md_poly: A Performance-Portable Polyhedral Compiler Based on Multi-Dimensional Homomorphisms

Ari Rasch (Advisor: Sergei Gorlatch)
a.rasch@wwu.de
University of Muenster, Germany

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 [3, 25] 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 formal representation of the code, based on concepts from mathematical geometry, 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 polyhedral 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 achieve 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)* [25] achieves 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 achieve high performance also on its target architecture (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 that generates portable, high-performance code for CPUs and GPUs. For this, `md_poly` combines polyhedral techniques with the algebraic formalism of *Multi-Dimensional Homomorphisms (MDH)* [14, 17] and their code generation approach in OpenCL – the standard for uniformly programming different architectures (e.g., CPU and GPU). We demonstrate that the mathematical program representation of polyhedral compilers (a.k.a. *polyhedral model*) can be automatically transformed into an equivalent MDH representation; this MDH representation is suitable for generating high-performance program code that is performance portable over different architectures [17]. From a theoretical perspective, our findings show that regarding code generation, the recent formalism of MDHs is more expressive than the state-of-the-art polyhedral approach, because the MDH representation captures more information relevant for optimization (as we show in this paper).

Our preliminary experiments with two benchmarks – Gaussian Convolution and Matrix Multiplication – shows encouraging results: speedups up to 7× on Intel CPU and 3× on NVIDIA GPU over the state-of-the-art polyhedral compiler PPCG and hand-optimized vendor libraries on real-world input data from deep learning.

2 Overview

Figure 1 demonstrates the overview of `md_poly`'s internal design. Starting from a sequential C program, we first extract in step ① in the figure the polyhedral model – this is the same step in all C-based polyhedral compilers (e.g., PPCG) – using the *Polyhedral Extraction Tool (*`pet`*)* [24]. Afterwards, we transform in step ② the extracted polyhedral model into an equivalent MDH representation [14] – 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) [17] in step ③ to transform the MDH representation into an automatically optimizable (auto-tunable) OpenCL code – the standard for uniformly programming different architectures (e.g., CPU and GPU); the generated code is then auto-tuned in step ④ for different target architectures and input sizes using our *Auto-Tuning Framework (ATF)* [15, 16]. We execute the automatically generated and auto-tuned OpenCL code in step ⑤ using our dOCAL framework [13, 18].

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) [14]. Afterwards, we demonstrate how the polyhedral model can be transformed into an equivalent expression in the DSL for MDHs.

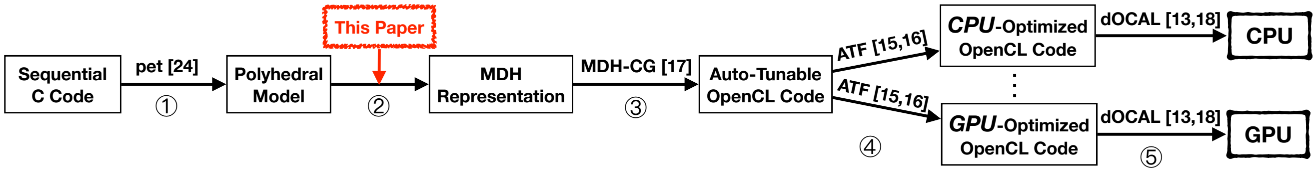

Figure 1. Overview of md_poly's internal design.

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 $h : T[\,N_1\,]\ldots[\,N_d\,] \to T'$ 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* $\circledast_1, \ldots, \circledast_d : T' \times T' \to T'$, such that for each integer $k \in [1, d]$ and arbitrary, concatenated input array $a \mathbin{+\!\!\!+}_k b$ in dimension k, the homomorphic property is satisfied:

$$h(\,a \mathbin{+\!\!\!+}_k b\,) \;=\; h(a) \circledast_k h(b)$$

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

We express MDHs using their high-level Domain-Specific Language (DSL) [14], 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 parallel pattern [14] which takes these functions as parameters:

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

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

$$\mathtt{MatMul} = \mathtt{out\_view\_MatMul} \circ$$
$$\mathtt{md\_hom}(\,*,\, (\mathbin{+\!\!\!+}_1, \mathbin{+\!\!\!+}_2,\, +)\,) \circ \mathtt{in\_view\_MatMul}$$

The formula shows MatMul expressed as an instance of the md_hom parallel pattern. We first fuse the domain-specific input of MatMul – two matrices A∈T[M][K] and B∈T[K][N] of type T (e.g., T=float or double) – to a 3-dimensional array comprising pairs of type $\mathtt{T}^2$. For this, we use pattern in_view which MDHs' DSL provides to uniformly prepare a domain-specific input for md_hom. For MatMul, its view pattern in_view_MatMul is an alias for: in_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 MatMuls's two input matrices via in_view_MatMul, we apply the scalar function f=* (multiplication) of MatMul's md_hom expression to each output pair (A[i][k], B[k][j]) of in_view_MatMul, and we combine the obtained results in dimension 1 and 2 by concatenation (i.e., $\circledast_1, \circledast_2 = \mathbin{+\!\!\!+}$), and in dimension 3 by addition ($\circledast_3 = +$). The results are stored in result matrix C, using out_view_MatMul – an alias for pattern: out_view(C)(i,j) (C[i][j]). The pattern takes the output matrix C and indices i and j; it straightforwardly stores the computed results at position i, j in matrix C at position i, j. In the MDH formalism, output views enable conveniently implementing different variants of computations, e.g., storing the results of computation MatMul as transposed in result matrix C – by only swapping indices i and j when defining the access pattern on output matrix C: out_view(C)(i,j)(C[j][i])

3.2 Transformation: Polyhedral Model to MDH Representation

We show how the polyhedral model can be automatically transformed into an equivalent representation in the MDHs' DSL (step ② in Figure 1) consisting of patterns md_hom, in_view, and out_view.

For brevity, we present in this paper our transformation using an only particular but important example: MatMul as implemented in Listing 1. We will present our general transformation – from the polyhedral model of an arbitrary, sequential C program (i.e., which does not necessarily represent MatMul) to a corresponding MDH representation – in future work.

```
for( int i = 0; i < M; ++i )
  for( int j = 0; j < N; ++j )
  {
    C[i][j] = 0; // Statement 1 (S1)
    for( int k = 0; k < K; ++k )
      C[i][j] += A[i][k] * B[k][j]; // Statement 2 (S2)
  }
```

Listing 1. Sequential Matrix-Matrix Multiplication in C.

3.2.1 Polyhedral Model

We extract the polyhedral model from the sequential implementation of MatMul (Listing 1) straightforwardly using the *Polyhedral Extraction Tool (pet)* [24] (step ① in Figure 1). The model's two basic building blocks are the so-called *iteration* *domain* and *access relations*; we discuss both briefly in the following.

Iteration Domain An iteration domain represents the statements in the sequential C program (in the example of Listing 1, the statements are: `S1` in line 4 and `S2` in line 6). Each statement is dependent on the particular values of the enclosing loop iterators – iterators `i` and `j` in case of statement `S1`, and iterators `i`, `j`, and `k` in case of statement `S2`. Consequently, the iteration domain of `MatMul` is (in polyhedral notation [25]):

$$
\begin{aligned}
(M, N, K) &\rightarrow \{S1(i, j) \quad \mid 0 \le i < M,\ 0 \le j < N \} \\
(M, N, K) &\rightarrow \{S2(i, j, k) \mid 0 \le i < M,\ 0 \le j < N,\ 0 \le k < K \}
\end{aligned}
$$

Access Relation The access relation describes how data is accessed by statements – `read`, `write`, or `read/write`. For example, in case of `MatMul`, arrays `A` and `B` represent the input matrices; both are read only in Listing 1. In contrast, matrix `C` is read as well as written in the listing. Consequently, the access relation of `MatMul` is defined as follows (in polyhedral notation):

`Read` accesses:	$S2(i, j, k) \rightarrow A[i, k]\ ,\ B[k, j]$
`Write` accesses:	void
`Read/Write` accesses :	$S1(i, j) \rightarrow C[i, j]$

3.2.2 MDH Representation

We use the information provided by polyhedral model's iteration domain and access relation to automatically generate a corresponding MDH representation, consisting of patterns `in_view`, `out_view`, and `md_hom` (introduced in Section 3.1).

Input View As input parameters of pattern `in_view`, we have to extract from the polyhedral model the following information:

1. *input data* (matrices A, B, and C for `MatMul`);
2. *access indices* (`i,j,k` for `MatMul`);
3. *accessed data* (`C[ i ][ j ]`, `A[ i ][ k ]`, and `B[ k ][ j ]`).

We can extract parameters 1.-3. from polyhedral model's access relation – all data with `read` or `read/write` accesses are considered as input data.[1]

Output View For pattern `out_view`, we need parameters:

4. *output data* (matrix C in case of `MatMul`);
5. *access indices* (`i,j` for `MatMul`);
6. *accessed data* (`C[ i ][ j ]`).

Analogously as for parameters of pattern `in_view`, we can extract parameters 4.-6. from polyhedral model's access relation; all data with `write` or `read/write` accesses are considered as output data.

Pattern **`md_hom`** As parameters for pattern `md_hom`, we need:

7. *scalar function* `f`;
8. *combine operators* $\circledast_1, \ldots, \circledast_d$;

Scalar Function Listing 2 shows the scalar function `f` of `MatMul`'s `md_hom` expression when generated automatically according to `MatMul`'s polyhedral model. The function's basic building blocks are statements `S1` (line 4) and `S2` (line 6) taken from Listing 1 (lines 4 and 6); we extract both statements from polyhedral model's iteration domain.

We set variables with `read` or `read-write` accesses (for `MatMul`, these are: `A[i][k]`, `B[k][j]`, and `C[i][j]` – see Section 3.2.1) – as the arguments of function `f` (line 2 in Listing 2); variables with `write` access – not existent in the computation of `MatMul` – would be declared and zero initialized at the beginning of scalar function `f`'s definition. We return the value of variables with `write` or `read-write` accesses at the end of `f`'s function definition (line 8 in Listing 2). Polyhedral model's access relation provides all information about variables' access types (see Section 3.2.1) that are required for generating scalar function `f`.

Note that the automatically-generated scalar function `f` of `MatMul`'s `md_hom` expression in Listing 2 is different from the hand-implemented scalar function for `MatMul` in Section 3.1: the automatically-generated function in Listing 2 performs also addition `+` (line 6), while the hand-implemented scalar function in Section 3.1 is only multiplication `*`. This is because combine operators different from concatenation $+\!\!+$ (e.g., combine operator `+`, as in case of `MatMul` – see Formula 1) cannot be extracted automatically from the polyhedral model, as discussed in the following.

```
1 T_OUT f( int i, int j, int k, /* default parameters */
2          T_R_1 A_i_k, T_R_2 B_k_j, T_RW_1 C_i_j ) {
3   if( k==0 )
4     C_i_j = 0; // S1
5
6   C_i_j += A_i_j * B_k_j; // S2
7
8   return C_i_j;
9 }
```

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 [6, 19, 20] – 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 polyhedral tool `isl` [23] (in exactly the same way

[1] Note that matrix `C` is used as both output and also input, because the implementation in Listing 1 performs an additional initialization of matrix `C` (in line 4), which is not expressed in the MDH formula for `MatMul` in Section 3.1 for simplicity.

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 [5]) directive: `#mdh parallel (++,++,+:C[i][j])`. For a fair comparison with PPCG, we experiment in Section 4 with solution *1)*.

Note that MDH approach requires rectangular iteration domains for combine operators [14], e.g., where loops in the sequential C program are incremented by 1 after each loop iteration. If loops' iteration domain is not rectangular, we transform the domain via affine transformation – using `isl` – to an equivalent, rectangular form [1].

3.3 Polyhedral Model vs. MDH Representation

We have shown in the previous subsection that the polyhedral program representation can be automatically transformed into a corresponding MDH representation. For brevity, we have presented this transformation only for `MatMul`; we will present our general transformation – for arbitrary programs (i.e., which might be different from `MatMul`) – in future work.

Compared to a hand-implemented MDH representation, an MDH representation that is automatically generated from the polyhedral model has restrictions: combine operators different from concatenation (e.g., addition `+`, as in case of `MatMul`) cannot be set in the generated MDH representation, because such combine operators are not explicitly represented in the polyhedral model (as discussed in Section 3.2.2). This restriction is an inherent weakness of the polyhedral model, because it inhibits parallelization in non-concatenation dimensions and thus, it can negatively affect performance (as we show in detail in future work).

In contrast, every MDH representation (automatically generated as well as hand implemented) can be automatically transformed into an efficient polyhedral representation, because the MDH representation captures all information represented in the polyhedral model.

Summarizing, the MDH representation expresses more information relevant for high-performance code generation than the state-of-the-art polyhedral model. We will demonstrate this – MDH formalism is more expressive for code generation than the polyhedral model – formally and in more detail in future work.

4 Experimental Evaluation

All our experiments can reproduced using our artifact implementation [2].

To auto-tune and execute our generated OpenCL code (steps ④ and ⑤ in Figure 1), the user passes to `md_poly` conveniently via compiler flags: i) the *data types* of the in- and output (e.g., `float`); ii) the *input sizes* (e.g., `M,N,K` in case of `MatMul`).

Figure 2 shows the speedup of `md_poly`'s automatically generated, optimized, and executed OpenCL code (steps ①-⑤ in Figure 1) – for benchmarks *Gaussian Convolution* (left) and *Matrix Multiplication* (right) (for Gaussian, we implement the most-recent version in [21]) – over PPCG and hand-optimized vendor libraries (VL). As VLs, we use for Gaussian Convolution libraries Intel MKL-DNN [7] and NVIDIA cuDNN [10]; for Matrix Multiplication, we use Intel MKL [8] and NVIDIA cuBLAS [11]. 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 `1×512×7×7×512` taken from the deep-learning framework TVM [4], and for Matrix Multiplication, we use input matrices of size `10x64` and `64x500` which are repeatedly called in the Caffe deep-learning framework [9]. As PP sizes, we use for Gaussian `1x1x4096x4096x1` and for Matrix Multiplication, we use square input matrices of sizes `1024`. We auto-tune both the programs generated by `md_poly` and the optimization parameters of PPCG for 48h – the wall time of our system – using the *Auto-Tuning Framework (ATF)* [15].

		Gaussian Convolution				Matrix Multiplication			
		CPU		GPU		CPU		GPU	
		RW	PP	RW	PP	RW	PP	RW	PP
md_poly	GF/s	155	208	871	4195	22	340	107	9777
	SP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
PPCG	GF/s	20	44	660	3721	11	74	106	9481
	SP	7.78	4.75	1.32	1.13	2.03	4.58	1.01	1.03
VL	GF/s	119	15	738	219	10	466	64	12799
	SP	1.30	14.31	1.18	19.11	2.24	0.73	1.67	0.76

Figure 2. Speedup (higher is better) of `md_poly`'s automatically generated, optimized, and executed code over: i) PPCG, and ii) hand-optimized vendor libraries (VL).

We observe competitive and often better performance of `md_poly` than both PPCG and vendor libraries. As compared to PPCG, `md_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 – this is discussed in detail in [17]; 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.

In future work, we will compare to further polyhedral compilers, e.g., TensorComprehensions [22], and we will present and discuss in more detail the efficiency of `md_poly` over polyhedral compilers for all benchmarks from the popular PolyBench [12] suite which is specifically designed toward performance evaluation of polyhedral compilers.

5 Conclusion

We present `md_poly` – a novel compiler that automatically generates portable high-performance code for CPU and GPU from sequential C programs. For this, `md_poly` combines the recent algebraic formalism of Multi-Dimensional Homomorphisms (MDHs) and their code generation approach in OpenCL with the state-of-the-art polyhedral model – a mathematical program representation based on concepts from mathematical geometry. We show that the polyhedral model can be automatically transformed into a corresponding MDH representation which is amenable for generating high-performance code for different architectures. From a theoretical perspective, our findings show that the MDH formalism is more expressive than the polyhedral approach regarding code generation, because the MDH representation captures more information relevant for optimization.

Our preliminary experiments demonstrate that `md_poly` achieves better performance than both: i) state-of-the-art polyhedral compiler PPCG – by relying on a more fine-grained auto-tuning process; ii) hand-optimized vendor libraries (such as Intel MKL and NVIDIA cuBLAS) – by relying on auto-tuning rather than hand-crafted heuristics.

References

[1] 2019. isl Development Forum. https://groups.google.com/forum/#!topic/isl-development/bscRqEc2_Xg

[2] Artifact Implementation. 2020. https://gitlab.com/mdh-project/src_grand_finals.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices* 43, 6 (2008), 101–113. https://doi.org/10.1145/1379022.1375595

[4] 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

[5] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998), 46–55.

[6] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's polyhedral scheduling in the presence of reductions. *arXiv preprint arXiv:1505.07716* (2015).

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

[8] Intel. 2019. Math Kernel Library. https://software.intel.com/en-us/mkl

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia (MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[10] NVIDIA. 2018. cuDNN library. https://developer.nvidia.com/cudnn

[11] NVIDIA. 2019. cuBLAS library. https://developer.nvidia.com/cublas

[12] Louis-Noel Pouchet. 2015. PolyBench/C: the Polyhedral Benchmark Suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[13] Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2019. dOCAL: high-level distributed programming with OpenCL and CUDA. *The Journal of Supercomputing* (30 Mar 2019). https://doi.org/10.1007/s11227-019-02829-2

[14] Ari Rasch and Sergei Gorlatch. 2018. Multi-dimensional Homomorphisms and Their Implementation in OpenCL. *International Journal of Parallel Programming* 46, 1 (01 Feb 2018), 101–119. https://doi.org/10.1007/s10766-017-0508-z

[15] Ari Rasch and Sergei Gorlatch. 2019. ATF: A generic directive-based auto-tuning framework. *Concurrency and Computation: Practice and Experience* 31, 5 (2019), e4423. https://doi.org/10.1002/cpe.4423 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4423 e4423 cpe.4423.

[16] A. Rasch, M. Haidl, and S. Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 64–71.

[17] A. Rasch, R. Schulze, and S. 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

[18] A. Rasch, M. Wrodarczyk, R. Schulze, and S. Gorlatch. 2018. OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 408–416. https://doi.org/10.1109/PADSW.2018.8644541

[19] Chandan Reddy, Michael Kruse, and Albert Cohen. 2016. Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/2967938.2967950

[20] Xavier Redon and Paul Feautrier. 1993. Detection of recurrences in sequential programs with loops. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 132–145.

[21] Philippe Tillet and David Cox. 2017. Input-aware Auto-tuning of Compute-bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 43, 12 pages. https://doi.org/10.1145/3126908.3126939

[22] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[23] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.

[24] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France. http://impact.gforge.inria.fr/impact2012/workshop_IMPACT/verdoolaege.pdf

[25] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4 (January 2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713