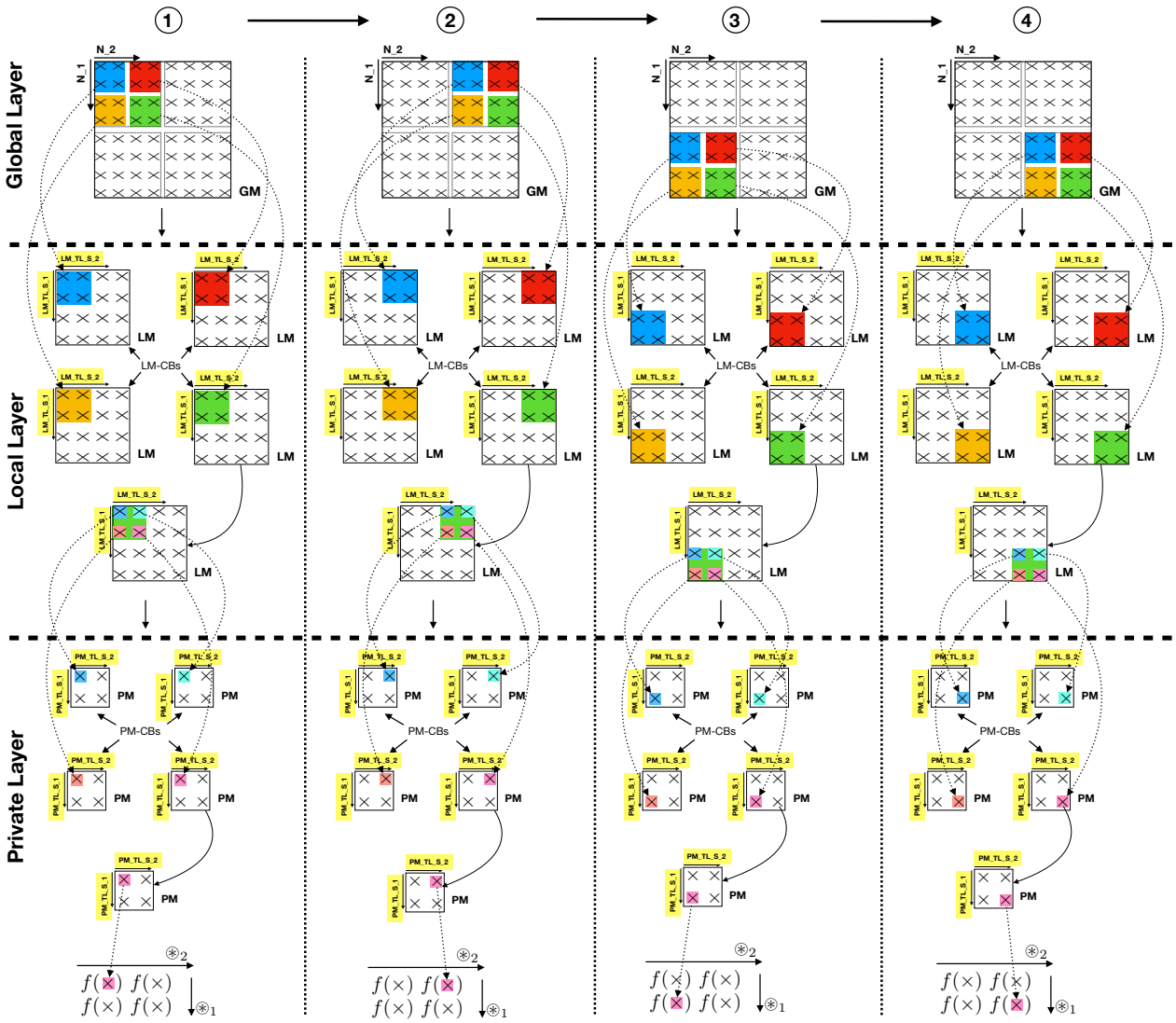


Fig. 1: Illustrative schema of our generated OpenCL code for MDHs.

d-dimensional MDH (indicated by superscript $\mathtt{i} \in \{1, \ldots, d\}$ in tuning parameters' names). By flexibly choosing the numbers of WGs and WIs, we efficiently exploit the OpenCL's two layers of parallelism [2]: OpenCL schedules WGs to *compute units*, whose number can differ significantly over devices, because compute units usually correspond to the device's physical cores; WIs are scheduled to *processing elements*, correspondingly. Note that we enable a fine-granular parallelization also for irregularly shaped inputs by arbitrarily setting the number of WIs and WGs per particular dimension, e.g., when some dimensions of the input array are large and others very small (e.g., as in deep learning applications).

b) Tiling Parameters: We use parameters 3 and 4 in Table I to flexibly set the sizes of local and private tiles in our implementation, i.e., chunks of the input that are processed by a WG or WI, correspondingly. We tile our input on both layers and in all d dimensions of the multi-dimensional input. In contrast, the related work often has restrictions in the tile size, e.g., a fixed size of 1 in some dimensions, leading to poor performance, as we demonstrate experimentally in Section VII. Note that we have no correlation between our parallelization and tiling parameters (1-2 and 3-4): we allow WGs to process an arbitrary number of local tiles and WIs to process an arbitrary number of private tiles; this increased flexibility enables optimizing our code individually for each of the two OpenCL's models.

We use parameter 5 to flexibly set the parts of the local and private tiles that we aim to explicitly cache in local or private memory. We define the parameter individually for each i) layer $\mathtt{LYR} \in \{\mathtt{G}, \mathtt{L}, \mathtt{P}\}$ – global (G) where the local tiles are processed, local (L) where private tiles are processed, and private (P) where the individual elements

within a private tile are processed; ii) input buffer b of the (input) view; iii) dimension $i \in \{1, \ldots, d\}$ of the MDH. For example, we can use this parameter to explicitly cache local tiles in local memory by setting $\texttt{MEM\_INP}^{\texttt{<L,b,i>}} = \texttt{\_\_local}$ (local memory) for all dimensions $i \in \{1, \ldots, d\}$ and view's input buffers b. Note that we can use this parameter also to avoid explicitly caching the input, by setting $\texttt{MEM\_INP}^{\texttt{<LYR,b,i>}} = \texttt{\_\_global}$ (global memory) for all layers, buffers, and dimensions, which is beneficial for devices with automatically managed caches (e.g., CPUs) – note that when caching in local/private memory is disabled, we still tile the input according to parameters 3 and 4 (a.k.a. *cache blocking* [59]).

Parameter 6 specifies the memory region in which computed results of WGs and WIs are stored. The parameter specifies the memory region individually for: i) OpenCL's different layers $\texttt{LYR} \in \{\texttt{G}, \texttt{L}, \texttt{P}\}$, ii) the MDH's different dimensions $i \in \{1, \ldots, d\}$, and iii) buffers b of the output view. For example, the parameter enables avoiding a too heavy private memory usage (which causes, e.g., register spilling [58]), by storing the results of processed private tiles in local memory, rather than in private memory.

c) Parallelization/Tiling Parameters: Parameter 7 is a permutation function on dimensions $\{1, \ldots, d\}$ and defined individually for OpenCL's local (L) and private (P) layer $\texttt{LYR} \in \{\texttt{L}, \texttt{P}\}$. We use it to flexibly set the mapping between the dimensions of the multi-dimensional input array and WGs/WIs; for example, $\sigma^{\texttt{<L>}}_{\texttt{arr}\rightarrow\texttt{ocl}}(0) = 1$ means that consecutive array elements in dimension 0 are processed by WGs with consecutive OpenCL id in dimension 1. Using this parameter, we enable locality-aware memory access patterns for WGs and WIs for arbitrary data layouts of the input array (e.g., transposed/non-transposed), because locality is exploited in OpenCL when WGs and WIs with consecutive OpenCL ids in first dimension access consecutive memory regions. Locality is a fundamental hardware principle and as such important for high performance.

Parameter 8 further contributes to better exploiting locality; it is again a permutation function on dimensions, and it is set individually for OpenCL's local (L) and private (P) layer, $\texttt{LYR} \in \{\texttt{L}, \texttt{P}\}$, and buffer b of the input view. The parameter sets the order of dimensions of the local and private memory buffers that we use for explicitly caching tiles of the input array. For example, the parameter enables caching a transposed input buffer as non-transposed in local memory (by switching the order of local buffer's dimensions), which usually contributes to better exploiting locality.

Parameter 9 sets the order in which we process the MDH's d different dimensions on OpenCL's global (G), local (L), and private (P) layer, $\texttt{LYR} \in \{\texttt{G}, \texttt{L}, \texttt{P}\}$. For example, using this parameter, we enable postponing processing dimensions whose combine operator is concatenation. This often improves performance, because other combine operators (e.g., addition) can be applied to the individual elements of the (non-concatenated) data in parallel. Furthermore, postponing concatenation also allows concatenating results directly in global memory, which saves local and private memory.

Parameter 10 sets if WIs' results should be combined on global, local, or private layer. For example, combining the WIs' results early, e.g., after each WI processed a private tile, reduces consumption of fast memory resources, but requires to combine the WIs' results very often (once per private tile). In contrast, combining WI's results late, e.g., after all local tiles are processed, requires combining WIs' results only once; however, at the cost of an increased usage of fast memory resources.

C. OpenCL Implementation for MDHs

a) Implementation of Views: Before presenting our generated OpenCL code for MDHs, we discuss *views* which prepare a domain-specific input (or output) for MDHs, as discussed in Section IV. We implement views by performing straightforward index computations, rather than costly memory operations.

The user can conveniently define view functions in our approach. For example, the following view expression defines function view_gemm for BLAS routine GEMM (see Section IV):

```
view( A,B )( i,j,k )( A(i,k),B(k,j) )
```

Our code generator automatically generates from this expression an OpenCL preprocessor macro of the following form:

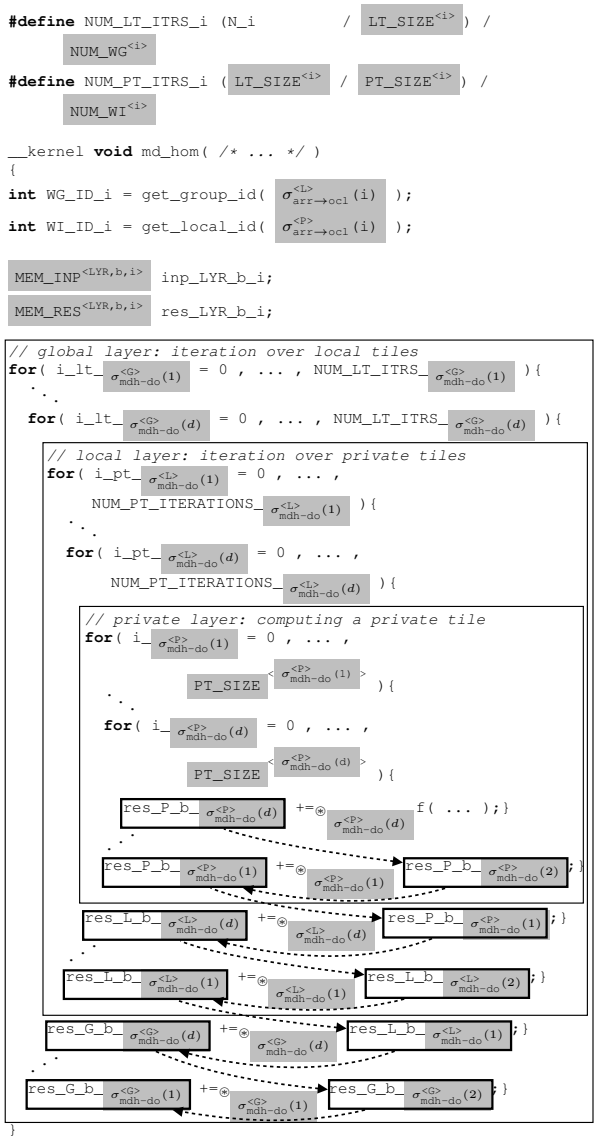
```
#define view( A,B , i,j,k )
                        ( A[i][k], B[k][j] )
```

The macro takes as input the two matrices A and B and the array indices i, j, k; it yields the pair `( A[i][k], B[k][j] )` which is used as input for GEMM's scalar function $f = *$ (see Section IV).

Views have to provide two additional functions for explicitly copying the local and private tiles to local or private memory – both functions are automatically generated in our approach. For example, for BLAS routine GEMM, we copy an $M \times K$-sized chunk of input matrix A and a $K \times N$-sized chunk of matrix B to local memory, and we generate a preprocessor macro to access the chunks in local memory as a 3-dimensional array of size $M \times N \times K$, as discussed above. We refrain from discussing our generated code for these two copy functions, because both implement straightforward copy operations in OpenCL.

b) Pseudocode: Listing 1 shows our generated OpenCL code (as pseudocode) for an arbitrary d-dimensional MDH $\texttt{md\_hom}(\ f,\ (\circledast_1, \ldots, \circledast_d)\ )$. The code is executed by the WIs in parallel in a Single-Program Multiple-Data (SPMD) fashion. Tuning parameters (see Table I) are highlighted in the listing; we determine the optimized values of these parameters via auto-tuning (this is described in Section VI), and we textually replace the parameters in our code by these optimized values before executing the code.

In the listing, each WI iterates over the local tiles that are processed by its corresponding WG, in each of the MDH's d dimensions (lines 13-15), and in each such iteration,



Listing 1: OpenCL implementation (pseudocode) for MDHs.

the WI iterates over the private tiles within its WG's local tile of the actual iteration (lines 17-19). Tuning parameters $\sigma^{\langle G\rangle}_{\texttt{mdh-do}}$ and $\sigma^{\langle L\rangle}_{\texttt{mdh-do}}$ (no. 9 in Table I) set the order in which local and private tiles are processed on global (`G`) and local (`L`) layer; for example, on local layer, if $\sigma^{\langle L\rangle}_{\texttt{mdh-do}}(1) = d$,$\sigma^{\langle L\rangle}_{\texttt{mdh-do}}(2) = d-1, \ldots$, then the private tiles within a local tile are processed first in dimension d, then in dimension $d-1$, and so on. The parameters enable, e.g., to postpone concatenation as discussed above. The number of loop iterations over local/private tiles (in lines 13-15 and 17-19) are computed straightforwardly in lines 1 and 2. Note that in our example in Figure 1, for a better illustration, we started exactly one WG/WI per local/private tile, causing only one iteration of these loops.

A WI computes (in parallel to the other WIs) a private tile (lines 21-26); for this, the WI applies the MDH's scalar function f to the tile's individual elements (line 24), and it combines the obtained results in the MDH's different dimensions using the corresponding combine operators (denoted as $+=_{\circledast_d}, \ldots, +=_{\circledast_1}$ in the listing). The operators are applied in the order $\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(d) \rightarrow \ldots \rightarrow \sigma^{\langle P\rangle}_{\texttt{mdh-do}}(1)$, according to parameter 9 in Table I.

To enable fast memory accesses, we cache the input explicitly in local and private memory, according to parameter $\texttt{MEM\_INP}^{\langle \texttt{LYR,b,i}\rangle}$ (no. 5 in Table I), using the view's automatically generated copy functions (discussed in the previous section). For example, when $\texttt{MEM\_INP}^{\langle \texttt{L,b,i}\rangle} = \texttt{\_\_local}$, for all $i \in \{1, \ldots, d\}$ and buffers b of the input view, we cache local tiles in local memory by calling the corresponding copy function after line 15 (not shown in the listing for brevity). In this case, we generate the following local memory buffer (in line 9) to cache the content of view's input buffer `b`: `__local inp_L_b[LT_SIZE`$^{\langle \sigma^{\langle L,b\rangle}_{\texttt{buff-do}}(1)\rangle}$`]...[LT_SIZE`$^{\langle \sigma^{\langle L,b\rangle}_{\texttt{buff-do}}(d)\rangle}$`]`. Here, parameter $\sigma^{\langle \texttt{LYR,b}\rangle}_{\texttt{buff-do}}$ (no. 8 in the table) flexibly sets the buffer's order of dimensions, which enables better exploiting locality, e.g., when the input is transposed (see Section V-B).

The WI combines its results on private (`P`) layer in the d different dimensions, using the MDH's combine operators. For this, it uses the memory regions $\texttt{res\_P\_b\_}\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(d), \ldots, \texttt{res\_P\_b\_}\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(1)$ (lines 24-26) – i.e., `res_P_b_d,...`, `res_P_b_1` (declared in line 10) after permuting dimensions according to function $\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(i)$. The WI uses first $\texttt{res\_P\_b\_}\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(d)$ as the memory space for combining its results in dimension $\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(d)$ (line 24); analogously, it uses $\texttt{res\_P\_b\_}\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(d-1), \ldots, \texttt{res\_P\_b\_}\sigma^{\langle P\rangle}_{\texttt{mdh-do}}(1)$ for combining its results in the further dimensions (line 25-26). We flexibly set the memory region in which the $\texttt{res\_P\_b\_i}$, $i \in \{1, \ldots, d\}$, reside (line 10) via tuning parameter $\texttt{RES}^{\langle \texttt{LYR,i}\rangle}$ (no. 6 in the table) to avoid a too heavy private and/or local memory usage, as discussed in Section V-B. Results on local and global layer are combined analogously (lines 27-29 and 30-32).

We combine the results of different WIs using parallel reduction. The WIs' results can be combined at different layers (according to tuning parameter 10 in Table I), e.g., after a private or local tile has been processed. For example, combining results early usually decreases consumption of fast memory resources, because afterwards, memory space is required for only one WI; however, in contrast to combining results late, WIs' results have to be combined more often. We do not present the combination of WIs' results in our pseudocode, because this is straightforward.

Note that in our code, we efficiently exploit locality by flexibly setting the mapping between array and OpenCL dimensions (lines 6-7), using tuning parameter $\sigma^{\langle \texttt{LYR}\rangle}_{\texttt{arr}\rightarrow\texttt{ocl}}$ (no. 7 in Table I).

Note further, that for clarity, we present a simplified pseudocode of our OpenCL implementation. In particular, we assume in our discussion that the input can be evenly partitioned in local tiles (i.e., the input size `N_i` is evenly divisible by

LT_SIZE$^{<i>}$), and that the number of local tiles is a multiple of the number of WGs (i.e., N_i / LT_SIZE_i is divisible by NUM_WG$^{<i>}$) – see line 1 in Listing 1; the same applies in line 2 to the private tile size and number of WIs. We do not have these restrictions in our actual implementation.

In the following, we implement MDHs using our own code generator, implemented in C++: it takes as input an MDH's md_hom expression and view function (see Section III), and it straightforwardly emits the corresponding OpenCL code for the MDH according to Listing 1.

VI. Auto-Tuning

We fully automatically optimize our generated code (Listing 1) – for any combination of an MDH function, device, and input size – by automatically choosing (auto-tuning) optimized values of its tuning parameters listed in Table I.

As concrete auto-tuner, we use the generic *Auto-Tuning Framework (ATF)* [60] – it automatically determines optimized values of the tuning parameters by exploring the parameters' search space (i.e., each possible combination of parameters' values). To provide good search results for applications from various domains, ATF uses for exploration ensembles of search techniques, e.g., Differential Evolution, Nelder-Mead, and Particle Swarm, similarly as in [61].

A major feature of ATF is that it efficiently handles tuning parameters with *interdependencies* between them. For example, in our generated code, the private tile size has to be smaller or equal to the local tile size (parameters 3 and 4 in Table I), because a private tile is a chunk of the local tile. Consequently, these tuning parameters have the following interdependency: PT_SIZE$^{<i>}$ $\leq$ LT_SIZE$^{<i>}$ for $i \in \{1, \ldots, d\}$.

For using ATF, we provide to it: i) our generated OpenCL code (Listing 1), ii) its tuning parameters (Table I) and their corresponding interdependencies (expressed in ATF as straightforward boolean expressions), iii) the OpenCL id of our target device, and iv) the sizes of the input array in the particular dimensions. ATF's usage is discussed in [60].

ATF requires a reasonable amount of time for auto-tuning our code, e.g., 4h for BLAS routines. As compared to the time and effort required for manually implementing and hand-optimizing code, for a particular device and input size, by an expert implementer, this time for fully automatic optimization can be neglected. Furthermore, in important application areas, e.g., deep learning, auto-tuning has to be performed only once per target device, because the same algorithms and input sizes are reused in multiple program runs, i.e., auto-tuning is an only one-time overhead. However, when the code for a particular device and/or input size is not reused, then auto-tuning can be time-critical. For such cases, we can auto-tune our code to achieve an average high performance over different input sizes. Alternatively, we can significantly reduce the auto-tuning time by straightforwardly splitting the search space into equally-sized parts and exploring each part in parallel on a different device. For example, using this approach, we reduce our tuning time for BLAS routines from 4h to only 1h when using 4 GPUs simultaneously for auto-tuning.

VII. Experimental Evaluation

We use our approach to automatically generate optimized code for computations from four important areas: 1) dense liner algebra (BLAS), 2) stencil computations, 3) data mining, and 4) tensor contractions. We compare the performance of our automatically generated and auto-tuned code on two fundamentally different devices – Intel CPU and NVIDIA GPU – to several state-of-practice approaches. To make comparison challenging for us, we use input sizes that are: i) used in real-world applications, e.g., the deep-learning framework Caffe [9], or ii) very preferable for our competitors which are often optimized toward specific input sizes, e.g., large powers of two.

A. Linear Algebra Routines (BLAS)

We focus on two of the most relevant BLAS routines (Section IV-A) – GEMM (GEneral Matrix-Matrix multiplication) and GEMV (GEneral Matrix-Vector multiplication). We express these two routines using the md_hom pattern (as described in Section IV-A), and we generate and auto-tune our code according to Sections V and VI.

a) Competitors: We compare the performance of our code on Intel CPU and NVIDIA GPU against well-performing BLAS competitors: 1) Lift – an academic framework that is closely related to our approach and has proven to provide high, portable performance for BLAS [47], and 2) Intel MKL 2019.3.199 and NVIDIA cuBLAS 10.1 – the latest versions – which are vendor-provided BLAS libraries that are optimized by hand at the assembly level to provide high performance on Intel or NVIDIA hardware, correspondingly. We also compare to the MDHs' initial OpenCL implementation described in [5].

b) Input Size: For comparison, we use for each routine two sizes: i) a real-world input size (abbreviated by RW in the following) taken from the state-of-the-art deep-learning framework Caffe [9]; for example, we use for GEMM's $M \times K$ and $K \times N$ input matrices a size of $(M, N, K) = (10, 500, 64)$ which is repeatedly called in Caffe's siamese sample, and ii) an input size that is preferable for our competitors (abbreviated by PC), e.g., big, square, power-of-two input matrices of size 1024×1024 taken from [62] for which Lift, MKL and cuBLAS are highly optimized.

c) Auto-Tuning: For each routine, we auto-tune our corresponding code for 4h for each individual combination of a device and input size (8 combinations in Figure 2). The Intel MKL and NVIDIA cuBLAS libraries rely on handcrafted heuristics and thus, they do not require tuning. For BLAS, Lift's search space is pruned by the Lift experts to only one OpenCL implementation per routine, and thus, Lift also does not require additional auto-tuning; however, with drawbacks presented in the following.

d) Experimental Results: In Figure 2, we demonstrate the speedup of our code for GEMM and GEMV on both Intel CPU (left) and NVIDIA GPU (right) over the Lift approach – the Lift programs are taken from the expert-implemented artifact implementation in [62]. We can observe that our GEMM implementation provides competitive and

CPU		GEMM		GEMV	
		RW	PC	RW	PC
MDH	GF/s	31	339	23	30
	SP	1.00	1.00	1.00	1.00
Lift	GF/s	fails	111	15	15
	SP	fails	3.04	1.51	1.99
MKL	GF/s	7	456	22	35
	SP	4.22	0.74	1.05	0.87
MKL-JIT	GF/s	23	N/A	N/A	N/A
	SP	1.37	N/A	N/A	N/A

GPU		GEMM		GEMV	
		RW	PC	RW	PC
MDH	GF/s	99	8996	415	440
	SP	1.00	1.00	1.00	1.00
Lift	GF/s	23	7713	118	148
	SP	4.33	1.17	3.52	2.98
cuBLAS	GF/s	34	10820	404	438
	SP	2.91	0.83	1.03	1.00
cuBLASLt	GF/s	84	36474	N/A	N/A
	SP	1.18	0.25	N/A	N/A

Fig. 2: Speedup (SP) of our automatically generated and auto-tuned code for dense linear algebra (BLAS) on Intel CPU (left) and NVIDIA GPU (right) using both a Real-World (RW) input size and an input size that is Preferable for our Competitor (PC). Lift fails on CPU for the RW size. MKL-JIT and cuBLASLt are not applicable for computing GEMV; MKL-JIT is also not applicable for GEMM on the PC size.

often even significantly better performance than Lift for both, the real-world, deep-learning input size (RW) and also for the Lift's preferable square, large, power-of-two size (PC). This is because for GEMM, the Lift's search space is hand-pruned by the Lift experts to only one single OpenCL implementation; most likely, the implementation is chosen with focus on large, power-of-two input sizes (e.g., PC) on GPUs. This pruning avoids tuning overhead; however, it severely affects Lift's performance on CPU and real-world input sizes as used in deep-learning: we obtain a speedup of $4.33\times$ of our approach over Lift on GPU for the real-world RW size, and on CPU, the Lift's OpenCL implementation fails for this size because of a too heavy resource usage. We also obtain a speedup over Lift for its preferable input size (PC) on GPU, because Lift's optimization relies on hand-pruning the search space, which often tends to miss good solutions. For example, our auto-tuned GEMM implementation avoids local memory usage for the PC size on GPU, while the Lift's hand-chosen implementation relies on local memory – most likely, because using local memory is recommended by NVIDIA, as it often leads to high performance.

In Figure 2, we compare also to the expert-designed, hand-optimized BLAS libraries Intel MKL and NVIDIA cuBLAS and their novel extensions – MKL-JIT [49] and cuBLASLt [50] – for high-performance GEMM. MKL-JIT is highly optimized for GEMM on small input sizes, e.g., as occurring in deep learning, and cuBLASLt provides the special `cublasLtMatmul` routine that provides advanced optimization options, e.g., using NVIDIA's novel tensor cores and selecting an optimized so-called *algorithm* for high performance on arbitrary input sizes.

We achieve competitive and, on the RW deep-learning input size, even better performance for GEMM as compared to MKL and cuBLAS. Note that we achieve this high performance, even though our approach is not designed and/or optimized toward BLAS and/or a particular device, while our competitors are specifically designed and hand-optimized for BLAS on either Intel CPU or NVIDIA GPU, correspondingly. We explain our good performance by the fact that we rely on a parameterized implementation that is auto-tuned for high performance for a particular combination of device and also input size; in contrast, MKL and cuBLAS rely on a set of pre-defined implementations, each optimized for a range of input sizes, to avoid auto-tuning overhead. This is beneficial when a BLAS routine is not reused on the same input size, because auto-tuning then becomes a bottleneck; however, in important application areas such as deep learning, the same sizes are reused in each program run, so that tuning time can be neglected.

The better performance of MKL and cuBLAS than our implementation for GEMM on the PC size is arguably explained by their BLAS-specific optimizations at the assembly level which cannot be expressed in OpenCL [63]. In particular, cuBLASLt provides significantly better performance than our approach, because it exploits the NVIDIA's novel Tensor Cores [64] which currently cannot be programmed using OpenCL. Note that MKL-JIT is specifically designed for small input sizes (such as our RW size) and thus, it is not applicable for large sizes such as PC.

We compare the performance of our code also to the MDHs' initial implementation in [5]. In all cases, we measure a speedup > 1. Our highest speedups are $2.64\times$ on CPU and $5.5\times$ on GPU – in both cases for GEMM on the PC size. The reason for this improvement is that our MDHs' implementation is targeted to both OpenCL's models – platform and memory – by incorporating a parameterized parallelization and tiling strategy, while the implementation in [5] targets the platform model only. Moreover, we exploit OpenCL's platform model more efficiently as compared to [5] by avoiding correlations between our tuning parameters for the platform and memory model (see Sections II and V).

B. Stencil Computations

In the following, we demonstrate experimental results for two important stencil computations (Section IV-B) – Gaussian Convolution (2D) and Jacobi (3D) [48].

a) Competitors: We compare the performance of stencil computations when implemented using our approach against: 1) Lift which provides high, portable performance for stencil computations [48], and 2) Intel MKL-DNN 0.18 [65] and NVIDIA cuDNN 7.5 [66] – the latest versions – which are both optimized by hand for high performance on either Intel CPU or NVIDIA GPU, respectively.

For a fair comparison to Lift, we configure our stencils according to Lift's artifact implementation [67], i.e.: we use in our experiments for the Gaussian stencil a 5×5 filter size, and we use a 7pt stencil shape for the Jacobi stencil; for both stencils, we use a single time step (i.e., $T = 1$), exactly as in the Lift's artifact.

b) Input Size: We use i) real-world sizes (RW in the following) of 224×224 for Gaussian Convolution taken from the area of deep learning [68], and $256 \times 256 \times 256$ for Jacobi [67], and ii) large input sizes which are preferable for our competitors (PC), of 4096×4096 for Gaussian Convolution and $512 \times 512 \times 512$ for Jacobi, both taken from [67].

c) Auto-Tuning: While for BLAS, the Lift's search space is pruned to only one OpenCL implementation, Lift's search space for Gaussian Convolution is hand-pruned to four implementations and to one implementation for Jacobi. In contrast to Lift's BLAS implementations, these implementations for Gaussian Convolution and Jacobi are generic in performance-critical parameters – the number of work-groups and work-items, and the size of local tiles (similar to parameters 1-3 in Table I). Lift auto-tunes each of the 4 Gaussian implementations for 1.25h (i.e., 5h in total) and the Jacobi implementation for 5h; for this, Lift also uses the ATF framework as we do. For a fair comparison, we use the same auto-tuning time of 5h for optimizing our code for each individual combination of a stencil computation, device, and input size (8 combinations in Figure 3). Intel MKL-DNN and NVIDIA cuDNN do not require additional tuning, because they rely on handcrafted heuristics.

		CPU			
		Gaussian (2D)		Jacobi (3D)	
		RW	PC	RW	PC
MDH	GF/s	73	208	49	60
	SP	**1.00**	**1.00**	**1.00**	**1.00**
Lift	GF/s	15	35	25	24
	SP	**4.90**	**5.96**	**1.94**	**2.49**
MKL-DNN	GF/s	10	15	N/A	N/A
	SP	**6.99**	**14.31**	**N/A**	**N/A**

		GPU			
		Gaussian (2D)		Jacobi (3D)	
		RW	PC	RW	PC
MDH	GF/s	386	4195	1137	1005
	SP	**1.00**	**1.00**	**1.00**	**1.00**
Lift	GF/s	165	3852	995	984
	SP	**2.33**	**1.09**	**1.14**	**1.02**
cuDNN	GF/s	102	219	N/A	N/A
	SP	**3.78**	**19.11**	**N/A**	**N/A**

Fig. 3: Speedup (SP) of our automatically generated and auto-tuned code for stencil computations on Intel CPU (left) and NVIDIA GPU (right) using a real-world input size RW and an input size PC that is preferable for our competitors. MKL-DNN and cuDNN are both not applicable for Jacobi.

d) Experimental Results: In Figure 3, when comparing to Lift – the Lift programs are taken from the artifact implementation in [67] – we observe better results with our MDH approach, because Lift's search space of OpenCL implementations is manually pruned toward GPU architectures and large input sizes. Thus, even though Lift performs auto-tuning (similarly as in our approach), it misses good results for CPU and real-world input sizes as used in deep learning. Increasing the auto-tuning time of Lift, to 24h, could not improve its performance, because well-performing implementations have been pruned out of its search space.

We obtain better results also than Intel MKL-DNN and NVIDIA cuDNN, because both approaches are optimized for *Multi-Channel Convolutions (MCC)* [65], [69] where a stencil computation is performed multiple times, e.g., as in deep learning. Note that for comparison, we have chosen Gaussian and Jacobi, because both are very favorable for Lift which is the fair competitor for us. We cannot compare to Intel MKL-DNN and NVIDIA cuDNN for Jacobi, because both are only applicable for stencils that are convolutions, such as Gaussian.

We can easily express MCC in our approach as a 5-dimensional stencil computation where the last combine operator is addition. We compare the performance of our MDH approach for MCCs to both MKL-DNN and cuDNN using a real-world input size of $1 \times 512 \times 7 \times 7 \times 512$, taken from the area of deep learning for which MKL-DNN and cuDNN are optimized.

Our speedup for MCC over Intel's MKL-DNN (CPU) is $1.3\times$, and over NVIDIA's cuDNN (GPU), we reach a speedup of $3.31\times$. Our good results are because MKL-DNN and cuDNN use hand-crafted heuristics for optimization, while we rely on auto-tuning to provide high performance. We cannot compare our MCC implementation to Lift, because currently, there is no Lift implementation of MCCs available for Intel CPU or NVIDIA GPU.

When comparing our code generation approach to the MDHs' initial implementation in [5], we reach for stencils again significantly better results – a speedup > 1 in all cases, with speedups up to $2.65\times$ on CPU and $1.69\times$ on GPU. Analogously to BLAS, this is because we target also the OpenCL's memory model, and we better exploit the OpenCL's platform model.

C. Data Mining

a) Competitor: We compare our generated and auto-tuned code for PRL (Section IV-C) to the PRL implementation of the *Epidemiological Cancer Registry (EKR)* in North Rhine-Westphalia (Germany) – the currently largest cancer registry in Europe. EKR uses for PRL a parallel Java implementation that targets multi-core CPUs.

b) Input Size: We perform our experiments using real-world input data provided by EKR. The EKR's data set contains 2^{20} patient records; we present results for differently-sized subsets of the the EKR-provided data.

		Probabilistic Record Linkage					
		2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH	SP	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
EKR	SP	**1.87**	**2.06**	**4.98**	**13.86**	**28.34**	**39.36**

Fig. 4: Speedup (SP) of our automatically generated and auto-tuned code for Probabilistic Record Linkage (PRL) on Intel multi-core CPU. We compare to the EKR's currently used parallel Java implementation for PRL.

c) Auto-Tuning: We auto-tune our implementation only once for a batch-size of 1024×1024. This batch size fully exploits the capabilities of our CPU or GPU, respectively, so that we can reuse our auto-tuned implementation for that size for all input sizes shown in Figure 4, i.e., our tuning

time (of 24h) becomes negligible, because it is an only one-time overhead. The Java implementation of EKR does not rely on auto-tuning – it starts as many threads as the target CPU provides cores – and thus, it has no additional overhead for tuning.

d) Experimental Results: Figure 4 demonstrates that our generated and auto-tuned OpenCL code has significantly better performance than EKR's parallel Java implementation – speedups $> 39\times$. This is because the EKR's implementation inefficiently exploits memory, while our implementation incorporates an efficient tiling strategy. Our generated OpenCL code can also easily be auto-tuned and executed on GPU, leading to speedups of up to $> 65\times$ as compared to the EKR's parallel Java implementation that works for only multi-core CPU.

D. Tensor Contractions

a) Competitors: We compare our generated and optimized code for tensor contractions (Section IV-D) to COGENT [22] and Facebook's Tensor Comprehensions [21] – both are specifically designed and optimized for tensor contraction on NVIDIA GPUs.

b) Input Size: For a fair comparison, we use the 9 real-world tensor contraction samples from COGENT's artifact implementation [22], which are taken from the TCCG benchmark suite [70].

c) Auto-Tuning: Tensor Comprehensions relies on auto-tuning (as our approach) and thus, it has an additional tuning overhead – of 12h on average for each of the 9 samples. The overhead is negligible when targeting deep learning – a common application field for tensor contractions – because the same contractions are reused in multiple program runs, i.e., auto-tuning has to be performed only once for a neural network on a particular system. For a fair comparison, we auto-tune our implementation also for 12h, analogously to Tensor Comprehensions. COGENT relies on hand-crafted heuristics and thus, it does not require additional tuning time.

d) Experimental Results: In Figure 5, we observe that we reach competitive and often even significantly better performance than both COGENT and Tensor Comprehensions – speedups of up to $2\times$ over both competitors. This is because we provide a more flexible implementation: for example, COGENT's private tile sizes are restricted in the summation dimensions (i.e., where the combine operator is $+$) to the fixed size of 1, while we allow arbitrary tile sizes in all dimensions, leading to higher performance. Note that increasing Tensor Comprehension's auto-tuning time, to 24h for the `RW4` size which has arguably the highest potential for higher performance, could not improve its performance. As compared to MDH's initial implementation [5], we again reach significantly better results for all input sizes (speedups of up to $2.46\times$).

Note that we reach better results than COGENT and Tensor Comprehensions, even though both approaches are specifically optimized for tensor contractions on NVIDIA GPUs, while our MDH approach is applicable for various application classes and devices.

		Tensor Contractions								
		RW 1	RW 2	RW 3	RW 4	RW 5	RW 6	RW 7	RW 8	RW 9
MDH	GF/s	5771	5322	7824	5739	5500	5029	6760	6628	6777
	SP	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***
COGENT	GF/s	4573	4584	3697	4631	4647	3685	4573	4604	3670
	SP	**1.26**	**1.16**	**2.12**	**1.24**	**1.18**	**1.36**	**1.48**	**1.44**	**1.85**
F-TC	GF/s	4834	2660	5476	1987	4088	3273	5426	3286	4561
	SP	**1.19**	**2.00**	**1.43**	**2.89**	**1.35**	**1.54**	**1.25**	**2.02**	**1.49**

Fig. 5: Speedup (SP) of our automatically generated and auto-tuned code for Tensor Contractions over Facebook's Tensor Comprehensions (F-TC) and COGENT on NVIDIA GPU using 9 real-world (`RW`) samples.

VIII. Conclusion & Future Work

We present a novel OpenCL code generation approach for Multi-Dimensional Homomorphisms (MDH) – a class of functions which can be conveniently expressed using the `md_hom` parallel pattern. We demonstrate for important data-parallel computations – from different domains: 1) dense linear algebra (BLAS), 2) stencil computations, 3) data mining, and 4) tensor contractions – that they can be conveniently expressed in the MDH formalism, and that we can automatically generate OpenCL code for them. Our generated code achieves high, portable performance over different parallel devices and input sizes by incorporating a parameterized parallelization and tiling strategy, generically for arbitrary MDH functions. We show that our code can be automatically optimized – for any combination of an MDH function, target device, and input size – by using classical auto-tuning for determining optimized values of our two strategies' parameters. Our experimental results show competitive and often even better performance of our automatically generated and auto-tuned code as compared to several state-of-practice approaches: we demonstrate speedups of up to $5\times$ over popular, performance-portable approaches, and competitive or even better performance as compared to hand-optimized approaches (e.g., Intel MKL and NVIDIA cuBLAS) on real-world input data as used in deep learning.

In future work, we aim to generate efficient, portable code for composed `md_hom` expressions – by generating a single, optimized OpenCL implementation for such expressions with fused loops – thereby enabling with the MDH approach code generation for more complex applications, e.g., machine-learning graphs [71]. We aim to target also sparse computations, for which the MDH approach has to be slightly extended, by irregularly shaped input arrays.

Acknowledgment

This work has been supported by the BMBF project HPC2SE and the DFG Cluster CiM. We thank our shepherd and the anonymous reviewers for their valuable feedback and remarks.

REFERENCES

[1] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[2] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. [Online]. Available: https://aip.scitation.org/doi/abs/10.1109/MCSE.2010.69

[3] P. Tillet and D. Cox, "Input-aware Auto-tuning of Compute-bound HPC Kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 43:1–43:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126939

[4] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," in *ACM Transactions on Graphics*, 2003, pp. 908–916.

[5] A. Rasch and S. Gorlatch, "Multi-Dimensional Homomorphisms and Their Implementation in OpenCL," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 101–119, Feb. 2018. [Online]. Available: https://doi.org/10.1007/s10766-017-0508-z

[6] S. Gorlatch and M. Cole, "Parallel Skeletons," in *Encyclopedia of Parallel Computing*, 2011, pp. 1417–1422.

[7] Intel. (2019) Math Kernel Library. [Online]. Available: https://software.intel.com/en-us/mkl

[8] NVIDIA. (2019) cuBLAS library. [Online]. Available: https://developer.nvidia.com/cublas

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[10] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 193–205. [Online]. Available: http://dl.acm.org/citation.cfm?id=3314872.3314896

[11] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "AlphaZ: A System for Design Space Exploration in the Polyhedral Model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 17–31.

[12] C. Chen, J. Chame, and M. Hall, "CHiLL: A Framework for Composing High-Level Loop Transformations ," Technical Report 08-897. University of Southern California, Tech. Rep., 2008.

[13] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, Jun. 2006. [Online]. Available: https://doi.org/10.1007/s10766-006-0012-3

[14] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines," 2012.

[15] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop Transformation Recipes for Code Generation and Auto-Tuning," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2009, pp. 50–64.

[16] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized Optimizations for Empirical Tuning," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[17] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea, "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 556–571. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062354

[18] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct. 2015, pp. 138–149.

[19] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375595

[20] T. Grosser, A. Groesslinnger, and C. Lengauer, "Polly — Performing Polyhedral Optimizations on a Low-Level Intemediate Representation," vol. 22, no. 04, p. 1250010, 2012. [Online]. Available: https://doi.org/10.1142/S0129626412500107

[21] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," vol. abs/1802.04730, 2018. [Online]. Available: http://arxiv.org/abs/1802.04730

[22] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on gpus," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 85–95. [Online]. Available: http://dl.acm.org/citation.cfm?id=3314872.3314885

[23] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 429–443. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694364

[24] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. v. d. Geijn, "The science of deriving dense linear algebra algorithms," vol. 31, no. 1, pp. 1–26, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1055531.1055532

[25] D. Fabregat-Traver and P. Bientinesi, "A domain-specific compiler for linear algebra operations," in *High Performance Computing for Computational Science - VECPAR 2012*, M. Daydé, O. Marques, and K. Nakajima, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 346–361.

[26] F. G. Van Zee and R. A. van de Geijn, "Blis: A framework for rapidly instantiating blas functionality," vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[27] N. Kyrtatas, D. G. Spampinato, and M. Püschel, "A basic linear algebra compiler for embedded processors," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015, pp. 1054–1059.

[28] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, , F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.

[29] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: Extreme Performance Portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, Nov. 2018.

[30] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[31] M. W. Moskewicz, A. Jannesari, and K. Keutzer, "Boda: A holistic approach for implementing neural network computations," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: ACM, 2017, pp. 53–62. [Online]. Available: http://doi.acm.org/10.1145/3075564.3077382

[32] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The Tensor Algebra Compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 77:1–77:29, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133901

[33] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance gpu code for stencil computations," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1902–1920, Nov. 2018.

[34] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[35] O. Aumage, D. Barthou, and A. Honorat, "A Stencil DSEL for Single Code Accelerated Computing with SYCL," in *SYCL 2016 1st SYCL Programming Workshop during the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Barcelone, Spain, Mar. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01290099

[36] M. Ciznicki, M. Kulczewski, P. Kopta, and K. Kurowski, "Scaling the gcr solver using a high-level stencil framework on multi-and many-core architectures," in *Parallel processing and applied mathematics*. Springer, 2016, pp. 594–606.

[37] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2464996.2467268

[38] R. Membarth, F. Hannig, J. Teich, and H. Köstler, "Towards domain-specific computing for stencil codes in hpc," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 1133–1138.

[39] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Resource conscious reuse-driven tiling for gpus," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 99–111. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967967

[40] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2d and 3d stencils on gpus," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA: ACM, 2016, pp. 92–102. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884047

[41] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel, "Program generation for small-scale linear algebra applications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 327–339. [Online]. Available: http://doi.acm.org/10.1145/3168812

[42] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in haskell," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10. New York, NY, USA: ACM, 2010, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/1863543.1863582

[43] J. Svensson, M. Sheeran, and K. Claessen, "Obsidian: A domain specific embedded language for parallel programming of graphics processors," in *Implementation and Application of Functional Languages*, S.-B. Scholz and O. Chitil, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 156–173.

[44] F. Franchetti, F. de Mesmay, D. McFarlin, and M. Püschel, "Operator language: A program generation framework for fast kernels," in *Domain-Specific Languages*, W. M. Taha, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 385–409.

[45] L. Liu and Z. Li, "Improving parallelism and locality with asynchronous algorithms," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693483

[46] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code," vol. 50, no. 9, pp. 205–217, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2858949.2784754

[47] M. Steuwer, T. Remmelg, and C. Dubach, "Matrix Multiplication Beyond Auto-tuning: Rewrite-based GPU Code Generation," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: ACM, 2016, pp. 15:1–15:10. [Online]. Available: http://doi.acm.org/10.1145/2968455.2968521

[48] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High Performance Stencil Code Generation with Lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018, pp. 100–112. [Online]. Available: http://doi.acm.org/10.1145/3168824

[49] Intel. (2019) Intel Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM). [Online]. Available: https://software.intel.com/en-us/articles/intel-math-kernel-library-improved-small-matrix-performance-using-just-in-time-jit-code

[50] NVIDIA. (2019) cuBLASLt. [Online]. Available: https://developer.nvidia.com/cuda-toolkit/whatsnew

[51] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Nov. 1998, p. 38.

[52] D. E. Tanner, "Tensile: Auto-tuning gemm gpu assembly for all problem sizes," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018, pp. 1066–1075.

[53] C. Nugteren, "Clblast: A tuned opencl blas library," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '18. New York, NY, USA: ACM, 2018, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/3204919.3204924

[54] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient gemm on gpus," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '19. New York, NY, USA: ACM, 2019, pp. 229–241. [Online]. Available: http://doi.acm.org/10.1145/3293883.3295734

[55] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices," *Parallel Computing*, vol. 81, pp. 1 – 21, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118301091

[56] P. Christen, *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[57] Intel, "OpenCL Optimization Guide," 2014. [Online]. Available: https://software.intel.com/sites/default/files/managed/72/2c/gfxOptimizationGuide.pdf

[58] NVIDIA, "NVIDIA OpenCL Best Practices Guide," 2009. [Online]. Available: https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[59] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/106972.106981

[60] A. Rasch and S. Gorlatch, "ATF: A Generic, Directive-Based Auto-Tuning Framework," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 5, p. 14 pp., 2019, e4423 cpe.4423. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4423

[61] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628092

[62] Lift BLAS Artifact, "https://gitlab.com/michel-steuwer/cgo_2017_artifact," 2017.

[63] J. Lai and A. Seznec, "Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs," in *International Symposium on Code Generation and Optimization*, 2013, pp. 1–10.

[64] NVIDIA. (2019) NVIDIA Tensor Cores. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensorcore/

[65] Intel. (2018) Math Kernel Library for Deep Learning Networks. [Online]. Available: https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation

[66] NVIDIA. (2019) CUDA® Deep Neural Network library. [Online]. Available: https://developer.nvidia.com/cudnn

[67] Lift Stencil Artifact, "https://gitlab.com/larisa.stoltzfus/liftstencil-cgo2018-artifact," 2018.

[68] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer

parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[69] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[70] P. Springer and P. Bientinesi. (2019) Tensor Contraction Code Generator (TCCG). [Online]. Available: https://github.com/HPAC/tccg

[71] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

ARTIFACT DESCRIPTION

A. Abstract

This artifact describes the experiments that were conducted for the PACT'19 paper *Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms*.

The following workflow makes use of scripts to compile, run, and evaluate the presented benchmarks in Figures 2 - 5. At least one OpenCL-enabled device is required and, in case the user intends to execute the NVIDIA cuBLAS, NVIDIA cuBLASLt, NVIDIA cuDNN, COGENT or Tensor Comprehensions benchmarks, an NVIDIA GPU is required, as well. We expect performance results analogous to those presented and discussed in Section VII.

B. Artifact check-list (meta-information)

- **Compilation:** C++ Compiler with C++14 support.
- **Run-time environment:** Linux OS with OpenCL, CMake 3.8 or higher, Boost 1.56 or higher, Python 2.7, Java 8 SDK, Intel MKL 2019.3.199, NVIDIA cuBLAS 10.1, NVIDIA cuDNN 7.5, Tensor Comprehensions 0.1.1.
- **Hardware:** An OpenCL-enabled device; optionally, also an NVIDIA GPU.
- **Execution:** Ensure that no other programs are running while performing steps E.2 and E.3 in Sections *Auto-Tuning* and *Benchmarking*.
- **Output:** Tables showing speedups (analogous to Figures 2 - 5).
- **Experiments:** All experiments can be executed using our provided bash scripts.
- **How much disk space required (approximately)?:** 3 GB.
- **Publicly available?:** Yes.

C. Description

1) How delivered: The artifact archive is hosted at https://gitlab.com/mdh-project/pact_2019_artifact.

2) Hardware dependencies: At least one OpenCL-enabled device is required in a system with at least 16 GB RAM. Additionally, an NVIDIA GPU is required in case the user intends to execute the NVIDIA cuBLAS, NVIDIA cuBLASLt, NVIDIA cuDNN, COGENT, or Tensor Comprehensions benchmarks.

3) Software dependencies: The artifact has several software dependencies:

- A compiler supporting C++14 or higher,
- OpenCL 1.2,
- CMake 3.8 or higher,
- Python 2.7,
- Java 8 SDK,
- tabulate Python package,
- OpenTuner 0.8.0,
- jq,
- Intel MKL 2019.3.199,
- NVIDIA cuBLAS 10.1,
- NVIDIA cuDNN 7.5,
- Tensor Comprehensions 0.1.1.

In addition, Lift has the following dependencies:

- Boost 1.56 or higher
- OpenGL (libmesa)
- OpenSSL
- finger

D. Installation

The following steps are required for installing the artifact:

1) **Install the dependencies:**
   Detailed installation instructions can be found in file `README.md` in the artifact archive.
2) **Download the artifact:**
   ```
   $ git clone https : / / gitlab . com / mdh - project/pact_2019_artifact.git
   $ cd pact_2019_artifact
   ```
3) **Configure the benchmarks:**
   Adapt the file `environment.env` to set the OpenCL platform and device id, and to optionally disable reference implementations that should not be evaluated (e.g., to reduce artifact's runtime). Note that in order to evaluate COGENT, the `CUDA_ARCH` variable in file `environment.env` has to be set appropriately.
   Afterwards, execute:
   ```
   $ source environment.env
   ```
4) **Compile the benchmarks:**
   ```
   $ ./scripts/install.sh
   ```
   We expect dependencies to be installed in their default locations; otherwise, the user has to pass corresponding flags to our `install.sh` script, which will be directly forwarded to the underlying CMake project.

E. Experiment workflow

The user is invited to perform the following steps:

1) **Initializing the artifact:**
   Change into the artifact directory and initialize the artifact:
   ```
   $ cd pact_2019_artifact
   $ source environment.env
   ```
2) **Auto-Tuning:**
   Per default, our artifact provides pre-tuned kernels for Intel Xeon E5-2640 v2 CPU and NVIDIA Tesla V100-SXM2-16GB GPU (provided in the artifact's `defaults` folder). To use our pre-tuned kernels, please execute:
   ```
   $ ./scripts/use_defaults.sh
   ```
   Be aware that – in case the artifact is executed on devices different from the ones listed in the paper – auto-tuning has to be performed in order to achieve the best, and thus portable, performance. Omitting the tuning and using our provided pre-tuned kernels for new devices may prevent the user from reproducing the results shown in the paper!
   The auto-tuning is started by executing:
   ```
   $ ./scripts/tune.sh
   ```
   Note that this step may take a long time ($\sim$ 475h per device for the full evaluation in Figures 2 - 5), because for each routine and input size, both our generated OpenCL code and some of the reference implementations (in particular: Lift and Tensor Comprehensions), are auto-tuned for several hours. The user may edit the file `experiments.json` to decrease the auto-tuning time, e.g., by reducing the number of input sizes to auto-tune for or the tuning time per framework. *Be aware that reducing the tuning time can lead to poor performance!*
3) **Benchmarking:**
   - Execute benchmarks:
     ```
     $ ./scripts/benchmark.sh
     ```
   - Print results:
     ```
     $ ./scripts/print_results.sh
     ```

   The best found configurations can be found in the `results` folder.

F. Evaluation and expected result

We compare the performance of our automatically generated and auto-tuned code against several state-of-practice approaches. We

expect performance results that are analogous to those presented in Section VII of our paper.

G. Experiment customization

The artifact can be arbitrarily customized (e.g., regarding input sizes and frameworks to compare to) using files `environment.env` and `experiments.json` in the artifact's root folder.

In `environment.env`, the user can configure the compilation step of the artifact workflow (Section D.4), e.g., disable CPU or GPU evaluation. Make sure to apply changes made to the `environment.env` file by executing: `$ source environment.env`.

In `experiments.json` the user can configure the auto-tuning and benchmarking step (Sections E.2 and E.3). For each routine examined in the experimental section, the user can choose which input sizes and frameworks to run, and the amount of time to use for auto-tuning our code and the reference implementations, respectively.

.