**Abstract**

Polyhedral compilers automatically parallelize sequential programs for multi- and many-core architectures, such as CPU and GPU. However, parallel code generated by state-of-the-art polyhedral compilers often lacks performance portability, because the existing compilers are usually optimized toward only a single particular architecture (e.g., GPU). Moreover, even on their target architecture, polyhedral compilers sometimes fail to reach high performance, because they often miss important optimizations, e.g., efficiently exploiting fast memory resources.

We present our work-in-progress results for md_poly – a novel polyhedral compiler that generates portable high-performance code from sequential C programs with perfect loop nests and rectangular iteration domains. In contrast to the existing polyhedral compilers, md_poly relies on the code generation approach for Multi-Dimensional Homomorphisms (MDHs): we show that the internal program representation of polyhedral compilers (a.k.a. polyhedral model) can be automatically transformed into an equivalent MDH representation; this representation is suitable for generating high-performance program code that is performance portable over different architectures. Our preliminary experimental comparison against PPCG with two benchmarks – Gaussian Convolution and Matrix Multiplication – shows encouraging results: speedups up to 7× on Intel CPU and 3× on NVIDIA GPU on real-world input sizes from deep learning.

**1 Motivation**

Programming state-of-the-art parallel architectures such as multi-core CPU and many-core GPU is challenging. For high performance, the programmer has to optimize its source code for the complex hardware of modern parallel devices which are characterized by deep and complex core and memory hierarchies. Moreover, for portable performance over such architectures, the programmer has to consider that architectures may differ significantly in their characteristics, e.g., the number of cores and sizes of caches.

Polyhedral compilers [1, 21, 24] simplify parallel programming by automatically parallelizing sequential program code, e.g., implemented in the C programming language. For this, a polyhedral compiler extracts from the sequential program code the so-called polyhedral model – a mathematical representation of the code, which captures important information, e.g., the number of loop iterations and memory access relations (read and/or write). The extracted model is then optimized by the compiler via so-called affine transformations which enable important optimizations, e.g., tiling.

State-of-the-art polyhedral compilers have a major weakness: they are usually optimized toward only a single particular architecture (e.g., only GPU) and thus, they often fail to reach high performance on other architectures (e.g., multi-core CPU). For example, we demonstrate experimentally that the popular polyhedral compiler PPCG (Polyhedral Parallel Code Generator) [24] reaches lower relative performance on Intel CPU than on NVIDIA GPU as compared to hand-optimized approaches. Moreover, our experiments show that PPCG sometimes fails to reach high performance also on NVIDIA GPU, because its generated code lacks important optimizations, e.g., efficiently exploiting fast memory resources.

In this paper, we present our work-in-progress results for md_poly – a novel compiler, with a polyhedral front end, that generates portable high-performance code for both multi- and many-core architectures, e.g., Intel CPU and NVIDIA GPU. For this, md_poly relies on the code generation approach for Multi-Dimensional Homomorphisms (MDH) [13, 16]: we demonstrate that the polyhedral program representation – currently limited to programs with perfect loop nests and rectangular iteration domains – can be automatically transformed into an equivalent MDH representation which is suitable for generating high-performance code that is performance portable over different architectures (e.g., CPU and GPU) [16].

Our preliminary experiments show encouraging results: we show that md_poly achieves better performance than the popular polyhedral compiler PPCG – by up to 7× on CPU and 3× on GPU – on two benchmarks taken from the Polybench suite [11]: Gaussian Convolution and Matrix Multiplication on real-world input sizes from deep learning.

**2 Overview**

Figure 1 demonstrate the overview of md_poly’s internal design. Starting from a sequential C program (with perfect loop nests and rectangular iteration domains), we first extract in step (1) in the figure the polyhedral model – this is same step
in all C-based polyhedral compilers – using the Polyhedral Extraction Tool (pet) [23]. Afterwards, we transform in step ② the extracted polyhedral model into an equivalent MDH representation [13] – this transformation is the focus of this paper and discussed in the next section. The MDH representation is suitable for generating portable high-performance code: we use the MDHs’ code generator (MDH-CG) [16] in step ③ to transform the MDH representation into an automatically optimizable (auto-tunable) OpenCL code; the generated code is then auto-tuned in step ④ for different target architectures and input sizes using the Auto-Tuning Framework (ATF) [14, 15]. We execute the automatically generated and auto-tuned OpenCL code in step ⑤ using the dOCAL framework [12, 17].

3 Approach
The focus of this paper is the transformation of the extracted polyhedral model into an equivalent MDH representation (step ② in Figure 1).

In the following, we first briefly recapitulate the definitions of MDHs and their corresponding Domain-Specific Language (DSL) [13]. Afterwards, we demonstrate how the polyhedral model can be transformed into an equivalent expression in the the DSL for MDHs.

3.1 Multi-Dimensional Homomorphism
Multi-Dimensional Homomorphisms (MDHs) are formally defined as follows.

**Definition 3.1.** Let $T$ and $T'$ be two arbitrary data types. A function $h : T [N_1] \times \ldots \times [N_d] \rightarrow T'$ 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 @1, \ldots, @d : T' \times T' \rightarrow T'$, such that for each integer $k \in [1, d]$ and arbitrary, concatenated input array $a \oplus_k b$ in dimension $k$, the homomorphic property is satisfied:

$$h( a \oplus_k b ) = h(a) @_k h(b)$$

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

We express MDHs using their high-level Domain-Specific Language (DSL) [13], as follows. Every MDH $h$ is uniquely determined by its combine operators @$_1, \ldots, @_d$ 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 parallel pattern [13] which takes these functions as parameters:

$$h = md\_hom( f, (@_1, \ldots, @_d) )$$  
$$\quad = @_1 \ldots @_d f( a[i_1| \ldots |i_d] )$$

We demonstrate the usage of md_hom – the basic building block of MDHs’ DSL – based on the example of Matrix Multiplication (MatMul):

$$MatMul = md\_hom( *❦, (+1, +2, +) ) \circ view\_MatMul \quad (1)$$

Formula 1 shows MatMul expressed as an instance of the md_hom parallel pattern. We first fuse the domain-specific input of MatMul – two matrices $A \in T[M][K]$ and $B \in T[K][N]$ of type $T$ (e.g., $T$=float or double) – to a 2-dimensional array comprising pairs of type $T^2$. For this, we use pattern view which MDHs’ DSL provides to uniformly prepare a domain-specific input for md_hom. For MatMul, its view function view_MatMul is defined as: $view(A,B)(i,j,k) = (A[i][k], B[k][j])$; it takes as input the two matrices $A$ and $B$ and the array indices $i, j, k$; it yields the pair $(A[i][k], B[k][j])$. After fusing MatMul’s two input matrices via view_MatMul, we apply MatMul’s scalar function $(\times)$ (multiplication) to each output pair of view_MatMul, and we combine the obtained results in dimension 1 and 2 by concatenation (i.e., @$_1, @_2 = +$), and in dimension 3 by addition (@$_3 = +$).

3.2 Transformation: Polyhedral Model to MDH Representation
We show how the polyhedral model can be transformed into an equivalent MDH representation (step ② in Figure 1) consisting of patterns md_hom and view. For this, for the input parameters of pattern view, we have to extract from the polyhedral model the following information:

1. the input data (e.g., matrices $A$ and $B$ for MatMul);
2. the access indices $(i, j, k$ for MatMul$);
3. the access data $(A[i][k]$ and $B[k][j])$.

For pattern md_hom’s parameters, we need:

4. the scalar function (e.g., $(\times)$);
5. the combine operators (e.g., @$_1, @_2 = +$).

To auto-tune and execute our generated OpenCL code, we need moreover:
6. the data types of the input (e.g., float);
7. the input sizes (e.g., M,N,K for MatMul);

For brevity, we present in this paper our transformation – from the polyhedral model to the MDH representation – only for md_hom’s parameters (points 4. and 5. above), using the simple but important example of MatMul in Listing 1.

```cpp
for( int i = 0; i < M; ++i )
    for( int j = 0; j < N; ++j )
        for( int k = 0; k < K; ++k )
            C[i][j] += A[i][k] * B[k][j];
```

**Listing 1.** Sequential Matrix-Matrix Multiplication in C.

**Scalar Function** Listing 2 shows the already generated scalar function of MatMul’s md_hom expression in Formula 1. The function’s basic building block (line 2) is the loops’ body in line 4 of Listing 1, which we extract straightforwardly from the polyhedral model [23]. We set variables with read or read-write accesses – which are in line 4 of Listing 1: i) read accesses A[i][k] and B[k][j]; ii) read-write access C[i][j] – as the arguments of function f (line 1 in Listing 2); variables with write access – not existent in Listing 1 – would be declared and zero initialized at the beginning of f’s function definition. We return the value of variables with write or read-write accesses at the end of f’s definition (line 3 in Listing 2). Note that MatMul’s scalar function in Listing 2 performs also addition (+) (line 2) and thus differs from the scalar function of MatMul in Formula 1 (which is multiplication * only). This is because in our generated code, we are not able to compute MatMul’s combine operator + (see Formula 1) in parallel, as discussed in the following.

```cpp
T f( T A_i_k, T B_k_j, T C_i_j ) {
  C_i_j = A_i_k * B_k_j;
  return C_i_j;
}
```

**Listing 2.** Scalar function of MatMul.

**Combine Operators** In general, combine operators different from concatenation + (e.g., addition +) cannot be captured in (and thus extracted from) the polyhedral model [4, 18, 19]: automatically identifying such combine operators would require a complicated semantic analysis of the sequential code in Listing 1. We provide two different solutions to circumvent this problem: 1) ignoring the parallelism potential in such dimensions (e.g., as in PPCG); for the dependence analysis, we use isl [22] (in exactly the same way as PPCG); 2) requesting combine operators explicitly from the user; for example, in case of MatMul, the user annotates the code in Listing 1 with the following (OpenMP-like) directive: #mdh parallel (+,++,+:C[i][j]). For a fair comparison with PPCG, we experiment in the next section with solution 1).

**4 Experimental Evaluation**

All our experiments can be reproduced using our artifact implementation [5].

Figure 2 shows the speedup of nd_poly’s automatically generated and optimized code – for benchmarks Gaussian Convolution (left) and Matrix Multiplication (right) from Polybench [11] (for Gaussian, we use the most-recent version in [20]) – over PPCG and hand-optimized vendor libraries (VL). As VLs, we use Intel MKL-DNN [6] and NVIDIA cuDNN [9] for Gaussian Convolution; for Matrix Multiplication, we use Intel MKL [7] and NVIDIA cuBLAS [10]. We experiment on both Intel Xeon E5-2640v2 CPU and NVIDIA V100 GPU. As input sizes, we use i) real-world sizes (abbreviated with RW in the figure) from deep learning, and ii) sizes that are preferable for PPCG (abbreviated with PP). For example, we use for Gaussian a real-world input size of $1\times152\times7\times7\times512$ taken from the deep-learning framework TVM [2], and for Matrix Multiplication, we use input matrices of size $10\times64$ and $64\times500$ which are repeatedly called in the Caffe deep-learning framework [8]. As PP sizes, we use for Gaussian $1\times1\times4096\times4096\times1$ and for Matrix Multiplication, we use square input matrices of sizes $1024$. We auto-tune the programs generated by md_poly and the optimization parameters of PPCG both for 48h – the wall time of our system – using the Auto-Tuning Framework (ATF) [14].

![Figure 2. Speedup (higher is better) of nd_poly’s automatically generated and optimized code over: i) PPCG, and ii) hand-optimized vendor libraries (VL).](image-url)

We observe competitive and often better performance of nd_poly than both PPCG and vendor libraries. As compared to PPCG, nd_poly’s better performance is because our generated OpenCL code has more tunable parameters than PPCG, e.g., parameters for enabling/disabling using OpenCL’s fast local and private memory resources [16]; thereby, we enable a more fine-grained optimization of our generated code. In comparison to vendor libraries, we rely on auto-tuning, while the libraries use hand-crafted heuristics.
References


