(De/Re)-Compositions Expressed Systematically via MDH-Based Schedules

Ari Rasch
a.rasch@uni-muenster.de
University of Muenster, Germany

Richard Schulze
r.schulze@uni-muenster.de
University of Muenster, Germany

Denys Shabalin
shabalin@google.com
Google, Switzerland

Anne Elster
elster@ntnu.no
NTNU, Norway

Sergei Gorlatch
gorlatch@uni-muenster.de
University of Muenster, Germany

Mary Hall
mhall@cs.utah.edu
University of Utah, USA

Abstract

We introduce a new scheduling language, based on the formalism of *Multi-Dimensional Homomorphisms (MDH)*. In contrast to existing scheduling languages, our MDH-based language is designed to systematically *de-compose* computations for the memory and core hierarchies of architectures, and *re-compose* the computed intermediate results back to the final result – we say *(de/re)-composition* for short. We argue that our scheduling langauge is easy to use and yet expressive enough to express well-performing (de/re)-compositions of popular related approaches, e.g., the TVM compiler, for MDH-supported computations (such as linear algebra routines and stencil computations). Moreover, our language is designed as auto-tunable, i.e., any optimization decision can optionally be left to the auto-tuning engine of our system, and our system can automatically recommend schedules for the user, based on its auto-tuning capabilities. Also, by relying on the MDH approach, we can formally guarantee the correctness of optimizations expressed in our language, thereby further enhancing user experience. Our experiments on GPU and CPU confirm that we can express optimizations that cannot be expressed straightforwardly (or at all) in TVM's scheduling language, thereby achieving higher performance than TVM, and also vendor libraries provided by NVIDIA and Intel, for time-intensive computations used in real-world deep learning neural networks.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: scheduling languages, GPU, CPU

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02.
https://doi.org/10.1145/3578360.3580269

ACM Reference Format:
Ari Rasch, Richard Schulze, Denys Shabalin, Anne Elster, Sergei Gorlatch, and Mary Hall. 2023. (De/Re)-Compositions Expressed Systematically via MDH-Based Schedules. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580269

1 Introduction

Program code in state-of-the-art low-level approaches, like CUDA and OpenCL, requires complex optimization to efficiently target the deep and complex memory and core hierarchies of modern architectures, such as GPU and CPU. Modern high-performance compilers [8, 10, 12, 21, 22, 27, 39, 50, 53] automatically generate well-performing low-level code; the optimization processes of these compilers are often manually guided by a performance expert who explicitly expresses code optimizations for the compiler (such as tiling and parallelization) in form of programs in a so-called *scheduling language*. While such compilers with an expert-guided optimization process have a high performance potential, their scheduling languages usually consist of a set of fine-grained low-level commands that have to be combined in complex ways for expressing well-performing optimizations, making the optimization process complex, cumbersome, and error-prone for the performance expert.

We introduce a new scheduling language, based on the formalism of *Multi-Dimensional Homomorphisms (MDH)* [41, 42]. Our MDH-based scheduling language enables a systematic optimization process for MDH-supported computations [42] (such as linear algebra routines and stencil computations), by offering a single scheduling primitive that systematically *de-* and *re-composes* computations to/from the memory and core hierarchies of state-of-the-art architectures (in the following, referred to as *(de/re)-composition*). We argue that the systematic nature of our language simplifies implementing and reasoning about schedules, thereby contributing to a simplified code optimization process for performance experts. To further simplify the optimization process for the expert, our language expresses all optimization decisions (e.g., choosing an optimized memory access

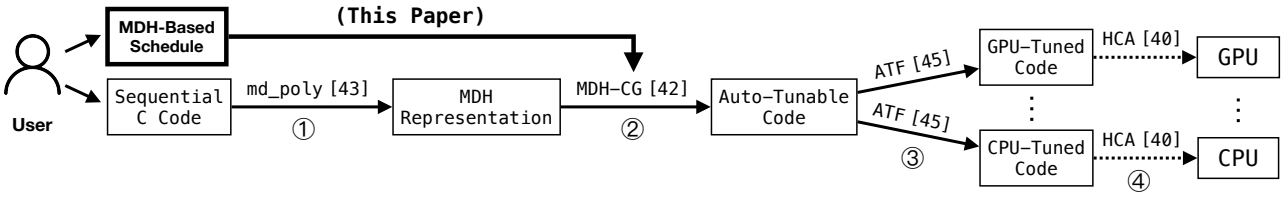

Figure 1. Overview of our approach (contribution of this paper highlighted in bold)

pattern) as auto-tunable, thereby enabling selecting particular (or even all) optimization decisions fully automatically, as an alternative to choosing all optimizations manually by the human expert. In contrast, existing scheduling languages support auto-tuning often only for a restrictive set of optimizations only, e.g., for automatically choosing optimized sizes of tiles but not for choosing an optimized memory access pattern, as we discuss in this paper. Moreover, based on its auto-tuning capabilities, our system can automatically recommend an already well-performing schedule (as we confirm experimentally in Section 6) that can be fine-tuned by an expert user toward even higher performance. Also, by relying on the MDH formalism and its algebraic foundation, we can mathematically ensure the correctness of optimizations expressed in our language, thereby further contributing to user's productivity.

Our experiments on two NVIDIA GPUs and two Intel CPUs confirm that our scheduling language is capable of expressing the optimization decisions of the popular TVM [12] compiler for deep learning computations (matrix multiplication and convolution), using real-world data sets taken from three important neural networks: *ResNet-50* [23], *VGG-16* [46], and *MobileNet* [26]. Our experiments also confirm that we are able to achieve higher performance than TVM, as well as vendor libraries provided by NVIDIA and Intel, on both kinds of architectures, by exploiting the auto-tuning capabilities of our language design and its expressivity which is sometimes beyond that of TVM's scheduling language.

The rest of the paper is structured as follows. In Section 2, we give a general overview of our approach and highlight its particular contribution. Section 3 briefly summarizes the existing MDH approach, and Section 4 recapitulates existing scheduling languages. We introduce our new scheduling language in Section 5 and experimentally evaluate our approach in Section 6. Section 7 discusses related work, and Section 8 concludes our paper.

2 Overview

Figure 1 shows the overview of our approach. The original work on MDHs, without the contributions of this paper, takes as input a sequential C program consisting of a perfect loop nest with static loop bounds (possibly annotated with an optional, OpenMP-like directive that enables advanced optimization, like parallelization in reduction dimensions [43]). Arrays in the loop body are expected to be accessed via pure index functions that take loop iterator variables as input and can be statically resolved. An example input for our approach is presented in Listing 1 and discussed later in Section 5. In Figure 1, the loop nest is transformed, in step ①, to an equivalent MDH representation, using the `md_poly` compiler [43]. Based on the program's MDH representation, the original MDH work automatically generates auto-tunable code, in step ②, using the MDH's *Code Generator (CG)* [42]; the generated code is then auto-tuned[1], in step ③, to optimized, executable code (e.g., in CUDA or OpenCL) using the *Auto-Tuning Framework (ATF)* [45]. Finally, the auto-tuned code is executed, in step ④, on the target device (e.g., a GPU or CPU) using *Host Code Abstraction (HCA)* [40] – a high-level library for programming *host code* which is required in modern approaches, like CUDA and OpenCL, for program execution (e.g., for managing data transfers between host and device memory).

```
1  #pragma mdh( C[i][j] : ++ , ++ , + )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

In this paper, we extend the existing MDH workflow in Figure 1 by the part highlighted in bold in the figure: we enable performance experts to incorporate expert knowledge about optimizations into the workflow, via *MDH-based schedules*. Our schedules conveniently express MDH-supported optimizations, e.g., exploiting fast memory resources and parallelization, in a structured, systematic way (focus of Section 5). The expert's decisions are then incorporated in step ② into the generated code, rather than generating the code in step ② as generic in these decisions and requesting the decisions later in step ③ from the auto-tuner (as done in the original MDH work).

[1] While code generation approaches often use auto-tuning for simple, numerical values only (e.g., identifying optimized sizes of tiles and numbers of threads), MDH uses auto-tuning also for more advanced optimizations, e.g., identifying optimized memory access patterns and exploiting fast memory resources for the input/output data [42].

By incorporating the user into the optimization process, we enable two major advantages over the original MDH work: 1) better optimization, as an auto-tuning system might not always make the same high-quality optimization decisions as a human expert; 2) faster auto-tuning, as some (or even all) optimization decisions might be made by the expert user and thus are not left to the costly auto-tuner.

3 The MDH Approach

We demonstrate the existing MDH formalism by expressing and discussing the example of *Matrix Multiplication (*`MatMul`*)*.

```
MatMul<Type T | int I,J,K> :=
       out_view<T>( C:(i,j,k)->(i,j) ) o
         md_hom<I,J,K>( *, (++,++,+) ) o
           inp_view<T,T>( A:(i,j,k)->(i,k) ,
                          B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (`MatMul`) expressed in the MDH formalism

Listing 2 shows how `MatMul` is expressed in MDH – we derive such MDH expressions automatically for the user from straightforward, annotated C code (Listing 1), according to step ① in Figure 1. In Listing 2, we first fuse the domain-specific input of `MatMul` – two matrices $A \in T^{I \times K}$ and $B \in T^{K \times J}$ both of type `T` (e.g., `T=float`) – to a 3-dimensional array of pairs `(A[i,k],B[k,j])` $\in T \times T$. For this, we use MDH's higher-order function (a.k.a *skeleton* or *pattern* in programming [18]) `inp_view` which the MDH formalism provides to specify accesses to the input data (two input matrices in the case of `MatMul`). Higher-order function `md_hom` expresses the basic computation part of computations: for our `MatMul` example, after fusing `MatMul`'s two input matrices via `inp_view`, function `md_hom` applies `MatMul`'s scalar function `*` (multiplication) to each pair within the output array of `inp_view`, and `md_hom` combines afterwards the obtained results (multiplied pairs) in dimensions 1 and 2 via `++` (concatenation), and in dimension 3 via `+` (addition). Pattern `out_view` specifies `MatMul`'s access to its output matrix. The access is trivial in this example, but `out_view` could potentially be used to store `MatMul`'s result matrix as, e.g., transposed (by replacing index function `"(i,j,k)↦(i,j)"` with `"(i,j,k)↦(j,i)"`) or in a stride fashion (by using `"(i,j,k)↦(i*s1,j*s2)"`, for strides `s1,s2` $\in \mathbb{N}$), etc.

Based on the expression in Listing 2, the existing MDH approach fully automatically generates executable program code (steps ②-④ in Figure 1), e.g., in CUDA for GPUs or OpenCL for CPUs.

4 State-of-the-Art Scheduling Languages

Existing scheduling languages operate on a low abstraction level: they offer primitives, like `tile` and `bind`, to express fine-grained code optimizations, making the existing languages expressive, but also complex, as the primitives have to be combined in complex ways to express well-performing optimizations. The low-level design of the existing languages makes them particularly hard to error-check automatically and also hard to combine with techniques from automatic program optimization (auto-tuning [9]), as we discuss later.

```
# exploiting fast memory resources for "C":
matmul_local, = s.cache_write([matmul], "local")
matmul_1, matmul_2, matmul_3 = tuple(matmul_local.op.axis) + tuple(matmul_local.op.reduce_axis)
SHR_1, REG_1 = s[matmul_local].split(matmul_1, factor=1)
# 9 further split commands
s[matmul_local].reorder(BLK_1, BLK_2, DEV_1, DEV_2, THR_1, THR_2, DEV_3, SHR_3, SHR_1, SHR_2, REG_3, REG_1, REG_2)

# ... (loop unrolling)

# tiling:
matmul_1, matmul_2, matmul_3 = tuple(matmul.op.axis) + tuple(matmul.op.reduce_axis)
THR_1, SHR_REG_1 = s[matmul].split(matmul_1, factor=1)
# 5 further split commands
s[matmul].reorder(BLK_1, BLK_2, DEV_1, DEV_2, THR_1, THR_2, SHR_REG_1, SHR_REG_2)
s[matmul_local].compute_at(s[matmul], THR_2)

# block/thread assignments:
BLK_fused = s[matmul].fuse(BLK_1, BLK_2)
s[matmul].bind(BLK_fused, te.thread_axis("blockIdx.x"))
# ... (similar to lines 18 and 19)

# exploiting fast memory resources for "A":
A_shared = s.cache_read(A, "shared", [matmul_local])
A_shared_ax0, A_shared_ax1 = tuple(A_shared.op.axis)
A_shared_ax0_ax1_fused = s[A_shared].fuse(A_shared_ax0, A_shared_ax1)
A_shared_ax0_ax1_fused_o, A_shared_ax0_ax1_fused_i = s[A_shared].split(A_shared_ax0_ax1_fused, factor=1)
s[A_shared].vectorize(A_shared_ax0_ax1_fused_i)
# ...
s[A_shared].compute_at(s[matmul_local], DEV_3)

# exploiting fast memory resources for "B":
# ... (analogous to lines 23 – 29)
```

Listing 3. TVM+Ansor schedule (shortened for brevity) for Matrix Multiplication as used in ResNet-50 network on NVIDIA Ampere GPU

Listing 3 shows an example schedule of the popular TVM compiler (shortened and simplified for brevity) generated using TVM's recent Ansor optimization engine [55]. The TVM schedule expresses optimization decisions for *Matrix Multiplication (*`MatMul`*)* in CUDA when computed on NVIDIA A100 GPU for input matrices taken from the real-world ResNet-50 neural network. In lines 2-6 the primitives express that CUDA's fast *register memory* should be used for storing intermediate results of the computed `C` output matrix. For this, the computation of the `C` matrix is de-composed using multiple times primitive `split` (lines 4 and 5) and reordered (line 6). Similarly, the schedule expresses using fast *shared memory* for the `A` and `B` input matrices (lines 23-32). Lines 11-15 prescribe a basic loop structure in which the copy operations for matrices to/from fast memory resources are inserted (lines 15, 29, 32).

```
1  I, J, K = 16, 1000, 2048
2
3  A = te.placeholder((I, K), dtype='float32')
4  B = te.placeholder((K, J), dtype='float32')
5
6  k = te.reduce_axis((0, K))
7  C = te.compute(
8      (I, J),
9      lambda i, j:
10     te.sum(A[i, k] * B[k, j], axis=k)
11 )
```

Listing 4. Matrix multiplication expressed in TVM's high-level program representation

Listing 4 shows for completeness how `MatMul` is expressed in TVM's high-level program representation[2] which operates on the same, high abstraction level as the MDH program in Listing 2. Based on this representation in Listing 4 and the scheduling program in Listing 3, TVM generates executable CUDA code for `MatMul` that is optimized according to the optimization decisions expressed in Listing 3.

5 MDH-Based Schedules

This section introduces and discusses our MDH-based scheduling language. In Section 5.1, we illustrate our language design using a CUDA example, and we show in Section 5.2 how our language is used for programming approaches different from CUDA. Section 5.3 outlines our code generation (e.g., in CUDA or OpenCL), and Section 5.4 discusses the formal correctness of our approach. Section 5.5 shows how autotuning is used in our language, and Section 5.6 discusses how we automatically generate schedules for the user that can be fine-tuned by hand toward toward higher performance. Section 5.7 demonstrates how our scheduling programs can be visualized and also be generated from visual inputs.

[2] Our approach frees the user from the burden of using special representations as in Listings 4 and 2, by taking as input straightforward, annotated program code in the well-known C programming language (Listing 1).

5.1 Language Design

We illustrate our MDH-based scheduling language by showing how it is used for expressing the particular optimization decisions of the deep-learning compiler TVM for matrix multiplication in Listing 3. To express matrix multiplication in our approach, we provide to our compiler: 1) a scheduling program, and 2) a straightforward C implementation of `MatMul` (Listing 1); the implementation is optionally annotated with an OpenMP-like MDH directive in line 1 which enables advanced optimizations, e.g., parallelizing loops whose iterations depend on each other (as in line 4 of Listing 1). The directive indicates that intermediate results computed by iterations of the first two loops (lines 2 and 3 in Listing 1) are combined straightforwardly (via symbol ++ which denotes concatenation in the MDH formalism) and that iterations of the third loop (line 4) are combined non-trivially via point-wise addition (symbol +).

Listing 5 shows the program in our scheduling language that is equivalent to the TVM schedule in Listing 3, i.e., we generate from Listings 5 and 1 the same CUDA code (apart from some minor syntactical differences) as TVM generates from Listings 3 and 4 for matrix multiplication when computed on NVIDIA A100 GPU using ResNet-50 input matrices.

Our language consists of exactly one primitive, namely `(de/re)-comp`; we use the primitive to split the computation (in this example `MatMul`) systematically into smaller subproblems that we assign to the memory and core hierarchies of the target architecture.
Our primitive has the following, general structure:

```
(de/re)-comp( /* sub-problem size */ )
            ( /* memory hierarchy assignments */ )
            ( /* core   hierarchy assignments */ )
```

We describe our primitive in the following by showing how it is used for successively (de/re)-composing `MatMul` for the GPU's memory and core hierarchies, in 6 steps shown in Listing 5: lines 7-10 (step 1), lines 12-15 (step 2), etc. Lines 1-5 in Listing 5 are optional in our approach and serve for completeness only; they indicate that: a) our iteration space has initially a size of `(I,J,K):= (16,1000,2048)`, according to the ResNet-50 input matrices which are of sizes `(I,K)` = `16x2048` and `(K,J)` = `2048x1000`; b) the input and output matrices are read/written from/to CUDA's device memory (`DM`) in CUDA and that matrices use a standard memory layout (indicated via `[1,2]` in lines 3 and 4); c) the computation (a.k.a. *kernel* in CUDA) is performed by a GPU. Low-level code optimizations (loop unrolling, constant substitution, etc) are implicit in our approach to keep our language simple – these optimization often do not need to be controlled explicitly to achieve high performance, as we confirm in our experiments later. We perform low-level optimizations automatically based on straightforward heuristics.

Block Parallelization (lines 8-10). The MatMul computation is split into tiles of size $8 \times 20 \times 2048$ (line 8), i.e., $(16/8) * (1000/20) * (2048/2048) = 2 * 50 * 1$ many tiles. Symbol ^ (a.k.a. *caret*) is used in line 8 for convenience: it indicates that the tile size of the previous step is re-used in the last dimension (i.e., this dimension is not tiled), and in line 9 that the memory regions of buffers are re-used, i.e., no memory optimizations are performed in this (de/re)-composition step. Line 10 indicates that we iterate over tiles in parallel, using *CUDA Blocks (*BLK*)*. Here, a non-standard *swizzle pattern* [37] is used, i.e., we use CUDA's block dimension x to iterate over tiles in the second tile dimension (i.e., 50 CUDA blocks are started in x dimension), and we use CUDA dimension y to iterate over tiles in the first dimension (2 CUDA blocks started in y dimension); the standard swizzle pattern would be using block dimension x for the first tile dimension and block dimension y for the second, correspondingly.

Tiling (lines 13-15). Each of the $(8 \times 20 \times 2048)$-sized tiles of the previous step is split into further tiles of size $4 \times 20 \times 2048$ (only the first dimension is tiled). No memory optimizations are performed in this (de/re)-composition step (indicated by caret symbols in line 14), and the tiles are processed sequentially, using a 3-level for-loop nest[3]; numbers .1/.2/.3 in line 15 prescribe the order of loops in our generated code (discussed in detail in Section 5.3): the first loop in the nest iterates over tiles in the first dimension (i.e., the loop makes $8/4 = 2$ iterations), the second loop over tiles in the second dimension (one iteration), and third loop over tiles in the third dimension (one iteration). Consequently, this (de/re)-composition step in lines 13-15 expresses a classical loop tiling optimization. Annotation 6: in line 13 expresses that in our generated code (discussed in Section 5.3), the loop nest for iterating over the $(8 \times 20 \times 2048)$-sized tiles is generated after loops for steps 1-5 in lines 8-10 and 19-36, i.e., at the innermost loop level.

Thread Parallelization & Register Memory Utilization (lines 19-21). Each tile of the previous step is again split into tiles of size $1 \times 1 \times 2048$, which are computed in parallel by *CUDA Threads (*THR*)* (using again a non-standard swizzle pattern). Each thread computes its part of the C output matrix in *Register Memory (*RM*)* (line 20).

Shared Memory Utilization (lines 24-26). Each of the (1×1×2048)-sized tiles of the previous step is split into further tiles of size $1 \times 1 \times 256$; for each of the new $(1 \times 1 \times 256)$-sized tiles, the corresponding parts of input matrices A and B are accessed in *CUDA's Shared Memory (*SM*)* (data copies are performed in our generated code). According to line 26, the tiles are processed sequentially, via a 3-level for-loop nest – in contrast to loops generated according to line 15, the loops

```
1  // initialization
2  0: (de/re)-comp( 16,1000,2048 )
3                 ( A:DM[1,2],B:DM[1,2] ;
4                   C:DM[1,2]           )
5                 ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de/re)-comp( 8,20,^ )
9                 ( ^,^ ; ^ )
10                ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de/re)-comp( 4,^,^ )
14                ( ^,^ ; ^ )
15                ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads &
18 // utilization of CUDA Register Memory
19 2: (de/re)-comp( 1,1,^ )
20                ( ^,^ ; C:RM[1,2] )
21                ( THR.y,THR.x,THR.z )
22
23 // utilization of CUDA Shared Memory
24 3: (de/re)-comp( ^,^,256 )
25                ( A:SM[1,2],B:SM[1,2] ; ^ )
26                ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de/re)-comp( ^,^,2 )
30                ( ^,^ ; ^ )
31                ( ^,^,^ )
32
33 // tiling 3
34 5: (de/re)-comp( ^,^,1 )
35                ( ^,^ ; ^ )
36                ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

according to line 26 are permuted: the first loop in the nest iterates over tiles in the last dimension (a.k.a. *k-dimension* in the context of MatMul), and the other two loops iterate over the first and second tile dimension (this is a typical locality optimization[4] for MatMul [32]).

Tiling (lines 29-31 and lines 34-36). Classical tiling is expressed (similarly to lines 13-15).

5.2 Targeting Different Programming Models

While Section 5.1 presents our scheduling language for expressing CUDA optimizations, our language can be used for targeting also other kinds of programming models, such as OpenMP for CPUs and OpenCL for multiple kinds of architectures; the models rely on deeper/shallower memory and core hierarchies than CUDA. For example, an OpenMP schedule

[3] CUDA offers blocks (BLK) and threads (THR) to iterate over multi-dimensional index spaces in parallel, and it uses nested for-loops (FOR) to iterate over spaces sequentially.

[4] In this particular example, the permutation has no effect on performance, because two loops in the nest make only one iteration. We express this permutation in Listing 5 to be consistent with the TVM schedule in Listing 3.

in our language would be very similar to a CUDA schedule as in Listing 5 with the only differences that: 1) the user can use in our OpenMP schedules only `THR.xxx` (as in line 21 of Listing 5), but not `BLK.xxx` (line 10 of the listing), because OpenMP relies on a 1-layered thread hierarchy only and thus has no notion of CUDA blocks (BLK); 2) instead of CUDA memory tags `DM`, `SM`, `RM` (as in lines 20,25 of Listing 5), our OpenMP schedules use memory tags `L1`,`L2`, `L3` (for caches) and `MM` (for main memory). In our generated OpenMP code, as the OpenMP model does not allow explicitly programming caches, our OpenMP memory optimizations correspond to the popular *packing* optimization [11].

5.3 Code Generation

Listing 6 shows as pseudocode the CUDA program that we generate according to the optimization decisions in Listing 5. Each (de/re)-composition step in Listing 5 corresponds to a part of a deep loop nest in the pseudocode (lines 2-32). Copy operations, e.g., from device to shared memory, are inserted at the corresponding positions in the pseudocode (lines 17 and 45). Buffers are accessed via indices `i_<l>_<d>` (lines 34-41) where `<l>` and `<d>` indicate the accessed tile's corresponding layer `l` and dimension `d`; functions `idx_<MEM>_<DIM>` straightforwardly flatten the index space, e.g.:

```
idx_RM_1(i_2_1,i_4_1,i_5_1,i_6_1) =
    i_2_1 * (I_4_1*I_5_1*I_6_1) + i_4_1 *
      (I_5_1*I_6_1) + i_5_1 * (I_6_1) + i_6_1
```

5.4 Correctness

Our scheduling language is designed such that the correctness of our scheduling programs can be statically verified, by checking the formal constraints defined by the MDH formalism [41, 42], e.g., that the results of different CUDA thread blocks can be re-composed in the reduction dimension (the loop in line 4 of Listing 1) in device memory only in CUDA [34]. For example, using `C:SM[1,2]` or `C:RM[1,2]` in line 9 of Listing 5 would be invalid in general[5] – we issue an error message – because CUDA cannot combine results of different thread blocks in shared memory `SM` or register memory `RM`.

5.5 Auto-Tuning

Our scheduling language is designed such that any optimization decision can optionally be selected automatically via auto-tuning [9], as an alternative to hand-choosing all optimizations manually by a performance expert. Thereby, we enable conveniently combining human expert knowledge with techniques from automatic program optimization.

[5] For the particular scheduling program in Listing 5, using `C:SM[1,2]` or `C:RM[1,2]` in line 9 is valid, because only one thread block is started in the reduction dimension.

```
1  // 1: parallelization over CUDA Blocks
2  i_1_2 = blockIdx.x; { // (1000/20) blocks
3  i_1_1 = blockIdx.y; { // (16/8)    blocks
4  for(i_1_3 = 0 ; i_1_3 < 2048/2048 ; ++i_1_3) {
5
6  // 2: parallelization over CUDA Threads &
7  //    utilization of CUDA Register Memory
8  i_3_2 = threadIdx.x; { // (20/1) threads
9  i_3_1 = threadIdx.y; { // (4/1)  threads
10 for(i_3_3 = 0 ; i_3_3 < 2048/2048 ; ++i_3_3) {
11
12 // 3: utilization of CUDA Shared Memory
13 for(i_4_3 = 0 ; i_4_3 < 2048/256 ; ++i_4_3) {
14 for(i_4_1 = 0 ; i_4_1 < 1/1      ; ++i_4_1) {
15 for(i_4_2 = 0 ; i_4_2 < 1/1      ; ++i_4_2) {
16
17 // copy parts of A & B from DM to SM
18
19 // 4: tiling 2
20 for(i_5_3 = 0 ; i_5_3 < 256/2 ; ++i_5_3) {
21 for(i_5_1 = 0 ; i_5_1 < 1/1   ; ++i_5_1) {
22 for(i_5_2 = 0 ; i_5_2 < 1/1   ; ++i_5_2) {
23
24 // 5: tiling 3
25 for(i_6_3 = 0 ; i_6_3 < 2/1 ; ++i_6_3) {
26 for(i_6_1 = 0 ; i_6_1 < 1/1 ; ++i_6_1) {
27 for(i_6_2 = 0 ; i_6_2 < 1/1 ; ++i_6_2) {
28
29 // 6: tiling 1
30 for(i_2_1 = 0 ; i_2_1 < 8/4       ; ++i_2_1) {
31 for(i_2_2 = 0 ; i_2_2 < 20/20     ; ++i_2_2) {
32 for(i_2_3 = 0 ; i_2_3 < 2048/2048 ; ++i_2_3) {
33
34 C_RM[ idx_RM_1( i_2_1,i_4_1,i_5_1,i_6_1 ) ,
35       idx_RM_2( i_2_2,i_4_2,i_5_2,i_6_2 ) ]
36 +=
37   A_SM[ idx_SM_1( i_2_1,i_5_1,i_6_1 ) ,
38         idx_SM_3( i_2_3,i_5_3,i_6_3 ) ]
39   *
40   B_SM[ idx_SM_3( i_2_3,i_5_3,i_6_3 ) ,
41         idx_SM_2( i_2_2,i_5_2,i_6_2 ) ];
42
43 }...} // i_2_3 - i_4_3
44
45 // copy parts of C from RM to DM
46
47 }...} // i_3_3 - i_1_1
```

Listing 6. CUDA pseudocode generated according to MDH schedule in Listing 5 and program specification in Listing 1

To exploit auto-tuning in our language, the user uses the question mark symbol. For example, if we used in line 20 of Listing 5 symbol ? for the memory layout (i.e., `RM[?,?]` instead of `RM[1,2]`), our auto-tuner would try to automatically identify a memory layout that is optimized for the particular target architecture and characteristics of the input/output data. In addition to symbol question mark, our language allows restricting the set of potential values, by explicitly stating them in curly braces, thereby reducing the search space size and consequently the tuning time.

Internally, our approach generates the search space of only valid optimization decisions; invalid configurations are detected according to Section 5.4 and excluded from the search space (a.k.a. *Constrained-Optimization Problem (COP)* [52]), for an efficient auto-tuning process. As concrete auto-tuner, we use the *Auto-Tuning Framework (ATF)* [45] which has proved to be efficient for complex COPs.

Combining human expert knowledge with auto-tuning is important: auto-tuning might make better performing decisions than humans for some optimizations (e.g., identifying optimized tile size values), and auto-tuning enables using the same scheduling program for multiple devices (a.k.a *performance portability* [36]).

5.6 Automatic Schedule Generation

Our system can automatically generate a schedule for the user; the generated schedule then can be fine-tuned by an expert user toward higher performance. We confirm in Section 6 that our automatically generated schedules already achieve encouraging performance results, even when they are not fine-tuned by a human expert.

For automatically generating schedules, our system uses a schedule that contains multiple (de/re)-composition steps; in each step, the `(de/re)-comp` primitive contains question mark symbols ? only, which our auto-tuner replaces for the user by concrete values optimized for the particular target architecture and characteristics of the input and output data. The concrete number of (de/re)-composition steps is chosen by our system specifically for the target programming model. For example, our system recommends a schedule with 3 + 3 (de/re)-composition steps when targeting CUDA, because CUDA relies on three core layers (*Core*, *Warp*[6], and *Block*) and three memory layers (*Device*, *Shared*, and *Private*), as discussed in Section 5.

5.7 Visualization

To further enhance the usability of our approach, we introduce an equivalent, graphical representation for our scheduling programs. Our graphical scheduling representation enables both: 1) visualizing a scheduling program implemented as code (as in Listing 5), and 2) generating the scheduling program from a visual input.

Figure 2 shows our graphical representation for the scheduling program in Listing 5 (lines 12-31 of the listing are abbreviated in Figure 2, via vertical ellipsis, for brevity). Flexible parts of our graphical schedule representation are highlighted gray in Figure 2 and are set manually by the user (or left to the auto-tuner via symbol ?) when generating scheduling code from the visual representation.

[6] Warps (`WRP`) represent a further thread layer in CUDA and can be used in our language the same as `BLK` and `THR` in Listing 6. Warps are not used in Listing 6, because the TVM schedule in Listing 3 does not exploit warp-level optimizations, such as *shuffle operations* [33].

In our graphical representation, each (de/re)-composition step is separated via dotted lines. The split sizes (e.g., from line 2 in Listing 5) are denoted in the center part of Figure 2. Memory regions for input/output buffers (as in lines 3,4 of Listing 5) are stated below buffers in Figure 2, and a buffer's desired memory layout (e.g., `[1,2]` also in lines 3,4) is visualized in form of a two-dimensional coordinate cross. To each dimension of the iteration space (depicted as three-dimensional coordinate crosses in the center part of Figure 2), we assign a corresponding core layer (e.g., denoted in line 5 in Listing 5). Formally, the MDH approach uses different iteration spaces for the input data (visualized in Figure 2 via the three-dimensional coordinate crosses below `INP MDA`) and output data (coordinate crosses below `OUT MDA`); the axes are annotated by our visualization tool with the corresponding combine operators: in MDH, always concatenation ++ for input data, and concatenations and addition for the output in the case of `MatMul` (as discussed in Section 3). However, when targeting with MDH program code in imperative-style languages (such as CUDA, OpenCL, and OpenMP), these two iteration spaces coincide. Consequently, in Figure 2, the two three-dimensional coordinate crosses must always be annotated equally in the gray text fields, as in this work, we focus on imperative-style programming models. The numbers `0-6` in boxes, which are also located in the center part of Figure 2, correspond to the numbers in, e.g., line 2 in Listing 5, and set the order in which (de/re)-composition steps are processed in our generated code. The index functions used to access input and output buffers are stated below view functions in Figure 2.

6 Experimental Evaluation[7]

We experimentally evaluate the performance achieved by our approach as compared to TVM. To make comparison challenging for us, we focus on time-intensive computations used in real-world deep learning neural networks, for which TVM is specifically designed and optimized: *Matrix Multiplication (*`MatMul`*)* and *Multi-Channel Convolution (*`MCC`*)* [14] as used in networks *ResNet-50*, *VGG-16*, and *MobileNet*, according to their TensorFlow implementations [47–49] when computing the popular *ImageNet* [28] data set.

We generate TVM schedules using its Ansor optimizer [55], and we use for our approach the schedules that are automatically generated by our system (discussed in Section 5.6).

For a fair comparison, we use for each tuning run of our auto-tuner [45] and Ansor the same, generous auto-tuning time of 12h. We use such generous tuning time to avoid tuning issues in our experiments, which are not relevant for this work and analyzed in [45] – in all tuning runs, the final tuning result could be found in less than 12h for both our approach as well as Ansor.

[7] All experiments described in this section can be reproduced using our artifact implementation [7].

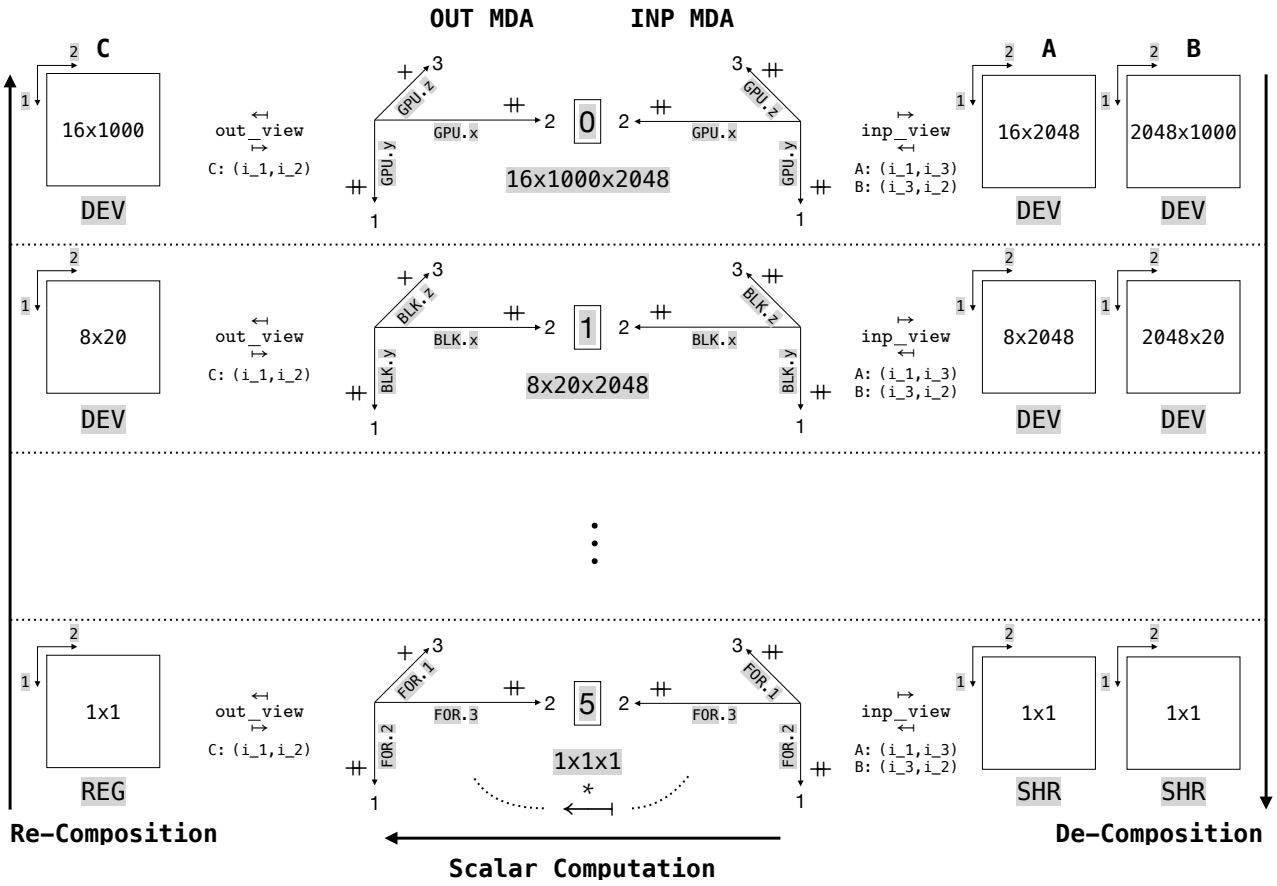

Figure 2. Visual representation of the `MatMul` (de/re)-composition in Listing 5 (lines 12-31 abbreviated via ellipsis, for brevity)

6.1 Experimental Setup

We run our experiments on a cluster containing two different kinds of GPUs and CPUs:

- `NVIDIA Ampere GPU A100-PCIE-40GB`
- `NVIDIA Volta GPU V100-SXM2-16GB`
- `Intel Xeon Skylake CPU Gold-6140`
- `Intel Xeon Broadwell CPU E5-2683`

The same as TVM, we use our approach to generate CUDA code when targeting GPUs, and OpenCL code for CPUs[8].

We use recent versions of frameworks, libraries, and compilers: TVM 0.8.0; NVIDIA cuBLAS and NVIDIA cuDNN from NVIDIA HPC SDK 22.1; Intel oneMKL/oneDNN 2022.0.0. For all experiments, we collect measurements until the 99% confidence interval was within 5% of our reported means, according to [25].

We use as operating system CentOS Linux release 7.9.2009, Linux kernel version 3.10.0-1160.80.1.el7.x86_64, and NVIDIA driver 520.61.05.

[8] TVM allows for CPUs also generating *LLVM code* [30] which includes assembly-level optimizations (currently beyond the scope of our work).

6.2 Case Study: Deep Learning

Figure 3 reports the speedup achieved by our approach over TVM+Ansor; performance achieved by assembly-optimized approaches are also reported for completeness, but currently beyond the scope of our approach which targets CUDA, OpenCL, and OpenMP, all of which operating at a higher abstraction level.

We observe that we usually achieve the high performance of TVM+Ansor and often perform even better. For example, we achieve a speedup $> 2\times$ over TVM on `NVIDIA Ampere GPU` for matrix multiplications as used in the inference phase of the ResNet-50 neural network, because our auto-tuning system has decided to exploit parallelization for computing inner tiles (a.k.a *strided memory access*), i.e., in lines 31 and 36 of Listing 5, rather than lines 10 and 21, which achieves higher performance for this particular `MatMul` example from ResNet-50. In contrast, Ansor rigidly parallelizes the computations of outer tiles (lines 10 and 21); most likely because binding parallelization to tiles cannot be expressed straightforwardly as auto-tunable in TVM's scheduling language. Also, for `MatMul`-like computations, Ansor always caches parts of the input matrices in GPU's shared memory, and it

Deep Learning	NVIDIA Ampere GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.00	1.26	1.05	2.22	1.00	1.42	1.00	1.14	1.00	1.00
NVIDIA cuDNN	0.92	–	1.85	–	1.22	–	1.94	–	1.81	2.14
NVIDIA cuBLAS	–	1.58	–	2.67	–	0.93	–	1.04	–	–

Deep Learning	NVIDIA Volta GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.00	1.21	1.00	1.79	1.00	1.11	1.06	1.00	1.00	1.00
NVIDIA cuDNN	1.21	–	1.29	–	2.80	–	3.50	–	2.32	3.14
NVIDIA cuBLAS	–	1.33	–	1.14	–	1.09	–	1.04	–	–

Deep Learning	Intel Skylake CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.05	1.14	1.20	1.97	1.14	2.38	1.27	3.01	1.40
Intel oneDNN	0.39	–	5.07	–	1.22	–	9.01	–	1.05	4.20
Intel oneMKL	–	0.44	–	1.09	–	0.88	–	0.53	–	–
TVM+Ansor (LLVM)	1.20	0.67	0.90	0.26	1.42	0.76	0.66	0.76	0.56	0.36

Deep Learning	Intel Broadwell CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.60	1.29	1.53	1.32	1.00	1.27	1.02	2.42	1.92
Intel oneDNN	1.30	–	1.81	–	2.94	–	2.85	–	1.83	4.47
Intel oneMKL	–	1.45	–	1.36	–	1.35	–	0.50	–	–
TVM+Ansor (LLVM)	1.45	1.35	1.06	0.72	1.63	0.85	0.98	0.79	1.14	0.52

Figure 3. Speedup (higher is better) of our approach over TVM+Ansor and vendor libraries for time-intensive deep learning computations. Dash symbol indicates unsupported computations. Assembly-optimized approaches (currently beyond the scope of our work) are separated by a straight line and are listed for completeness.

computes these cached parts always in register memory. In contrast, our auto-tuner has decided to not cache inputs into fast memory resources for this particular `MatMul` example in ResNet-50. For the `MatMul` example of ResNet-50's training phase, we achieve a positive speedup over TVM+Ansor, because our auto-tuner decided to store parts of input matrix A as transposed into fast memory (by using in our scheduling program `A[2,1]` instead of `A[1,2]`). Storing parts of the input/output data as transposed is not considered by Ansor as optimization, because such optimization cannot be expressed in TVM's scheduling language [4]. Moreover, we achieve a speedup over TVM+Ansor for `MatMul` used in the training phase of the VGG-16 network: we start 64 CUDA blocks for computing `MatMul`'s reduction dimension, while TVM+Ansor uses 1 block only for this dimension, because TVM cannot parallelize reductions over CUDA blocks [2].

Our positive speedups over TVM+Ansor for other experiments are for similar reasons as discussed above.

The better performance of vendor libraries in Figure 3 over our approach is because the libraries are optimized at the assembly level, whereas our approach operates at the higher CUDA/OpenCL abstraction level which offers less optimization opportunities [19, 29]. Our approach sometimes achieves better performance than libraries, because our implementations can be auto-tuned for a particular combination of architecture and also data characteristics (such as size and

memory layout), the same as the TVM+Ansor-generated implementations. In contrast, the libraries rely on hand-chosen heuristics optimized toward average high performance over data characteristics only.

6.3 Computations Different from Deep Learning

Our approach achieves encouraging performance results also for computations different from deep learning, which we do not discuss for brevity, because TVM is highly optimized toward deep learning computations, thereby often not achieving good performance results for other computations. For example, we achieve speedups over TVM+Ansor of $> 170\times$ already for straightforward dot products, because TVM struggles with optimizing reduction computations. We achieve the same encouraging performance results as reported for the existing MDH+ATF approach in [42–45], e.g., for quantum chemistry computations and data mining algorithms, because our approach internally uses MDH and ATF (see Figure 1) when generating schedules automatically.

7 Related Work

Popular scheduling approaches include TVM [12], Halide [39], Elevate [22], DaCe [10], Tiramisu [8], CUDA-CHiLL [27], Fireiron [21], Sequioa [16], DISTAL [53], and LoopStack [50]. All these approaches have in common that their scheduling languages rely on fine-grained, low-level primitives (as in Listing 3), which are expressive but complex and error-prone to use, usually even for experts. In contrast, our scheduling language relies on a single high-level primitive that systematically de- and re-composes computations to/from the memory and core hierarchies of architectures. We argue that the systematic, hierarchical nature of our language design simplifies implementing and reasoning about schedules. Furthermore, we have designed our language such that optimizations can be auto-tuned; in contrast, the related approaches often have only limited potential for auto-tuning. For example, TVM offers *AutoTVM* [13] which can be conveniently used for auto-tuning tile size values in TVM schedules, but not for identifying optimized memory access patterns, because memory access patterns are challenging to express as auto-tunable in TVM's scheduling language.

Moreover, we see the following advantages of our approach over the related work.

By relying on the MDH formalism, we are able to mathematically guarantee the correctness of our generated code and to provide strong error checking. In contrast, it is easy to implement schedules in Fireiron that generate incorrect GPU code, without issued error messages. Polyhedral approaches, such as CUDA-CHill and Tiramisu, can guarantee the correctness of some constraints, e.g., of tiling optimizations, but they often have difficulties with constraints set by programming models, e.g., that in CUDA, the results of thread blocks can only be combined in device memory. TVM also tends to struggle with detecting user errors [5, 6].

While existing scheduling languages often offer optimizations for multiple loop nests (e.g., so-called `fusing` primitives), they tend to suffer from expressivity issues for individual nests. For example, Fireiron has a particular focus on data movement optimizations, but it has difficulties with loop-level optimizations, e.g., expressing loop permutations. In contrast, TVM is able to conveniently express loop permutations, but it struggles with optimizing data movements [4].

Scheduling approaches are often limited to narrow classes of computations and architectures: Fireiron can only be used for matrix multiplications on GPUs, and TVM struggles with computations relying on multiple combine operators [1, 3].

Other related approaches include [15, 20, 24, 31, 35, 51, 54] which operate at a higher abstraction level than our approach. For example, [31] introduces program optimizations at the tensor abstraction level, whereas our approach is focussed on mapping tensor programs eventually to the memory and core hierarchies of state-of-the-art parallel architectures. We consider higher-level approaches as greatly combinable with our approach, by using them as a frontend for our system.

8 Conclusion & Future Work

We introduce a new scheduling language, based on the approach of Multi-Dimensional Homomorphisms (MDH). The goal of our language design is to express (de/re)-compositions in a systematic way, thereby simplifying the complex and error-prone optimization process for performance experts. In particular, we have designed our language such that correctness of our schedules can be checked automatically and that any optimization decision can optionally be left to our auto-tuner. We demonstrate that our language can express optimization decisions of the popular TVM compiler for computations from its target application class – deep learning – and often outperforms TVM in these computations due to the design and expressiveness of our language.

In future work, we aim to target also computations consisting of multiple loop nests, rather than optimizing loop nests individually. Moreover, we plan to target domain-specific hardware extensions, such as NVIDIA's tensor cores (inspired by [17]) and we aim to support further programming models, e.g., LLVM [30] to benefit from assembly-level optimizations. Another major goal is to support computations on sparse data formats, which requires slightly extending the MDH approach, similarly as in [38].

Acknowledgments

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project *PPP-DL (470527619)*. We thank the reviewers for their helpful comments and Albert Cohen and Cosmin Oancea for valuable discussions.

References

[1] Appache TVM Community. 2020. Non top-level reductions in compute statements. https://discuss.tvm.apache.org/t/non-top-level-reductions-in-compute-statements/5693.

[2] Appache TVM Community. 2022. Bind reduce axis to blocks. https://discuss.tvm.apache.org/t/bind-reduce-axis-to-blocks/2907.

[3] Appache TVM Community. 2022. Expressing nested reduce operations. https://discuss.tvm.apache.org/t/expressing-nested-reduce-operations/8784.

[4] Appache TVM Community. 2022. Implementing Array Packing via cache_read. https://discuss.tvm.apache.org/t/implementing-array-packing-via-cache-read/13360.

[5] Appache TVM Community. 2022. Undetected parallelization issue. https://discuss.tvm.apache.org/t/undetected-parallelization-issue/13224.

[6] Appache TVM Community. 2022. Undetected type issue. https://discuss.tvm.apache.org/t/undetected-type-issue/13223.

[7] Artifact Implementation. 2022. https://doi.org/10.1145/3554351.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

[9] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083. https://doi.org/10.1109/JPROC.2018.2841200

[10] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*.

[11] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. https://doi.org/10.48550/ARXIV.2003.00532

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[13] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf

[14] Vincent Dumoulin and Francesco Visin. 2018. A guide to convolution arithmetic for deep learning. arXiv:stat.ML/1603.07285

[15] Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-Conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 610–624. https://doi.org/10.1145/3314221.3314612

[16] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. Association for Computing Machinery, New York, NY, USA, 83–es. https://doi.org/10.1145/1188455.1188543

[17] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2022. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. https://doi.org/10.48550/ARXIV.2207.04296

[18] Sergei Gorlatch and Murray Cole. 2011. Parallel skeletons. In *Encyclopedia of parallel computing*. Springer-Verlag GmbH, 1417–1422.

[19] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[20] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27, 4 (dec 2001), 422–455. https://doi.org/10.1145/504210.504213

[21] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3410463.3414632

[22] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408974

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[24] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. *SIGPLAN Not.* 52, 6 (June 2017), 556–571. https://doi.org/10.1145/3140587.3062354

[25] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[26] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[27] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. *ACM Trans. Archit. Code Optim.* 9, 4, Article 31 (jan 2013), 25 pages. https://doi.org/10.1145/2400682.2400690

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[29] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10. https://doi.org/10.1109/CGO.2013.6494986

[30] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[31] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[32] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (jul 1996), 424–453. https://doi.org/10.1145/233561.233564

[33] NVIDIA. 2018. Warp-level Primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

[34] NVIDIA. 2022. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/.

[35] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (aug 2021), 29 pages. https://doi.org/10.1145/3473593

[36] S. J. Pennycook, J. D. Sewall, and V. W. Lee. 2016. A Metric for Performance Portability. https://doi.org/10.48550/ARXIV.1611.07409

[37] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

[38] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2020. Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR. In *Proceedings of the 29th International Conference on Compiler Construction (CC 2020)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/3377555.3377896

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[40] Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2020. dOCAL: high-level distributed programming with OpenCL and CUDA. *The Journal of Supercomputing* 76, 7 (2020), 5117–5138. https://doi.org/10.1007/s11227-019-02829-2

[41] Ari Rasch and Sergei Gorlatch. 2016. Multi-Dimensional Homomorphisms and Their Implementation in OpenCL. In *International Workshop on High-Level Parallel Programming and Applications (HLPP)*. 101–119.

[42] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 354–369. https://doi.org/10.1109/PACT.2019.00035

[43] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2020. `md_poly`: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms. In *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. 1–4.

[44] Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019. High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. Association for Computing Machinery, New York, NY, USA, 526–533. https://doi.org/10.1145/3297280.3297330

[45] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1, Article 1 (jan 2021), 26 pages. https://doi.org/10.1145/3427093

[46] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://doi.org/10.48550/ARXIV.1409.1556

[47] TensorFlow. 2022. MobileNet v1 models for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/mobilenet.py.

[48] TensorFlow. 2022. ResNet models for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/resnet.py.

[49] TensorFlow. 2022. VGG16 model for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/vgg16.py.

[50] Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. 2022. LoopStack: a Lightweight Tensor Algebra Compiler Stack. https://doi.org/10.48550/ARXIV.2205.00618

[51] M.E. Wolf and M.S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (1991), 452–471. https://doi.org/10.1109/71.97902

[52] Stephen Wright and Jorge Nocedal. 1999. Numerical Optimization. *Springer Science* 35, 67–68 (1999), 7.

[53] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

[54] Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying Dependent Reductions in the Polyhedral Model. *Proc. ACM Program. Lang.* 5, POPL, Article 20 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434301

[55] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Received 2022-11-10; accepted 2022-12-19