Scheduling Language Chronology: Past, Present, and Future

MARY HALL, University of Utah, United States
COSMIN E. OANCEA, University of Copenhagen, Denmark
ANNE C. ELSTER, Norwegian University of Science and Technology, Norway and Univ. of Texas at Austin, USA
ARI RASCH, University of Muenster, Germany
SAMEERAN JOSHI, University of Utah, United States
AMIR MOHAMMAD TAVAKKOLI, University of Utah, United States
RICHARD SCHULZE, University of Muenster, Germany

Scheduling languages express to a compiler—or equivalently, a code generator—a sequence of optimizations to apply. Performance tools that support a scheduling language interface allow exploration of optimizations, *i.e., exploratory compilers.* While scheduling languages have become a common feature of tools for experts, the proliferation of these languages without unifying common features may be confusing to users. Moreover, we recognize a need to organize the compiler developer community around common exploratory compiler infrastructure, and future advances to address, for example, data layout and data movement. To support a broader set of users may require raising the level of abstraction. This article provides a chronology of scheduling languages, discussing their origins in iterative compilation and autotuning, noting the common features that are used in existing frameworks, and calling for changes to increase their utility and portability.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Computing methodologies** → **Parallel programming languages;**

Additional Key Words and Phrases: Scheduling languages, compilers, domain-specific languages, parallel, distributed

ACM Reference Format:
Mary Hall, Cosmin E. Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. 2025. Scheduling Language Chronology: Past, Present, and Future. *ACM Trans. Arch. Code Optim.* 22, 3, Article 100 (September 2025), 31 pages. https://doi.org/10.1145/3743135

This work was partially funded by the *Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)* – project *MDH.DL* (470527619), the US National Science Foundation project CCF-2107556, the Novo Nordisk Foundation (project NNF24OC0090447) and SFI-CGF through Research Council of Norway Project no. 309960.
Authors' Contact Information: Mary Hall, University of Utah, Salt Lake City, United States; e-mail: mhall@cs.utah.edu; Cosmin E. Oancea, University of Copenhagen, Copenhagen, Denmark; e-mail: cosmin.oancea@di.ku.dk; Anne C. Elster, Norwegian University of Science and Technology, Trondheim, Norway and Univ. of Texas at Austin, Austin, USA; e-mail: elster@ntnu.no; Ari Rasch, University of Muenster, Münster, Germany; e-mail: a.rasch@uni-muenster.de; Sameeran Joshi, University of Utah, Salt Lake City, United States; e-mail: sameeran.joshi@utah.edu; Amir Mohammad Tavakkoli, University of Utah, Salt Lake City, United States; e-mail: amir.tavakkoli@utah.edu; Richard Schulze, University of Muenster, Münster, Germany; e-mail: r.schulze@uni-muenster.de.

ACM 1544-3973/2025/09-ART100
https://doi.org/10.1145/3743135

1 Introduction

The primary contribution of this article is a chronology of scheduling languages that illustrates how past work motivated and led to the present proliferation of scheduling languages, and how future improvements— aimed at easing the interaction with the domain expert and at supporting more general forms of computations—may cycle back to the past, prompted by the need for higher automation and integration. Table 1 summarizes the key properties of this evolution, which is highlighted in this section and expanded in the rest of the article.

Until the late 1990s, compilers were essentially black boxes that were controlled via optimization flags and a small set of directives to analysis and optimization. Even today, large open source community compiler projects like LLVM and gcc are organized as a series of passes over a common **intermediate representation** (**IR**) that has been lowered from source code, with each pass leaving the code in a consistent state. Therefore, the internals of each pass—including the decision algorithms that apply optimizations—are opaque to compiler users and even most compiler developers. At the end of the compilation process, code is lowered from the IR to architecture-specific machine code.

In the 1990s as compiler research introduced optimizations to achieve locality in caches (and vector and thread-level parallelism), something that is still true today became obvious: *it is difficult to predict the best sequence of code transformations to achieve high performance since it depends heavily on both architecture and input data.* Many sophisticated cache models were developed during this time to predict capacity and conflict misses (e.g., interference phenomena [135] and cache miss equations [54]) to guide architecture-specific optimization. However, as the complexity of these models grew, the alternative idea of simply executing the code to measure its performance on each platform, and adjust the algorithms accordingly, took hold. *Autotuning*, popularized by ATLAS (Automatically Tuned Linear Algebra) [37, 153] developed a tuning scheme for BLAS (Basic Linear Algebra) routines and a few LAPACK routines; ATLAS initially focused on cache and instruction-level parallelism optimizations. Other similar approaches arose in the late 1990s and early 2000s for autotuning computation kernels from specific domains, including linear algebra [20], sparse linear algebra [71, 152], bit reversal [45, 129], and FFTs [53, 92, 108].

These works on self-tuning libraries were complemented by programming languages and compilers designed to facilitate exploration of more general computations. Contemporaneously with ATLAS, early work in *iterative compilation* developed compiler technology that enabled exploration of code transformations [23, 78, 103, 107, 140]. Using iterative compilation, the compiler's internal algorithms were designed to explore alternative sequences of transformations, execute the code on the target hardware, and use that empirical data to decide which version provided the best performance. Iterative compilation led to the reorganization of compilers to support exploration. The search over transformation sequences was part of the compiler's algorithms, so it was still the compilers' responsibility to decide *what to explore*.

Subsequent research empowered expert programmers to collaborate with programming languages and compilers by exposing the optimization process to user control. On the programming language side, a number of systems allowed programmers to express alternative *code variants* — functionally equivalent implementations, potentially using different algorithms—and dynamically identify the most performant composition of variants [7, 50]. As well, Tao analysis [106] promoted a methodology for characterizing algorithms that informs parallelization strategies. On the compiler and code generation front, a number of systems supported the specification by an expert programmer of a sequence of transformations to apply to a code, expressed within the code using annotations or in a separate transformation recipe [31, 43, 56, 67, 157]. This direction was also explored in the functional context by allowing users to express code transformations as rewrite rules [74].

In the context of **domain-specific languages** (**DSLs**), Halide [109] popularized the idea of deriving a high-performance implementation by combining: (1) a simple, high-level specification, aimed to increase productivity of domain experts; together with, (2) a transformation recipe written by a compiler expert in a separate language—named a *scheduling* language.

Unlike standard interfaces to compiler optimizations – compile-time flags and pragmas inserted in the code – schedules are *programs* that express a sequence of optimizations to apply to a separate source code. Compilers that support a scheduling language interface allow exploration of compiler optimizations, and are referred to subsequently as *exploratory compilers*.

A scheduling language serves as an expert programmer's interface to the compiler's transformations, or as an abstraction that exposes transformations to automated search and prediction. Over the last decade, scheduling languages have been increasingly used to map performance-critical computations within domain-specific commercial applications. In principle, the use of a language to represent a schedule strengthens the optimization description by providing well-defined semantics, composability, and the opportunity for verification. In practice, today's scheduling languages lack many of these desirable properties of a programming language [62].

In this article, we consider the suitability of scheduling languages as a key abstraction in current and future compilers for achieving high performance on critical computations. While scheduling languages were originally designed to expose optimizations to experts because automation was too difficult, we envision a future where scheduling languages are part of any architecture-specific optimization workflow. This exploratory compiler structure exposes profitable code transformation sequences at a finer granularity than tuning based on compiler flags, and is externally controllable in contrast to iterative compilation. Moreover, a human-readable language facilitates improving and sharing schedules. Therefore, in the two decades since their origins, by using scheduling languages coupled with systematic approaches for exploring schedules, compilers have been restructured to provide the abstractions needed to close the circle and automate high-performance code generation.

The structure of this article mirrors Table 1. In the next three sections, we characterize work related to this theme, choosing papers from contemporary surveys [9, 13, 130] and papers that introduced new approaches that help categorize the solution space. Section 2 surveys past approaches (1997 – 2012) that were essentially aimed at optimizing scientific applications, beyond what the heuristic pipelines of general-purpose compilers could provide. They include self-tuning libraries, configurable programming languages, and advanced compilation techniques, such as iterative compilation, which are applied to mainstream languages and are characterized by a high degree of automation and integration. Section 3 surveys present work (2013–2023) using Halide as the starting point. These systems are predominantly domain specific, driven by the observation that *specialization* of the language and compiler gives rise to performance. Section 4 envisions a future that aims to broaden the specification language and the repertoire of code transformations available to scheduling, while at the same time interacting directly with the domain expert in simple(r) ways. The future likely requires a high degree of automation, reminiscent of the approaches of the past. Section 5 summarizes this article and future work.

2 Past: Optimization Exploration

Early exploratory optimization systems arose from the growing complexity of emerging architectures and the productivity challenge of producing architecture-specific code. These trends demanded that even expert programmers needed tools to accelerate exploration of different implementation strategies. This section reviews the origins of scheduling languages by discussing three distinct bodies of work: domain-specific autotuning libraries and code generators; iterative

Table 1. Motivations and Resulting Technology Related to Scheduling Languages in the Past, Present, and Expected Future

Timeline	Motivation	Focus	Approaches
PAST 1997 - 2012 *Explore*	Heuristic-based code optimization decisions ineffective Improve efficiency of expert users	Primarily loop nest computations Embedded & scientific applications & libraries	Self-tuning libraries Exploratory compilers and code generators Rewriting rules & lang. support for code variants
PRESENT 2013 - 2023 *Specialize*	Efficiency of expert users in specific domains	Domain specific languages and compilers	Separate high-level specification and schedule Narrow the search space to utilize autotuning and ML
FUTURE 2024 - *Popularize*	Increase user accessibility Raise abstraction	Broaden to more general applications Compose/incorporate into common infrastructures	Data layout/movement integration Runtime support Expand search space while maintaining practicality

compilation and early autotuning compilers and code generators; and programming languages that permitted exploration of optimizations and algorithms.

2.1 Early Domain-Specific Self-Tuning Library Generators

Self-tuning libraries are examples of early efforts to improve performance automatically (without programmer interactions) across different architectures. These approaches decompose the given functions into performant subprograms, e.g., customized to the cache hierarchy of a given architecture, its register file size, and instruction set. Subprogram sizes are parameterized and the parameters that offer the best performance are determined empirically; thus, *autotuning* for a given architecture. Some of the earliest efforts include PhiPAC, ATLAS, and **Fastest Fourier Transform in the West** (**FFTW**). We refer to these as self-tuning since the optimization search is built into the library.

PhiPAC [20] was a multi-level cache-blocked matrix multiply autotuning generator that was a precursor to ATLAS. ATLAS (Automatically Tuned Linear Algebra) [37, 153], focused on BLAS (Basic Linear Algebra) routines such as matrix-matrix multiplication and a few LAPACK routines. Autotuned compilation approaches combined with a small hand tuned assembly kernel have also been shown to beat both ATLAS and vendor-optimized libaries [73]. Since BLAS are performance-critical building blocks, the predominant hardware vendors offer BLAS implementations that are not only autotuned to their platforms, but also employ empirical based planners and optimizers, e.g., Intel's MKL (Math Kernel Library) and NVIDIA's cuBLAS.

Other efforts developed domain/algorithm specific libraries: such as for bit reversal algorithms [45, 129], sparse linear algebra [71, 152], and FFTs. For example, the FFTW library [53] takes advantage of the recursive nature of the FFT algorithm where smaller FFTs can be used as building blocks for larger FFTs. FFTW thus uses a high-level description execution plan for decomposing larger Fourier transforms into smaller, specialized kernels named "codelets". It then uses a dynamic programming-based search process at runtime, when the input transform size is known, to find the best execution plan. Similar techniques are used in cuBLAS. The Strandh-Elster bit-reversal library [45, 129] uses an empirical approach based on a linear bit-reversal algorithm [44], whereas the Spiral [92, 108] system applies to more general signal processing with high-level tensor notations and genetic algorithms-based search.

2.2 Exploratory Compiler and Code Generation Technology

Classical compilation approaches automatically generate low-level parallel code according to a set of internal optimization heuristics. In particular, polyhedral compilers such as Pluto [24] and PPCG [148] have demonstrated that efficient parallelization of (at least) affine programs is possible for multi-core hardware. In turn, such analyses are facilitated by the development of analytic cost models for locality of reference, capable of deriving, for example, asymptotic lower bounds at the level of misses in a set-associative cache hierarchy [54, 135]. Such heuristic-based approaches were the norm for locality-optimizing and parallelizing compilers, intended to be a productive way to protect the programmer from having to make complex optimization decisions. However, as previously noted, the payoff for optimizations is impacted by *execution context*; i.e., architecture details and input datasets. Heuristics fail to accurately capture this context, leading to performance loss. In this section, we describe exploratory compiler approaches designed to evaluate optimizations within their execution context.

2.2.1 Iterative Compilation. Concurrent with the emergence of domain-specific autotuning libraries, *iterative compilation* was developed as part of the OCEANS compiler project [1]. Early work from this project by Bodin et al. describes the use of an iterative algorithm [23] that applied tiling, unrolling and padding to matrix multiply, and then searched among a fixed set of tile and unroll sizes. The portability of the search algorithm is demonstrated on three target architectures (Ultrasparc, Pentium Pro, and embedded VLIW Trimedia TM1000). At that time, the high search cost of iterative compilation limited its use to embedded applications, where the assumption was that the application would be compiled once or infrequently and run repeatedly to amortize the search cost. Kisuki et al. present a variety of search space algorithms and limits on searching tile and unroll sizes to make it practical for general-purpose optimization [78]. As compared to these small, fixed search spaces for tiling and unrolling, Pouchet et al. developed an iterative compilation approach that explored different valid multi-dimensional schedules using the polyhedral model, which facilitates correct composition of a sequence of iteration space transformations [107]. Other work explored a large collection of compiler passes. The ***optimization-space exploration (OSE)*** compiler focused on VLIW optimizations for the Itanium processor, including VLIW-specific optimizations, standard loop transformations, and compiler flags [140]. Park et al. developed a prediction modeling technique called *tournament predictor* to discover optimization sequences that outperformed -Ofast and other predictors using the Open64 compiler [103].

A defining feature of iterative compilation research is that the search strategy was integrated into the compiler's algorithms and not intended to be accessible to application developers. Triantafyllis et al. refer to the configuration of the iterative optimization algorithm as happening at *compiler construction time* [140].

2.2.2 Autotuning Compilers and Code Generators. A few years later, motivated by the high performance enabled by autotuning libraries and iterative compilation, emerging compilers and code generators made it possible for expert users to control the sequence of transformations applied to a computation. The precursors to today's scheduling languages were focused on more general-purpose computation, typically limited to loop nests, where parallelizing compiler technology could be applied.

```
// code, named loops, xforms
#pragma xlang name iloop
for (i=0; i<NB; i++)
  #pragma xlang name jloop
  for (j=0; j< NB; j++)
    #pragma xlang name kloop
    for (k=0; k<NB; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
#pragma xlang transform stripmine iloop NU NUloop
#pragma xlang transform stripmine jloop MU MUloop
#pragma xlang transform interchange jloop NUloop
#pragma xlang transform interchange kloop NUloop
#pragma xlang transform fullunroll NUloop
#pragma xlang transform fullunroll MUloop
#pragma xlang transform scalarize_in b in kloop
#pragma xlang transform scalarize_in a in kloop
#pragma xlang transform scalarize_in&out c in kloop
#pragma xlang transform lift kloop.stores after kloop
```

(a) Xlang

```
// code
DO J=1,N
  DO K=1,N
    DO I=1,N
      C(I,J)=C(I,J)+
          A(I,K)*B(K, J)

// CHiLL
// transformation recipe
permute([3,1,2])
tile(0,2,TJ)
tile(0,2,TI)
tile(0,5,TK)
datacopy(0,3,2,[1])
datacopy(0,4,3)
unroll(0,4,UI)
unroll(0,5,UJ)
```

(b) CHiLL

Fig. 1. Examples of expressing locality optimizations for matrix multiply in Xlang[43] and CHILL[31].

Expressing Transformations as an Interface to Compilers. An initial question was how to express the optimizations to be applied. Initially, this approach permitted user access to compiler algorithms, and the scheduling language was primarily used to perform transformations on loop nest computations.

The X Language expresses a sequence of transformations as pragmas in the source code [43], as in Figure 1(a). An advantage of using pragmas is that the source code and optimization strategy are self-contained in a single file. However, a disadvantage is that only a single optimization strategy is supported. The X Language also made it possible to define new transformations in the compiler so as to add capability to the language.

CHiLL [31, 66, 139] in Figure 1(b) and **URUK** (**Unified Representation Universal Kernel**) [56] supported a separate script that provided a sequence of transformations, each designating the source code statement to which the transformation should be applied along with parameters to the compiler's optimization. The separate script has the advantage of supporting different optimization strategies or even architectures.

Figure 1 shows how transformations are expressed in the X language and in CHiLL for matrix multiply. In the X language, the loops are named so that transformations can use the names to designate where transformations are applied. As new loops are created, such as with the `stripmine` transformation, they too are named. In CHiLL's separate script, the `permute` command reorders the loop nest so that the I loop is outermost and the K loop is innermost, resulting in the dependence on C being carried by the innermost loop. For all but `permute`, the first parameter of each transformation is the statement to which the transformation should be applied. The matrix multiply has a single statement, so it is always 0. The second parameter is the loop, followed by any parameters to the transformation, e.g., the tile size or unroll factor. Note that after the first `tile` command, a tile controlling loop is added in the outermost position (loop level 1). Thus, in the next `tile` command, loop level 2 refers to the I loop, which was previously in the outermost position. Both the X Language and the CHiLL versions shown here are tiling i and j loops and unrolling

```
/*@ begin Loop (
transform UnrollJam(ufactor=Ui)
for (i=0; i<=M-1; i++)
  transform UnrollJam(ufactor=Uj)
  for (j=0; j<=N-1; j++)
    transform UnrollJam(ufactor=Uk)
    for (k=0; k<=O-1; k++)
      A[i][j] += B[i][k]*C[k][j];
) @*/
for (i=0; i<=M-1; i++)
 for (j=0; j<=N-1; j++)
  for (k=0; k<=O-1; k++)
    A[i][j] += B[i][k]*C[k][j];
/*@ end @*/
```

```
def performance_params {
  param Ui[] = range(1,33);
  param Uj[] = range(1,33);
  param Uk[] = range(1,33);
 constraint reg_capacity = Ui*Uj+Ui*Uk+Uk*Uj<=32;
}
def input_params {
  param M[] = [10,50,100,500,1000];
  param N[] = [10,50,100,500,1000];
  param O[] = [10,50,100,500,1000];
 constraint square_matrices = (M==N) and (N==O);
}
```

(a) ORIO

```
<define loopJ Loop#("j",0,"n",1)>
<define loopI Loop#("i",0,"m",1)>
<define loopK Loop#("k",0,"l",1)>
<define mmStmt "c[i+j*m]+= alpha*b[k+j*l] * a[i+m*k];">
<define nest1 Nest#(loopK,mmStmt)>
<define nest2 Nest#(loopI,nest1)>
<define nest3 Nest#(loopJ,nest2)>
<define mmHead "void dgemm(int m, int n, int l, double alpha, double *a, double *b, double *c)">
<define dgemm Function#(mmHead, "int i, j, k;", nest3)>
<define mm_block_unroll (Unroll#(mm_block, InnermostLoop#(mm_block)))>
<output mm_block_unroll.c (Block.bsize=(16 16 8);Unroll.ur=8; mm_block_unroll)>
```

(b) POET

Fig. 2. Examples of expressing matrix multiply in ORIO [67] and POET [157].

some or all of the inner loop tiles. CHiLL additionally tiles the k loop, and performs a datacopy of tiles of a and b into buffers to eliminate conflict misses in cache.

All three frameworks—X Language, URUK, and CHiLL—exposed interfaces designed for expert users with knowledge about compiler transformations and abstractions.

Expressing Transformations as an Interface to Code Generators. Around the same time, code generators that emit specific code combined with existing templates were developed to apply transformations to code, typically integrated with autotuning. Figure 2 shows early examples of this approach as applied to matrix multiply, ORIO [67] and POET [157]. Both examples apply unroll-and-jam for the loop nest computation. On the right side of Figure 2(a), the inputs to the autotuner are provided, which include a series of problem sizes and unroll factors. In both systems, it is possible to define new transformations by describing how they modify the code. For example, in Figure 2(b) `mm_block_unroll` combines blocking (i.e., tiling) and unrolling.

2.3 Language-Compiler Codesign

Complementary to exploratory compilers, this section provides a brief overview of a related strand of research that refers to enhancing the language with more powerful constructs as a way of lifting the level of abstraction at which the compiler reasons.

2.3.1 Decomposition and Algorithmic Variants. Sequoia [50] was one of the first works that proposed a specification language in which task decomposition is programmed explicitly—but generically in terms of array sizes—and a scheduling language that specializes the task to the particularities of the hardware by mapping the user-defined decomposition at each level of the

```
1  void task matmul::inner( in    float A[M][P]
2                         , in    float B[P][N]
3                         , inout float C[M][N] ){
4    // tunable parameters specify the
5    // size of subblocks of A, B, C
6    tunable int U, X, V;
7
8    // Partition matrices into sets of blocks
9    blkset Ablks = rchop(A, U, X);
10   blkset Bblks = rchop(B, X, V);
11   blkset Cblks = rchop(C, U, V);
12
13   // Compute all blocks of C in parallel
14   mappar(int i=0 to M/U, int j = 0 to N/V) {
15     mapreduce (int k=0 to P/X) {
16         matmul( Ablks[i][k]    // recursive
17               , Bblks[k][j]    // invocation
18               , Cblks[i][j] ); // on subblocks
19     }
20   }
21 }
22
23 void task matmul::leaf( in    float A[M][P]
24                       , in    float B[P][N]
25                       , inout float C[M][N] ){
26   for (int i=0; i<M; i++)
27     for (int j=0; j<N; j++)
28       for (int k=0; k<P; k++)
29         C[i][j] += A[i][k] * B[k][j];
30 }
```

(a) Algorithmic Specification

```
1  instance {
2    name     = matmul_cluster_inst
3    task     = matmul
4    variant  = inner
5    run_at   = cluster_level
6    calls    = matmul_node_inst
7    tunable U = 1024, X = 1024, V = 1024
8    A distribution = 2D block-block
9                (blocksize 1024x1024) ... }
10 instance {
11   name     = matmul_node_inst
12   task     = matmul
13   variant  = inner
14   run_at   = node_level
15   calls    = matmul_L2_inst
16   tunable U = 128, X = 128, V = 128    }
17 instance {
18   name     = matmul_L2_inst
19   task     = matmul
20   variant  = inner
21   run_at   = L2_cache_level
22   calls    = matmul_L1_inst
23   tunable U = 32, X = 32, V = 32
24   subtask arg A = copy
25   subtask arg B = copy                 }
26 instance {
27   name     = matmul_L1_inst
28   task     = matmul
29   variant  = leaf
30   run_at   = L1_cache_level            }
```

(b) Instantiation to a Cluster Hardware.

Fig. 3. Running example taken from Sequoia paper [50]: specification (left) and schedule (right).

memory hierarchy. Figure 3 shows the specification and schedule of the running example of the original article. The specification of a Sequoia task consists of:

— recursive definitions (variant `inner` at lines 1 – 21) that apply blocking primitives to generically-sized, multi-dimensional arrays (lines 9 – 11) together with recursive calls to sub-blocks from inside map-reduce constructs (lines 16 – 18);
— a base-case definition (`leaf` at lines 23 – 30) that expresses a sequentially-efficient implementation;
— algorithmic variants are, in principle, supported at both levels;
— communication is only possible between parent and child tasks using call-by-value-result semantics.

While the schedule in Figure 3(b) essentially performs tiling at each level of the memory hierarchy, the work on Sequoia has brought forth (at least) two key ideas:

(1) A slight strengthening of the language—e.g., mostly side-effect free specification, isolation of communication and delegating the task decomposition to the user—may simplify the compiler's reasoning and may result in simple implementations that offer competitive performance with state-of-the-art libraries. Many of the present DSLs borrow similar ideas from the data flow (or functional) context.
(2) Scheduling each level of the memory hierarchy allows to make explicit data-layout optimizations—e.g., lines 24 – 25 in Figure 3(b) specify that arguments `A` and `B` of `matmul::leaf` are remapped in contiguous storage so as to ensure stride-1 accesses (and

lines 8 – 9 specify the data-distribution policy). This is even more relevant today, e.g., because GPUs offer programmable memories and specialized execution units.

A significant body of work was aimed at composing algorithmic variants, notably PetaBricks [7] and previously-discussed FFT work. For example, PetaBricks demonstrates that the Poisson solver, symmetric eigenproblem and sorting can be efficiently implemented each from three algorithmic variants. Finally, Tao analysis [106] proposes a data-centric methodology for programmers to characterize algorithms based on their topological structure, the ways in which nodes become active, and the kinds of computation at node level (e.g., updates performed at local, neighborhood, or topology level). This decomposition informs suitable parallelization schedules and is demonstrated using Galois [82] on regular and irregular algorithms, such as graph analytics and finite-element mesh generation and refinement.

2.3.2 Rewrite-Rule Systems in Purely-Functional Languages. Purely-functional languages primarily aim to provide a programming environment that allows the implementation to match as closely as possible the algorithmic specification. The support for higher abstraction comes at the cost of (high) runtime overhead, which would be prohibitively expensive unless it is statically optimized. However, the low-level loop optimizations developed in the imperative context are not applicable here because, for example, recurrences are expressed in terms of recursive functions (rather than loops) and type abstraction requires aggressive boxing, which results in heavy use of indirection (pointers). Instead, one of the directions taken has been to exploit the richer semantics of higher-order operators by making their *algorithmic* properties available to the compiler under the form of rewrite rules. The key difference between rewrite rules and affine transformations is that the former can express identities that cannot be derived by reordering the statements of the original code pattern.

An example rewrite is the rule [22] that famously states that a segmented scan [21] (prefix sum) with an arbitrary associative operator—i.e., scanning in parallel each subarray of an irregular array of arrays–can be rewritten as a scan with a lifted operator that is applied to the array of tuples obtained by zipping the flattened-data array with a flag array that encodes with 1 the start of each subarray (and 0 otherwise).

A large body of work [25, 40, 41, 88, 150] has studied in the functional context how to allow a library writer to extend the compiler by means of rewrite rules that encode domain-specific optimizations. The Haskell GHC compiler employs such a mechanism that has passed the test of time and differentiates itself from related approaches by means of its practical simplicity [74]. Rewrite rules are written in source-to-source Haskell, which enables simple pattern matching and a simple rewrite strategy.

An example rewrite rule (used in [74]) is shown at the bottom left of Figure 4. It consists of (1) a name `‘‘foldr/build’’`, (2) a `forall` clause that declares which of the variables used in the rule are universally quantified, and (3) a body in which the left-hand side must be the application of a function that is not one of the quantified variables (`foldr` in our case). When the rule fires, the left-hand side is always replaced with the right-hand side; it is the user's responsibility to ensure that (1) the rule is correct, (2) the right-hand side is more efficient than the left-hand side, and that (3) the system of rewrites terminates.

The rule uses the higher-order functions `foldr` and `build`. `foldr` is conventionally defined, i.e., it has the semantics $\textbf{foldr} \odot z\, [x_1, \ldots, x_n] = x_1 \odot (x_2 \odot \ldots (x_n \odot z))$. `build` takes as a parameter a functional representation of a list `g` — that abstracts over its cons and nil constructors — and it applies `g` to ordinary list constructors `:` and `[]`. The rule states that the cases when `foldr` consumes the application of `build` to `g` as its third argument can be re-written by applying `g` directly. The

```
foldr :: (α → β → β) → β → [α] → β
foldr k z [] = z
foldr k z (x:xs) = f x (foldr f z xs)

build :: (forall b. (a → b → b) → b → b)
        → [a]
build g = g (:) []

{-# RULES
"foldr/build"
forall k z.(g::forall b.(a→b→b) → b → b)
foldr k z (build g) = g k z
#-}
```

```
sum :: [Int] → Int     -- sum [5,4,3,2,1] = 15
sum xs = foldr (+) 0 xs
down :: Int → [Int]    -- down 5 = [5,4,3,2,1]
down v = build (λ n → down' v n)
down' 0 cons nil = nil
down' v cons nil = cons v (down' (v-1) cons nil)

sum (down 5)
⇒ -- inlining sum and down
  foldr (+) 0 (build (down' 5))
⇒ -- applying foldr/build rewrite
  down' 5 (+) 0
  -- computes 5 + (4 + (3 + (2 + (1 + 0))))
```

Fig. 4. Example of Haskell Rewrite Rule from [74]: Shortcut deforestation rule (left) and its application (right).

right-hand side of Figure 4 demonstrates how this rule eliminates the creation of the intermediate list [1, 2, 3, 4, 5].

Essentially, GHC performs the standard transformations—such as beta reduction, inlining, case switching, let floating, case swapping, and elimination—and relies heavily on the library writer to directly communicate the smarts to the compiler. Common examples include fusion rules for user-defined ADTs (e.g, rose trees) and specialization rules for (common cases of) overloaded instances, where special attention is dedicated to eliminating the overheads of maintaining a modular programming style. Notably, the compiler also automatically generates and then applies rewrite rules, typically pertaining to specializations.

3 Present: What's in a Schedule?

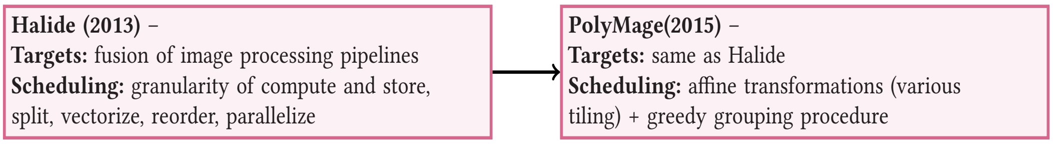
The chronology categorizes the present as beginning with Halide [109], which as noted in the introduction played an important role in popularizing scheduling languages. The present, which represents a decade of progress, has coincided with the recognition that *specialization gives rise to efficiency*, leading to a proliferation of domain-specific hardware and software tools. Consequently, we organize this section into a set of key domains where scheduling languages were paramount to unlocking high performance. Of note, the systems described here increasingly target mainstream commercial applications, while past approaches of Section 2 were developed in the context of scientific computing or embedded systems.

The presentation is organized as follows: The first five subsections discuss bodies of work aimed at specific application domains, namely image processing, tensor algebra, deep learning, graph processing, and more general flavors of data-parallel computations; they are accompanied by figures summarizing how the discussed central works have influenced each other. Finally, Section 3.6 discusses the integration of scheduling languages with performance tuning, and Section 3.7 reviews improvements to general-purpose compiler infrastructure.

3.1 Image Processing Approaches (Summarized in Figure 5)

Halide [109] is one of the first works to pragmatically combine the strengths of the functional and imperative approaches in the context of a DSL that is primarily aimed at image-processing pipelines. The key idea in Halide is to separate the algorithm's expression from the optimization concerns by using:

— a simple and pure data-flow specification that is implicitly parallel and accessible to the domain expert,
— an optimization recipe that is written by the compiler expert or is derived by autotuning.



Halide (2013) –
Targets: fusion of image processing pipelines
Scheduling: granularity of compute and store, split, vectorize, reorder, parallelize

PolyMage(2015) –
Targets: same as Halide
Scheduling: affine transformations (various tiling) + greedy grouping procedure

Fig. 5. Overview of two of the central works related to DSL scheduling languages targeting image processing.

```
UniformImage in(Uint(8), 2)
Var x, y
Func blurx(x,y) = in(x-1,y)
                + in(x,y)
                + in(x+1,y)
Func out(x,y) = blurx(x,y-1)
              + blurx(x,y)
              + blurx(x,y+1)
```

(a) Data-flow specification of a 3×3 un-normalized box filter

```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_o
       compute at out.y_i
out: split x by 4 → x_o, x_i
     split y by 4 → y_o, y_i
     reorder: y_o, x_o, y_i, x_i
     parallelize: y_o
     vectorize:   x_i
```

(b) Optimization Recipe (i.e., Schedule Specification)

Fig. 6. Halide running example [109].

Figure 6(a) shows the expression of a 3×3 un-normalized box filter, in which arrays (e.g., `blurx`, `out`)[1] are represented as index functions (i.e., from coordinates to values), hence their elements can be safely computed in parallel. In this context, the optimization of highest impact is *stencil fusion*, which can be realized by a combination of tiling, sliding-window and work replication strategies.[2] The best combination is, however, sensitive to the hardware and dataset characteristics. As such, the optimization recipe answers the questions: (1) at which granularity to *compute* and (2) at which granularity to *store* each of the intermediate arrays, and, within those grains, (3) in what order/fashion should the array domain be traversed?[3] For example, the optimization recipe in Figure 6(b) declares a schedule that combines the tiling and sliding window optimizations:

— `out` is tiled with tiles of size 4, creating (reordered) dimensions of indices y_o, x_o, y_i, x_i, from which y_o, x_o are parallelized, y_i is sequentialized (to implement a sliding window) and x_i is vectorized.
— `blurx` is *stored* at the level of loop y_o in `out` but is *computed* at a finer granularity (y_i) — hence the domain of `blurx` under y_i only has dimension x_i, which is vectorized for performance.

The legality of the transformations is enabled by the pure (and implicitly parallel) semantics of the array computations, e.g., tiling is legal because parallel loops are always safe to be interchanged inwards.

In summary, Halide has demonstrated that high-performance implementations of image-processing pipelines can be derived by means of a simple and clean DSL specification in conjunction with (1) either an optimization recipe written by the compiler expert or with (2) extensive autotuning. Initially, its autotunner used a genetic algorithm, but its slow convergence motivated the switch to the more robust OpenTuner framework [8]; this, however, still required hours-to-days to find the optimal solution for deep pipelines.

[1] `blurx` and `out` compute the horizontal and isotropic blur by averaging over 3×1 and 1×3 windows, respectively.
[2] These techniques cover a tradeoff space along three axes: the degree of locality, of exploited parallelism, and of redundant computation.
[3] For example, dimensions can be strip-mined, reordered, traversed sequentially, or in parallel (vectorization included).

PolyMage's approach [93] was (partly) motivated by the observation that even though the schedule space is vast,[4] only a limited subset of that space matters in practice. As such, PolyMage renounced the optimization recipe in favor of a *greedy grouping procedure*, which aggressively fuses computation until a maximal threshold of redundant computation is reached. Since the grouping procedure is parameteric in terms of tile and threshold sizes, the autotuning is simplified to explore a small space consisting of (three) threshold values and (seven) tile sizes per tiled dimension. This procedure is reported to find in up to one hour a near-optimal schedule for multi-core execution that offers performance competitive to code written/optimized by experts. Finally, PolyMage demonstrated that polyhedral reasoning can be applied in a DSL context to elegantly model:

- overlapped tiling—which, in principle, falls outside polyhedral scheduling since it introduces redundant computation (i.e., does not preserve the statements of the original program), and
- other classical tiling strategies—such as parallelogram, hexagonal, or split tiling—which were not feasible to be expressed in Halide.

Since then, a rich body of work has been aimed at improving Halide in various ways, for example (1) by adapting the scheduling strategy of PolyMage to generate competitive Halide schedules [94], (2) by extending Halide to generate code for Specialized **Digital Signal Processors** (**DSPs**) compilers [151], (3) by providing support for automatic computation of gradients [87], (4) by adding new optimization strategies to maximize producer-consumer locality [126], (5) by supporting automatic generation of GPU schedules [125], and (6) by refining the search algorithm to quickly produce high-quality schedules [3, 6].

A variety of image processing DSLs, compilation techniques, and autotuners have been developed independent of Halide and PolyMage. Modesto [58] (CPU+GPU), Absinthe [59] (CPU), and Stencil-Gen [115] (GPU) rely on analytical models to determine the optimal schedule among a multitude of variants generated using various tiling strategies (including overlapped tiling with streaming), storage optimizations, and (greedy) fusion heuristics. Another body of work refers to applying dynamic analysis to optimize stencils: OPS [118] uses a combination of delayed execution and dependence analysis to resolve at runtime hindrances to static analysis that typically occur in large applications, and ARTEMIS [116] uses bottleneck analysis via runtime profiling to guide the application of optimizations, and the tuning of various code generation parameters. ImageCL [47–49] autotunes by using ML for performance optimizations on CPUs/GPUs, as well as for load-balancing several GPUs on a node or GPUs on a cluster. Other work explores (1) scalable and adaptive autotuning frameworks for stencil computations [133], and (2) efficient mapping of image processing pipelines to FPGA hardware [35, 69].

3.2 Tensor Algebra Approaches (Summarized in Figure 7)

Tensor algebra describes an algebra applied to multi-dimensional tensors of any rank, with multiplication used to compute tensor products. Tensor contraction refers to multiplying elements of two tensors to produce a third tensor; a contraction dimension results in a summation of the products of the two tensors along that dimension. In this section, we separate dense and sparse tensor algebra, as the expression and optimization of them varies significantly. While dense tensor algebra exhibits high arithmetic intensity and demands optimizations to manage the memory hierarchy and parallelism, sparse tensor algebra suffers from being memory bound and involves a significantly more complex code generation process.

Dense Tensor Algebra. Beyond the various strategies for implementing matrix multiply, we focus this section on support for tensor contraction for higher-dimensional tensors. An early

[4]The schedule (search) space is exponential in the depth of the pipeline, i.e., the number of pipeline stages.

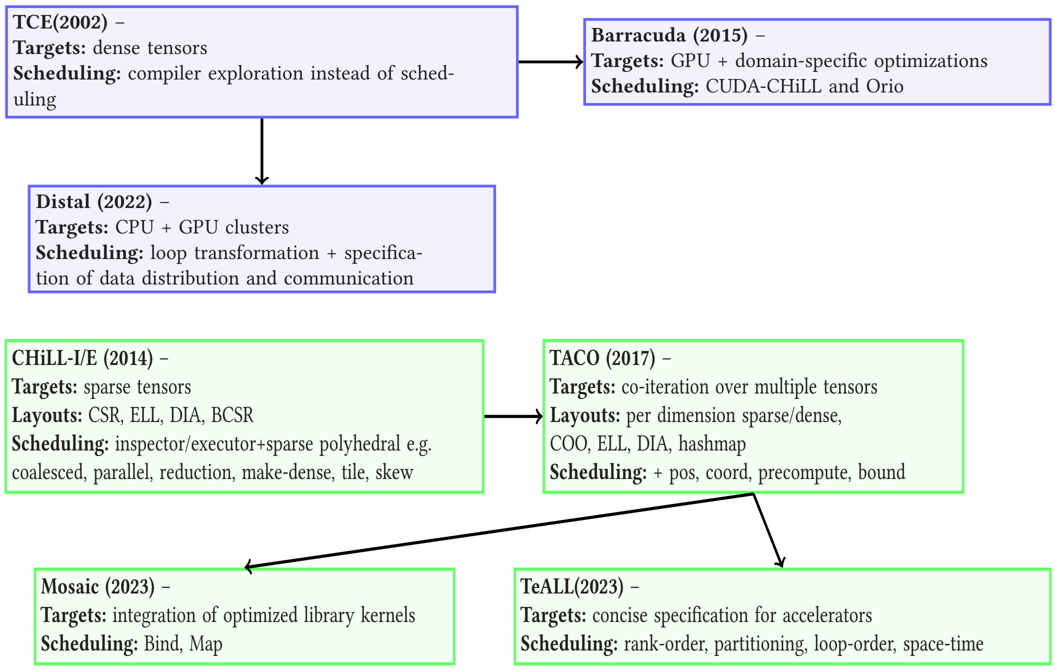

TCE(2002) –
Targets: dense tensors
Scheduling: compiler exploration instead of scheduling
Barracuda (2015) –
Targets: GPU + domain-specific optimizations
Scheduling: CUDA-CHiLL and Orio
Distal (2022) –
Targets: CPU + GPU clusters
Scheduling: loop transformation + specification of data distribution and communication
CHiLL-I/E (2014) –
Targets: sparse tensors
Layouts: CSR, ELL, DIA, BCSR
Scheduling: inspector/executor+sparse polyhedral e.g. coalesced, parallel, reduction, make-dense, tile, skew
TACO (2017) –
Targets: co-iteration over multiple tensors
Layouts: per dimension sparse/dense, COO, ELL, DIA, hashmap
Scheduling: + pos, coord, precompute, bound
Mosaic (2023) –
Targets: integration of optimized library kernels
Scheduling: Bind, Map
TeALL(2023) –
Targets: concise specification for accelerators
Scheduling: rank-order, partitioning, loop-order, space-time

Fig. 7. Overview of some central works related to scheduling languages for dense (blue) and sparse tensor algebra (green).

domain-specific system for tensor algebra was the **Tensor Contraction Engine (TCE)** [16, 38], designed for a class of quantum chemistry computations. These are characterized by contractions over 4-dimensional tensors with hundreds of terms. The computations are both compute-intensive, and access a large volume of data. The order in which the tensor contraction is performed impacts the overall amount of computation. Moreover, with so many terms with different dimension orders, many address streams are touched during the computation. These challenges are addressed using enumeration of feasible solutions that fit in memory, minimization of total computation, and fusion heuristics to limit the set of implementations explored [16, 38]. While work on TCE predates the use of scheduling languages in compilers, it employs search and search space pruning to explore a prohibitively large set of feasible implementations.

Barracuda [100] is a more recent work on 4-dimensional tensor contraction, optimizing quantum chemistry kernels and nuclear fusion computations on GPUs. The approach uses exploratory compilation, from mathematical description to optimized CUDA code as output. Starting with a high-level tensor input language, the approach combines tensor-specific mathematical transformations with a GPU decision algorithm, and autotuning of a large parameter space using a random forest search algorithm. Internally, this implementation uses CUDA-CHiLL's scheduling language, and Orio to manage the search algorithm. A key heuristic to limit the search space is to choose loop orders that match memory layout order for at least one of the tensors in the computation, thus limiting data movement.

DISTAL represents the state-of-the-art in scheduling languages for tensor algebra, targeting CPU and GPU clusters [156]. The scheduling language in DISTAL includes loop transformations to organize the computation as well as communication specification across distributed nodes. Moreover, a separate format description permits data layout within a node and distributed across nodes.

This approach makes it possible to specify common tensor algebra algorithms such as Cannon and COSMA at a high level, and automate their generation to optimized code.

Fireiron [60] introduces a scheduling language that is fully focused on optimizing dense matrix multiplication on NVIDIA GPUs. For this, Fireiron introduces scheduling primitives to target domain-specific hardware extensions of NVIDIA GPUs, namely Tensor Cores which compute matrix multiplication of small 4×4 matrices immediately in hardware, thereby enabling high performance potentials. However, while Fireiron achieves impressive performance results for its particular target domain - multiplication of dense matrices on NVIDIA GPUs – it cannot be used for other computations and/or architectures, which limits its applicability.

There are also heuristic-based tensor DSLs such as TC [142, 143] and FlexTensor [162], and work on heuristics for polyhedral compilers [149] has continued, as well.

Sparse Tensor Algebra. Sparse tensor algebra performs the same operations as dense tensor algebra, but the underlying data representation is fundamentally different. When many entries of a tensor are zero, sparse tensor representations only store nonzero values, to (1) reduce the size of data; (2) avoid unnecessary computation such as multiplying by zero or adding zero; and (3) reduce data movement through the memory hierarchy. Auxiliary data structures provide a mapping of nonzero values to their logical indices in a dense matrix. This physical-to-logical mapping makes it possible for sparse tensor code generators to perform the portions of a computation on the nonzero values.

As compared to dense tensor algebra, we observe that scheduling languages for sparse tensor algebra have several unique capabilities. First, they support a variety of sparse data representations – hereafter referred to as *data layouts* – since proper code generation must be customized to a layout. Moreover, loop optimizations must be reformulated whenever loop indices iterate over a sparse dimension of a tensor. They may also require constructs that utilize information only available at runtime into the optimization. These points will be illustrated by examples from sparse tensor algebra systems that employ scheduling languages.

CHiLL-I/E extended CHiLL's transformations and associated scheduling commands to support sparse linear algebra computations [75, 144, 145, 147]. CHiLL was extended to incorporate concepts from the sparse polyhedral framework [132], such as uninterpreted functions representing index arrays. CHiLL-I/E was able to convert sparse matrix representations using an *inspector/executor paradigm*, whereby a one-time inspector at runtime converted the matrix from a standard representation to an optimized one, and the executor was then able to generate optimized GPU or CPU code. The scheduling language constructs added to CHiLL included standard transformations such as *coalesce* (also called collapse and flatten), *parallel* and *reduction* [147]. In addition, the *make-dense* transformation designated a sparse dimension as being dense as a way for the compiler to reason about subsequent transformations. These included a sequence of standard transformations such as *tile* and *skew*. Ultimately, any *make-dense* was proceeded by *compact*, or *compact-and-pad* transformations, which generate both an inspector for format conversion and an executor for parallel execution on a GPU. This work enabled conversion from CSR format to common formats ELL, DIA and BCSR [144]. Uninterpreted functions were also used to incorporate dynamic parallel wavefronts [145].

While CHiLL-I/E focused on a single sparse tensor, **TACO** [79] introduced an approach to co-iteration over multiple sparse tensors, where the intersection (for multiply) or the union (for addition) of the nonzero locations must be identified. The user specifies the layout along with the computation in Einstein notation, and the compiler generates the code for the input with the specified layout. Originally, TACO represented data layouts by marking dimensions as being either *dense* or *compressed*, where compressed dimensions only represented nonzero values; a similar

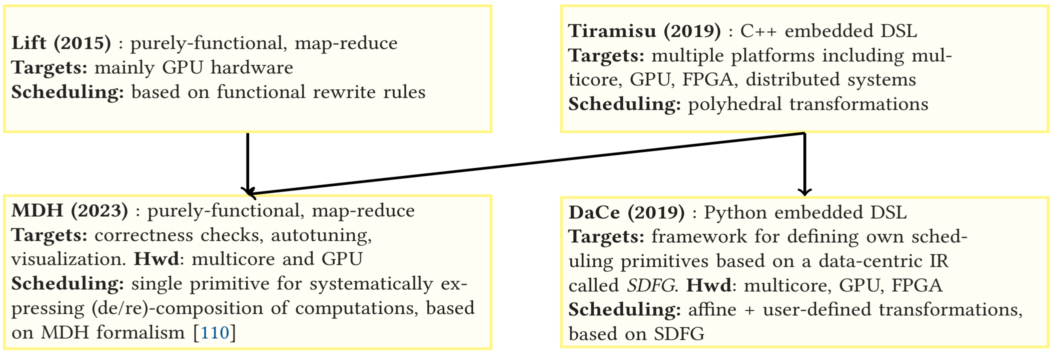

Lift (2015) : purely-functional, map-reduce
Targets: mainly GPU hardware
Scheduling: based on functional rewrite rules
Tiramisu (2019) : C++ embedded DSL
Targets: multiple platforms including multicore, GPU, FPGA, distributed systems
Scheduling: polyhedral transformations
MDH (2023) : purely-functional, map-reduce
Targets: correctness checks, autotuning, visualization. **Hwd**: multicore and GPU
Scheduling: single primitive for systematically expressing (de/re)-composition of computations, based on MDH formalism [110]
DaCe (2019) : Python embedded DSL
Targets: framework for defining own scheduling primitives based on a data-centric IR called *SDFG*. **Hwd**: multicore, GPU, FPGA
Scheduling: affine + user-defined transformations, based on SDFG

Fig. 8. Overview of some of the central works focusing on data parallel approaches and how they are related.

approach was integrated into the MLIR compiler [19]. Subsequent extensions to TACO incorporated *singleton*, *range*, *offset*, and *hash* to support other common sparse tensor layouts that are the higher-dimensional analogs of sparse matrix representations COO, ELL, and DIA, as well as a hashmap [34]. Ref. [5] extends the layout specification to describe regular and irregular patterns. To improve the performance and take advantage of the optimized data layouts, code transformations were later enabled in TACO through a scheduling language [122]. Key transformations are *split* (i.e., strip-mine), and *collapse* (also called coalesce or flatten), *reorder* (i.e., permute), *unroll* and *parallelize*. The transformations *pos* and *coord* enable optimizations to be applied to the position (or physical) space, and the coordinate (or logical) space. The transformation *precompute* permits subarrays to be computed in scratchpad memories, and *bound* introduces constants used by code generation. The transformations are applied on an iteration graph IR before sparse code generation.

Two systems for sparse tensor computations extend the use of scheduling languages in unique ways. **Mosaic**[14] extends TACO's scheduling language to combine code generation and optimization with integration with optimized library kernels. For this purpose, Mosaic introduces two essential scheduling commands, `Bind`, which indicates that a statement should be replaced by a function call, and `Map`, which provides partial automation of schedule generation. The **Tensor Algebra Accelerator Language** (**TeAAL**) uses a scheduling language to enable precise and concise specification of sparse tensor algebra accelerators, and inspiration comes from dense tensor algebra accelerator design[99]. Examples of scheduling language primitives include rank-order, partitioning, loop-order, and spacetime.

3.3 Data-Parallel Computations (Summarized in Figure 8)

Lift [127], **Tiramisu** [12], **Locus** [137], **DaCe** [17], and **MDH** [112] address a more general flavor of computations that include image processing and tensor kernels, but also more exotic computations, such as ***Probabilistic Record Linkage (PRL)*** [111], different kinds of *Stencil* [63], and climate modeling [18] applications.

Lift [127] (recently renamed as *RISE* [128]) proposes a restricted purely-functional language, together with a repertoire of code transformations that are expressed as rewrite rules (e.g., map, fusion, fission, stripmining). Since the optimization space is huge, the rewrites are either directly specified by the user or are guided by procedures that use stochastic search or equality saturation [80]. More recent work focuses on the design of a strategy language, named *ELEVATE* [62], that aims to allow programmers to define new optimization strategies in a composable and reusable way (and is inspired by the work discussed in Section 2.3.2). Lift mainly targets performance portability across GPU hardware. For example, extending Lift with `slide` and `pad` primitives (and associated rewrites) allows efficient computation of individual stencils [63] (e.g., by overlapped tiling), and extension with *macro rules* targets matrix multiplication [119].

Tiramisu [12] is a scheduling framework for a C++ embedded DSL. In contrast to Lift's functional rewrite rules, Tiramisu uses polyhedral transformations, rooted in dependence analysis on arrays. Similar to Halide, the algorithm specification is separated by the implementation details (hardware, iteration space, and other optimizations). Unlike Halide, which uses an interval-based representation of iteration spaces, Tiramisu opts for a more mathematically powerful polyhedral representation that supports composition of a complex sequence of affine transformations on dense multi-dimensional spaces (i.e., loop nests). Notably, Tiramisu uses a four-level IR that addresses separation of concerns without unnecessarily barring optimizations: The first level refers to the algorithmic specification, the second to the ordering of computations, the third to the management of data (i.e., storage location on device and layout), and the fourth to the communication between execution nodes. Tiramisu supports multiple backends, including multicore X86 CPUs, NVIDIA GPUs (both leveraging LLVM), Xilinx FPGAs (Vivado HLS) and distributed machines (MPI), and reports competitive performance with cuBLAS, cuDNN, Intel OneAPI, and other specialized libraries on image stencils, recurrent neural network (dense and sparse), and other non-rectangular iterations spaces. Recent work [11] reports a learning-based cost model for automatic code optimization.

Locus [137] is a scheduling framework that is aimed at orchestrating the optimization of legacy code written directly in mainstream languages, such as C, C++, Fortran, instead of a (restricted) DSL. The work addresses the challenges of manipulating large programs written in complex languages, in particular related to expressing clearly and concisely complex collections of transformations—rooted in dependence analysis on arrays—that are applied to (different) code regions, as selected by the programmer. Locus supports a number of optimization modules off the shelf, as well as procedures for automatically searching the space of code variants.

DaCe [17] is a framework that supports a collection of APIs for implementing optimization workflows. *The key idea is that the performance engineer uses the APIs to write their own DSLs and optimization pipelines, tailored to the target application*, albeit DaCe comes already equipped with a collection of transformations and optimization passes. The most important API is the **Stateful DataFlow multiGraph** (**SDFG**), which implements the IR and facilitates the *construction* of SDFGs by means of frontends for several language subsets. Other APIs refer to transformations on SDFG graphs (e.g., to develop optimization passes and tuners) and code generation APIs, which handle the mapping to different architectures, such as multicore, GPU, and FPGA.

Scheduling optimizations are expressed either on (1) the structural representation, e.g., tiling is represented as nested Map scopes; or, (2) the attributed representation, e.g., "schedule" attributes are attached to scope nodes to indicate OpenMP scheduling, GPU thread blocks, and so on, and "storage" attributes are attached to data nodes to indicate the memory type (CPU heap, GPU global, shared or register memory). For code transformation, DaCe uses a combination of polyhedral and rewrite rules on graphs. Recent works [18, 141] have reported an autotuning framework that automatically explores the space of some common polyhedral transformations by combining machine learning and performance profiling techniques.

MDH [112] is a recent approach that has a particular focus on a structured-language design. It is grounded in the algebraic formalism of ***Multi-Dimensional Homomorphisms (MDH)*** [110], and it is aimed to systematically express (de/re)-compositions of computations at each level of the memory and core hierarchies of parallel architectures. The claimed advantages are twofold: *First*, MDH

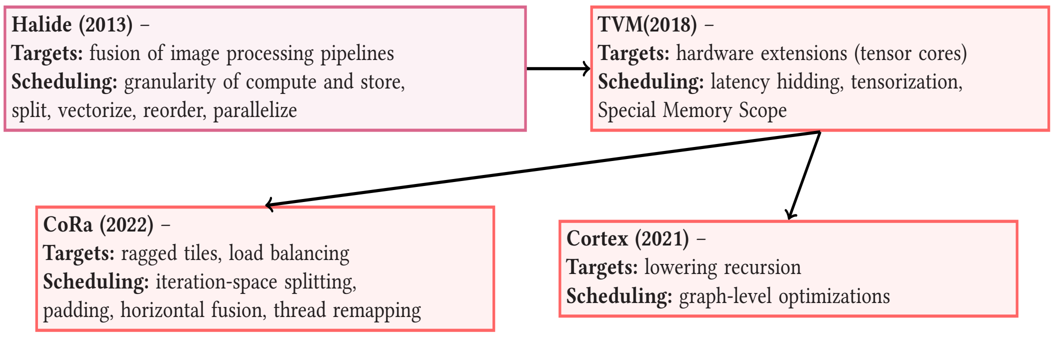

Halide (2013) –
Targets: fusion of image processing pipelines
Scheduling: granularity of compute and store, split, vectorize, reorder, parallelize
TVM(2018) –
Targets: hardware extensions (tensor cores)
Scheduling: latency hidding, tensorization, Special Memory Scope
CoRa (2022) –
Targets: ragged tiles, load balancing
Scheduling: iteration-space splitting, padding, horizontal fusion, thread remapping
Cortex (2021) –
Targets: lowering recursion
Scheduling: graph-level optimizations

Fig. 9. Overview of some of the central works related to scheduling languages targeting Deep Learning.

catches a multitude of user errors[5] and it issues precise error messages for the invalid schedules. *Second*, MDH optionally allows to leave unspecified any arbitrary optimization decision, which would then be solved by its internal autotuning engine (including automatic generation of entire device- and data-optimized schedules), thereby promoting user productivity and performance portability. Similar to Lift and Tiramisu, MDH allows autotuning of straightforward optimization parameters such as tile sizes, but also supports more advanced exploration, e.g., related to layout transformations and memory placement (register/private/shared/global). The automatically generated schedules have often been found to offer performance competitive with vendor libraries [112] and can also be used as a starting point for manual fine-tuning by a performance expert. In terms of limitations, MDH's scheduling language is designed to express optimizations at a high abstraction level and is thus unable to express low-level code optimizations (such as loop unrolling, which is handled in MDH internally by heuristics).

3.4 Deep Learning Approaches (Summarized in Figure 9)

Scheduling languages are a common feature in systems focused on code generation for deep learning workloads, supporting computations such as linear algebra, convolutions, and computations for reshaping and splitting data. Given the large body of recent work in this area, we focus our discussion here on **TVM**, which represents the current state-of-the-art in CPU and GPU mapping of deep learning workloads [32]. TVM achieves performance close to hand-optimized vendor libraries. TVM's scheduling language augments Halide schedule constructs with architecture-specific optimizations, e.g., applying to TPUs, NVIDIA Tensor Cores, and memory hierarchies. Extensions to TVM's scheduling language were added for **CoRa** [52], which supports ragged tiles resulting from problem sizes that are not divisible by the tile size. These include iteration space splitting, padding, horizontal fusion across operators, and thread remapping for load balance. **Cortex** provides additional extensions in a dynamic scheduling language to address how recursion is lowered [51]. Only a limited set of optimizations can be tuned in TVM's scheduling language, and more advanced optimizations require the code generation to be tunable (e.g., based on *Ansor* [161] discussed in Section 3.7). TVM supports *graph-level optimizations* across deep learning operators, particularly fusion of adjacent operators that do not exhibit data dependences. Graph-level optimizations are not accessible from the scheduling language, but are instead performed upon the IR. In contrast, operator fusion is implemented within a scheduling language as a loop transformation, as in the

[5] In addition to the errors caught by polyhedral approaches, such as specification of invalid tiling, MDH detects more sophisticated errors, e.g., when the user tries to combine the results of *CUDA Thread Blocks* across invalid memory regions, as per CUDA specification.

SWIRL system for wide SIMD CPUs [146]. Finally, a survey of techniques for accelerating DL models is given in [10]. It covers directions related to (i) "hardware-aware neural architecture search", which automates the **neural-network** (**NN**) design through a multi-objective optimization of accuracy and hardware efficiency, (ii) "automatic code optimization", which refers to searching the most efficient schedule for a given hardware and (iii) combinations of the two.

3.5 Graphs

Graph algorithms require a unique set of transformations that reflect structural changes and how to parallelize the execution of graphs. Galois [82, 86] takes a data-centric view of parallelism where the Parallel program is viewed as Operator + Schedule + Parallel Data Structures. For Galois, the schedule refers to constructs that relax the order in which graph vertices are processed, later extended with data placement annotations for scalability. Galois [101] demonstrated that it could serve as a backend for other vertex-centric graph DSLs PowerGraph [57], GraphChi [84], and Ligra [124], improving their performance and scalability.

More recently, GraphIt [158] exposes a scheduling language and a separate algorithm language describing operators on vertices and edges. The scheduling language has representations for edge traversal, vertex data layout, and structure optimizations. The schedules provide various functions to configure edge traversal direction, parallelization, data layout, and data placement optimizations. It also offers the unique opportunity for structural changes to the graph, such as fusing adjacent vertices into a single vertex.

3.6 Improvements to Heuristic-Based Compilers

The appearance of scheduling languages was counterbalanced by improvements to classical compiler approaches that used new ways to incorporate execution context into the optimization process. This includes work aimed at (1) optimizing synchronization in task-parallel programs, (2) analytical models for determining near-optimal tile sizes or data layout, (3) IR manipulation techniques promoting quick integration of new code transformations, (4) compiler exploration techniques that search a sweet point between the utilization of locality and parallelism.

Task-Parallel Languages. Task-parallel languages, such as X10 [28] and **HJ** (**Habanero-Java**) [27] employ non-trivial runtime systems to support constructs that subsume various kinds of synchronization, such as `async` and `finish` in X10 and HJ, `foreach` and `ateach` loops in X10, `forall` and `foreach` loops in HJ. Since such synchronizations incur significant runtime overheads, compiler work [97] was aimed at expanding the classical (static) loop transformations—such as loop interchange, distribution, fusion, (un)switching, unpeeling—to operate in the presence of synchronization primitives. Furthermore, composing rich subsets of these transformations in predefined ways has been demonstrated to result in high-impact program-level optimizations.

Analytical Modeling. Advances in analytical modeling, aimed, e.g., at selecting near-optimal tile sizes and data layout at small compile-time overhead, have the potential of enabling the adoption of the effective scheduling transformations studied in the (restricted) context of DSLs inside general-purpose compiler infrastructure. For example, in the context of CPU+GPU execution, static analysis [90] has been devised to compute the best layout in between **array of structures** (**AoS**) and **structure of arrays** (**SoA**) at the program level, in a way that takes into account the host-device transfers and the overhead of remapping to respective layouts. Related to tile-size selection, a proposed lightweight technique [98] (that builds on Wolf and Lam's approach [154]) models both spatial and temporal locality of a loop nest of depth n by computing a core tile size as the solution of a polynomial of degree n in one unknown. The tile sizes for each loop are then determined by multiplying the core tile size by a weight that models the amount of reuse of that loop. The solution accounts for prefetching, load balancing and vectorization, and is demonstrated on benchmarks from image processing, DSP, and linear algebra; in particular it extends PolyImage to cover applications from the linear algebra domain. Other work models and solves tile-size selection as **integer linear programming** (**ILP**) problems [2], and proposes affine scheduling frameworks that promises to unify and co-optimizes code reordering and data-layout transformations [123].

IR manipulation. IR-manipulation techniques allow compiler experts to quickly implement new code transformations. For example, in the context of multi-core execution, "Declarative Loop Tactics" [30] proposes a matcher-builder style for manipulating the tree-of-bands polyhedral IR, which is also extended with a node that represents change-of-layout transformations. Such approaches build on previous work in imperative—see for example the POET [157] and OSE [140] compilers reviewed in Sections 2.2.1 and 2.2.2—and functional languages, which use inference (rewrite) rules systems to implement code transformations, as detailed in Section 2.3.2.

Compiler Exploration. Exploration techniques are applied in a range of contexts ranging from purely functional to polyhedral. The main aim is to adapt the compilation strategy to the hardware and datset characteristics, which commonly refers to finding the right balance between exploiting (enough) locality and parallelism.

An instance of compiler exploration is implemented in the purely-functional array language Futhark: The application parallelism is mapped to the hardware by an "incremental flattening" procedure [68] that utilizes map fission and map-loop interchange to create code variants that systematically map more and more levels of application parallelism to the hardware hierarchy. The variants are independently optimized and combined into one program by branching on predicates that compare a dynamic program measure with a threshold. The best combination of code variants is derived by autotuning the threshold values [55], which is implemented by a deterministic procedure that is optimal as long as the dynamic measure conforms to a monotonic property [95]. Futhark does not support a scheduling language because, for example, program-level transformations such as automatic differentiation [26, 121] produces code that cannot be "reasoned", hence "scheduled", by the user.

An instance of polyhedral exploration is proposed in [29], in which a set of schedules for a target computation is automatically generated using a *performance lexicon* [81], i.e., a collection of ILP objectives based on dependencies, levels of parallelism, and other statically-derived properties. The performance lexicons allow (i) to prune the candidate schedules that do no exhibit one/any of the desired traits, and (ii) to navigate the legal space of transformations in a controlled fashion by testing only a small combination of objectives and by quantifying their impact by means of analytic modeling (i.e., without actual runs). In essence, this approach is intended as an adaptive scheduling scheme that bridges the gap between general-purpose and domain-specific compilation. Finally, a class of compilers, such as CLBlast [102] and MDH [110], do not rely on schedules, but instead expose their optimization spaces in the form of parameter spaces, which further enables automatic exploration of such spaces by means of auto-tuning frameworks [8, 113].

3.7 Integrating Scheduling Languages with Performance Tuning

Systems that use scheduling languages employ different techniques for deriving the optimized sequence of transformations described by the scheduling language. Here, we use the phrase *defining the optimization search space* to refer to the process of how possible optimization sequences are determined and specified. A *point in the search space* is then an instantiated schedule that can be provided to the system to generate code. These search spaces are typically prohibitively large, and it is impractical to evaluate each point in the search space. Therefore, various techniques are needed to limit the search space and sample a subset of points, which we refer to as *search space* *pruning*. A discussion of tools that collaborate with scheduling languages to explore a search space of code variants follows.

Autotuning. Autotuning is frequently used to measure the execution time of an optimized code variant running on a specific hardware platform. As previously discussed, autotuning systems arose because of the increasing challenge of deriving accurate performance models as architectural complexity has exploded. A very common approach to defining the optimization search space is to use manually-written scheduling templates, written by performance experts, with variables embedded in each template that are placeholders for parameters in the search space; e.g., in frameworks for deep learning such as AutoTVM [33] and SWIRL [146]. In domain-specific systems, where the set of possible optimizations is well understood, this approach can be effective because the search space can be kept narrow.

In other systems, the search space is generated by an algorithmic procedure, using various techniques to limit the size of the search space. CUDA-CHiLL [76] avoids combinatoric growth in the search space by separating the exploration of data placement in the GPU memory hierarchy (global/shared/texture memories or registers) from tuning optimization parameters (thread, block, tile and unroll factor sizes), treating them as independent variables. Proptuner [65] uses Monte Carlo Tree Search to look ahead when generating and evaluating complete schedules. FlexTensor [162] generates schedules by enumerating different combinations, based on a specific ordering within the schedule space. Ansor [161] uses evolutionary search and learned cost model to find optimized TVM schedules. This approach is superior to the approach of AutoTVM since it extends to optimizing operators not available as templates and captures complex optimization patterns.

For sparse tensor computations, Ahrens et al. [4] co-optimize the computation and format of a sparse tensor. This system outputs a schedule by ranking and filtering by asymptotic complexities and runtime, which reduces the scheduling space by orders of magnitude and generate kernels which perform asymptotically better than the default TACO schedules. WACO [155] co-optimization considers both sparse formats and sparse schedules together. A sparse CNN model extracts the feature set from sparse programs, and a superschedule explores the combined search space using the **Approximate Nearest Neighbour Search** (**ANNS**).

Predictive Models. An alternative approach to conventional autotuning is to construct a predictive model using deep learning: a learning phase derives an associated cost with different schedules and inputs, and inference performs a model lookup at runtime. While a search is still needed to build the model, an accurate model can be reused, and leverages the large investment in deep learning systems to make this efficient. Early work by Park et al. developed a tournament predictor model for iterative compilation [103]. ImageCL [47, 48] uses a predictive model to autogenerate schedules that optimize CUDA codes targeting stencils for image processing, and has been extended to a more general framework, AUMA, targeting stencils using OpenCL [49] as well as 3D adaptive mesh refinement [120]. Ongoing related work includes Judas[46].

4 Future: Call to Action

The story portrayed so far shows a past in which performance critical applications were predominantly optimized with general-compiler infrastructure operating on mainstream languages and a present in which we have amassed significant expertise in extracting near-optimal performance for specific computational kernels from various domains—e.g., image processing, tensor algebra—by using scheduling languages in conjunction with very restricted programming models (languages) and specialized compiler infrastructure.

However, at its core, the compiler pipeline employed by these systems consists of a combination of well-known code transformations along with architecture-specific optimizations. A natural next step might be to cycle back to the past by gradually lifting the language restrictions, composing and integrating scheduling languages, and extending their repertoire with more dynamic analyses and support for specifying data layout and movement. Since writing schedules is challenging, we envision that solutions might: raise the level of abstraction of the scheduling language such that it becomes accessible to domain experts; and/or rely on exploratory compilers that automate schedule generation and selection, while still allowing the user to provide key optimization insights. This section predicts several emerging technologies that would benefit scheduling languages in the quest of advancing the holy grail of automating high-performance code generation.

4.1 Composing and Integrating Scheduling Languages

We have discussed a number of separate programming system implementations, each with its own scheduling language. To accelerate the reach and impact of scheduling languages, we believe schedules should be integrated into mainstream programming language and compiler implementations. As this article has described constructs at different levels of abstraction, from algorithms to low-level hardware control, an integrated approach would need to compose and search across schedules at these different layers of abstraction.

We consider here the alternative strategies for mainstream integration of schedules. First, annotation systems such as OpenMP [36] expose a pragma interface that permits programmers to express thread-level parallelism at a high level, while relying on the underlying compiler to implement the details of the parallelization. In recent years, OpenMP pragmas describe loop transformations (currently *tile*, and *unroll*). The strength of standardization is that a committee reviews the language extensions. However, a weakness is that many of the users of scheduling languages are not part of the high-performance computing community, from which OpenMP arises, and that it takes a long time to add support for a new transformation.

An approach with a potentially broader reach is to integrate the scheduling language into a widely-used open source compiler infrastructure, particularly one designed to support domain-specific systems. The MLIR compiler [85] provides this capability in the form of the *transform dialect* [89]. The transform dialect approach is distinguished from other systems in this article where a schedule is customized to a particular computation. Instead, the use of the *pdl dialect* to identify applicability of transforms makes the schedules independent of the computation, more like an optional compiler pass, and facilitates "building libraries of composed compiled transforms" [89]. Therefore, it is not a direct replacement for schedules in other systems, but additional layers on top of transform dialect, such as in PEAK [134], could potentially improve this. Layered schedules would be needed for transformations that significantly change the IR. Such an approach is still limited in that it relies on a specific compiler infrastructure that cannot be reused outside that environment.

Alternatively, we can design high-level languages aimed at specifying *a source and target languages together with the code transformations operating on them.* For example, in the functional context, the essence of a code transformation can often be concisely and elegantly expressed in the form of inference rules, declared for each syntactic category of the source language. We hypothesize that the verbose, complex and error-prone code that implements the code transformation—i.e., that traverses the abstract-syntax tree, matches the pre-conditions, and applies the inference rules—can be automatically generated. Rather than requiring all compiler experts to work on the same code base, such an approach would democratize the implementation of a new language and its optimizing compiler by allowing to reuse, adjust, and compose available components from the specification of the code transformations belonging to other languages. Such solutions can build on the findings of strategy languages, such as ELEVATE [62] and MDH [112], which are aimed to improve the language aspects of schedules: clearly-defined semantics, composability, and opportunity for (automatic) verification. For example, it is arguably easier to reason about and verify a high-level specification than its low-level implementation.

For any of these approaches to be impactful, adoption by the community is a critical step.

4.2 Data Layout and Data Movement in Schedules

Since data movement is the key cost in terms of time and energy, incorporating specifications of data layout and data movement into scheduling languages should be a universal part of future exploratory compilers. Organizing data according to its access pattern then requires modifying the computation accordingly. We discussed the role of data layout in terms of sparse tensor computations, but other computations also benefit from data layout and data movement specifications.

Historically, data copy was applied to reorganize submatrices, especially to avoid conflict misses in cache or stage data in explicitly managed storage [15, 136]. CHiLL incorporated datacopy into its scheduling language [31], which was later adapted in CUDA-CHiLL to copy data to/from global memory, shared memory, and texture memory in GPUs [76]. More recently, Fireiron and MDH enrich these data movement specifications for GPUs [60, 112]. Follow-on work, for example in Graphene, optimizes data layout and thread mapping, particularly to prepare data and computation for tensor cores [61].

While not supported by scheduling languages, other layouts beyond strided rectangular regions and sparse tensor representations, have been shown to reduce data movement. For example, fine-grain data blocking, where logically adjacent three-dimensional subdomains are stored in contiguous memory, has been shown to significantly reduce data movement in structured grid computations [159]. A two-dimensional *tile*, also a fine-grain data block, has been used in Triton to accelerate deep learning computations [138].

Looking forward data layout specifications in scheduling languages should support these and other future layouts. To generalize this approach, the scheduling language should be integrated into a compiler supporting high-level abstraction such as MLIR. To generate code, the compiler must map between *logical* and *physical* data layouts, so that address calculations can map logical indices to their physical locations. In sparse tensor computations, where logical indices may not have corresponding physical entries, the inverse mapping from physical to logical indices can be used to find corresponding elements in other tensors during co-iteration, as is done in dlcomp [160]. The mapping from a variety of layouts to hierarchical sublayouts could be supported with additional logical dimensions, generalizing the approach in Graphene [61].

4.3 Architecture-Specific Scheduling Primitives

Related to data layout is the need to support architecture-specific hardware features, which was discussed above for Graphene [61] and Fireiron [60] in the context of matrix processing units for NVIDIA GPUs. How might architecture-specific constructs be integrated into a scheduling language?

One mechanism is to layer scheduling languages, providing an extensible programming language approach to build new scheduling constructs either from existing ones, or by describing how to rewrite code based on scheduling constructs. More recently, Exocompilation [70] addresses this challenge with a user-level scheduling language, named Exo, that permits a performance expert to describe architecture details that impact performance. In Exo, users may define specialized memories including lower precision data types; semantics of hardware instructions, e.g., fused multiply-add; and hardware state so as to avoid redundant computation or data movement. Such frameworks can enable *multi-objective* optimization of energy and performance, which requires the ability to utilize varying degrees of GPU's fast memories and SIMT length, e.g., by predictive modeling that finds effective tile sizes that improve locality and benefit from dynamic and frequency scaling [72].

The above approaches address the extensibility of scheduling languages, but not the widespread adoption of these extensions. As new classes of architectures arise, extensions permit exploration of how to derive performant schedules. Once the community settles on a solution, these architecture-specific layers can be integrated into the language and compiler mechanisms.

4.4 Integration with Runtime

Many optimizations require a combination of static and runtime information to make decisions. For example, inspector-executor strategies, as previously discussed in the context of sparse tensors, may wish to choose data layouts based on the nonzero structure of the sparse tensors – information only available at runtime. Similarly, a runtime data layout transformation, such as ZMorton order, would need to be part of the schedule, especially since it may require costly layout conversion that needs to happen early in the computation.

Furthermore, in computations such as preconditioners for sparse solvers, the resulting loop parallelism is data-dependent [145], hence the computation needs to be reorganized in a sequence of parallel wavefronts, whose structure is determined at runtime. In molecular-dynamics (astrophysics) simulations, the distribution of molecules (particles) is unknown at least until runtime, and it might also dynamically change. Dynamic analyses [42, 131] proposed in these contexts use a statically-generated inspector to perform data- and (loop-) iteration reordering in order to optimize spatial and temporal locality, respectively. Such runtime-reordering transformations have been later integrated in sparse-polyhedral frameworks [132] and have further inspired the notion of locality hypergraphs [131], for example, as a way to model the optimization of communication in distributed computations as graph-partitioning problems [114, 117].

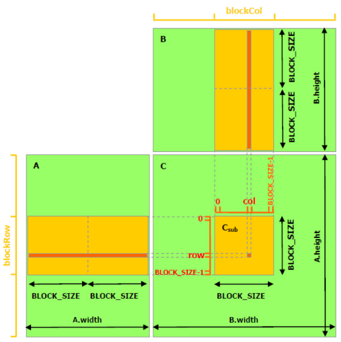
Finally, another facet of runtime integration is the decision process for selecting the appropriate optimized code variant if that choice is data dependent. The selection might involve a lookup into pre-autotuned variants based on data ranges, or use of inference based on problem size or other input data features.

Current scheduling languages are typically weak at supporting integration with the runtime system, of the kind described above. We hypothesize that future research directions will be aimed at incorporating such dynamic analyses into the repertoire supported by scheduling languages.

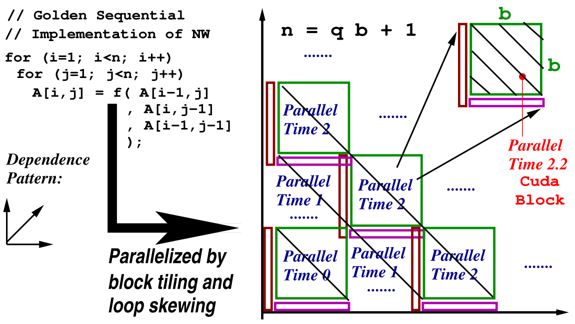
4.5 Practical Predictive Modeling

Like with conventional autotuning, the challenge of using machine learning to develop predictive models for choosing among potential schedules revolves around navigating a prohibitively large search space during the training phase. For instance, Merouani et al. extend Tiramisu's autoscheduler to explore a large space of optimizations on 29 million randomly-generated data points, taking eight months on a 15-node cluster [91]. Nonetheless, once trained, ML-based optimization techniques continue to exhibit state-of-the-art performance and some prominent work in this domain are [3, 33, 39, 64, 83, 103, 162].

Patabandi et al. identify a key challenge in the cost of training to be a ***Multiplicative Domain Formulation (MDF)*** [105], referring to the combinatorial exploration of multiple transformations' schedule domains. For convolutions, they demonstrate an example of ***Additive Domain Formulation (ADF)***, querying an existing model for one transformation (loop permutation) [104] when building the model for another transformation (tiling), demonstrating high accuracy and a 100× reduction in inference time.



(a) Picture schedule for Matrix Multiplication (MM)



```
// Golden Sequential
// Implementation of NW
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    A[i,j] = f( A[i-1,j]
              , A[i,j-1]
              , A[i-1,j-1]
              );
```

(b) Picture Schedule for Needleman-Wunsch (NW).

Fig. 10. Picture-like Schedules for MM [77] and NW [96]: they seem isomorphic with a language representing array slices, i.e., describing the result (and input) slices produce (read) at each level of the parallel-hierarchy of the hardware.
The NW figure (right) hints that the computation is organized as a wavefront, where the ith anti-diagonal of $b \times b$ blocks is processed at parallel time i. Each $b \times b$ logical block is processed within a Cuda block of size b: the green box denotes the write set of a Cuda block and the red/purple vertical/horizontal lines denote the read set of a Cuda block; all are being mapped to shared memory. The computation inside a Cuda block is also organized in a wavefront structure that computes at parallel time i the ith anti-diagonal (see the zoomed block).

4.6 Raising the Level of Abstraction

Most of the work reviewed so far expresses a schedule as a (precise and detailed) sequence of compiler transformations. Arguably, such a representation is unfriendly to domain experts, who might be well versed in (or at least willing to learn) “parallel thinking”, but may find that the gap to “thinking like a compiler” is too wide to bridge. Ideally, a scheduling language would specify in an ergonomic way *“do what I mean” concepts that are accessible to experienced programmers*, rather than only compiler experts, *from which a matching sequence of code transformations is automatically inferred*. Moreover, if the schedule introduces redundant computation (as in Halide), the intuitive concepts should make this cost visible. The automatic inference should verify not only the correctness of the transformation, but also that the resulting code conforms with the specified cost.

A possible approach in this sense is to allow the user to sketch code transformations in a manner that promotes visualization, e.g., by describing the array slices produced/used at each level of the hardware, respectively. Figure 10 demonstrates the pedagogical use of such visualizations in two scientific publications to explain locality optimizations for matrix multiplication [77] and for the Needleman–Wunsch problem [96]. The former requires tiling at each hardware level, and the latter requires tiling and two loop-skewing transformations.

While the domain experts might find it difficult to “think like a compiler"—e.g., loop skewing has seldom been called intuitive—they can arguably reason in terms of wavefront parallelism, both across the blocks and within a $b \times b$ block depicted in Figure 10(b), where the latter is intended to be mapped at CUDA-block level in shared memory. Such a visual representation of schedules is essentially isomorphic with a language of layouts that decomposes arrays in the manner that matches the desired iteration space of the transformed code. Allowing overlapping dimensions in the layout can even model (and specify the cost of) transformations introducing redundant computation.

Finally, a more holistic approach would be to have the domain expert sketch (de)composition opportunities, which are then hierarchically applied to each level of the hardware automatically, by means of compiler exploration and compositional tuning engines.

5 Summary

This article has provided a chronology of scheduling languages that illustrates how the past approaches aimed at optimizing performance-critical applications have motivated the emergence of scheduling languages, and how future improvements may cycle back to the past vision of more automation.

Past work (1997 – 2012) has targeted scientific applications expressed in mainstream languages, e.g., in terms of loop nests, that could not be adequately optimized by the heuristic pipelines of general-purpose compilers. This has led to self-tuning libraries and more advanced compilation techniques that better explore the optimization space. Examples include iterative compilation that, at each step, generates multiple candidates and selects the one that performs best on the target hardware; or rewrite-rule systems that encode algorithmic properties of high-level language constructs.

Present work (2013 – 2023) is driven by the observation that specialization gives rise to performance, in that restricting (and purifying) the specification language (DSLs) allows it to be effectively optimized by means of a relatively small number of code transformations, which are sequenced either explicitly by the compiler expert, or automatically by tuning strategies.

We envision that future work will broaden the specification language to cover more general computations and the repertoire of code transformations available to scheduling, e.g., with various dynamic analyses and with support for specifying data layout and data movement. On one hand, this will require improvements to the search strategies to cover this expanded space in practical time. On the other hand, this will require productivity-oriented improvements that minimize the user interaction with the scheduling language, while still allowing specification of key optimization insights. Ideally, the domain expert can directly interact with scheduling languages, by lifting their level of abstraction. Achieving this requires a level of integration and automation reminiscent of earlier exploratory compilers, thus circling back to the past.

Acknowledgments

We thank the Lorentz Center and its 2022 Workshop on Generic Auto-tuning for GPU Applications, which brought the authors together. We also thank Tor A. Haugdahl for his contributions. A. C. Elster would like to thank K. Pingali for hosting her at The Oden Institute.

References

[1] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, et al. 1997. OCEANS: Optimizing compilers for embedded applications. In *Euro-Par'97 Parallel Processing*. Springer-Verlag, 1351–1356. https://dl.acm.org/doi/10.5555/646662.699219

[2] K. Abdelaal and M. Kong. 2021. Tile size selection of affine programs for GPGPUs using polyhedral cross-compilation. In *Procs. Int. Conf. on Supercomputing (ICS)*.

[3] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4, Article 121 (Jul 2019), 12. https://dl.acm.org/doi/10.1145/3306346.3322967

[4] P. Ahrens, F. Kjolstad, and S. Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*.

[5] W. Ahrens, D. Donenfeld, F. Kjolstad, and S. Amarasinghe. 2023. Looplets: A language for structured coiteration. In *ACM/IEEE Int. Symp. Code Gen. Optim. (CGO)*.

[6] L. Anderson, A. Adams, K. Ma, T.-M. Li, T. Jin, and J. Ragan-Kelley. 2021. Efficient automatic scheduling of imaging and vision pipelines for the GPU. *Proc. ACM Program. Lang.* 5, Article 109 (Oct 2021). https://dl.acm.org/doi/10.1145/3485486

[7] J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Int. Conf. on Prog. Lang. Design and Impl. (PLDI)*.

[8] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Conf. Par. Arch. and Comp. (PACT)*.

[9] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sep 2018), 42. https://dl.acm.org/doi/10.1145/3197978

[10] I. Bachiri, H. Benmeziane, S. Niar, R. Baghdadi, H. Ouarnoughi, and A. Aries. 2024. Combining neural architecture search and automatic code optimization: A survey.

[11] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba, and S. Amarasinghe. 2021. A deep learning based cost model for automatic code optimization. In *Procs. of Machine Learning and Systems*. 181–193.

[12] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Int. Symp. on Code Gen. and Optim. (CGO)*. IEEE, 193–205.

[13] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. Hollingsworth, B. Norris, and R. Vuduc. 2018. Autotuning in high-performance computing applications. *Proc. IEEE* 106, 11 (2018), 2068–2083.

[14] M. Bansal, O. Hsu, K. Olukotun, and F. Kjolstad. 2023. Mosaic: An interoperable compiler for tensor algebra. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl. (PLDI)*.

[15] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Princ. Pract. of Par. Prog.* 1–10.

[16] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, Xiaoyang Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, et al. 2005. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proc. IEEE* 93, 2 (2005), 276–292.

[17] T. Ben-Nun, J. de Fine Licht, A. Ziogas, T. Schneider, and T. Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Int. Conf. on High Perf. Comp., Networking, Storage and Analysis (SC)*.

[18] T. Ben-Nun, L. Groner, F. Deconinck, T. Wicky, E. Davis, J. Dahm, O. D. Elbert, R. George, J. McGibbon, L. Trümper, et al. 2022. Productive performance engineering for weather and climate modeling with python. In *Procs. Int. Conf. on High Perf. Comp., Netw. Storage Anal.*

[19] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad. 2022. Compiler support for sparse tensor computations in MLIR. *ACM Trans. Archit. Code Optim. (TACO)* 19, 4, Article 50 (Sep 2022), 25. https://dl.acm.org/doi/10.1145/3544559

[20] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. 1997. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Int. Conf. on Supercomputing (ICS'97)*.

[21] G. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.* 38, 11 (1989), 1526–1538.

[22] G. Blelloch. 1990. *Vector Models for Data-parallel Computing*. Vol. 75. MIT Press, Cambridge.

[23] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Work. on Prof. Feedb.-dir. Compil.*

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Procs. of Conf. on Prog. Lang. Design and Imple. (PLDI'08)*.

[25] J. M. Boyle, T. J. Harmer, and V. L. Winter. 1997. The TAMPR program transformation system: Simplifying the development of numerical software. In *Modern Software Tools for Scientific Computing*, Birkhauser Boston Inc., 353–372.

[26] L. M. Bruun, U. S. Larsen, N. H. Hinnerskov, and C. E. Oancea. 2024. Reverse-mode AD of multi-reduce and scan in Futhark. In *Procs. Symp. on Implem. and Applic. of Funct. Lang. (IFL)*. ACM.

[27] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *International Conference on Principles and Practice of Programming in Java (PPPJ '11)*.

[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* 40, 10 (2005), 519–538.

[29] L. Chelini, T. Gysi, T. Grosser, M. Kong, and H. Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Procs. of Int. Conf. on Par. Arch. and Comp. Tech. (PACT)*.

[30] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal. 2019. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (2019), 25. https://dl.acm.org/doi/10.1145/3372266

[31] C. Chen, J. Chame, and M. W. Hall. 2008. CHiLL: A framework for composing high-level loop transformations. University of Southern California.

[32] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Conf. on Op. Sys. Design and Implem. (OSDI)*. 579–594.

[33] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, Curran Associates Inc., Montréal, Canada, 3393–3404.

[34] S. Chou, F. Kjolstad, and S. Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 1–30.

[35] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula. 2016. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Procs. of Int. Conf. on Par. Arch. and Comp. (PACT '16).* ACM.

[36] D. Clark. 1998. OpenMP: A parallel standard for the masses. *IEEE Concurrency* 6, 1 (1998), 10–12.

[37] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. New Trends in High Performance Computing.

[38] D. Cociorva, G. Baumgartner, C.-C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. E. Bernholdt, and R. Harrison. 2002. Space-time trade-off optimization for a class of electronic structure calculations. In *Conf. on Prog. Lang. Design and Impl. (PLDI).* ACM, 177–186.

[39] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-end deep learning of optimization heuristics. In *26th Int. Conf. on Par. Arch. and Comp. Tech. (PACT).* IEEE, 219–232.

[40] O. de Moor and G. Sittampalam. 1999. Generic program transformation. In *Advanced Functional Programming.* S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques (Eds.), Springer Berlin , 116–149.

[41] K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. 1994. OPAL: Design and implementation of an algebraic programming language. In *Prog. Languages and System Architectures.* Springer Berlin, 228–244. https://dl.acm.org/doi/10.5555/184716.184730

[42] C. Ding and K. Kennedy. 1999. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Procs. of Conf. on Prog. Language Design and Implem. (PLDI '99).* ACM, 229–241.

[43] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. 2006. A language for the compact representation of multiple program versions. In *Langs. and Compilers for Par. Comp. (LCPC).* Springer Berlin H., 136–151. https://dl.acm.org/doi/abs/10.1007/978-3-540-69330-7_10

[44] A. C. Elster. 1989. Fast bit-reversal algorithms. In *IEEE Int. Conf. on Accoust. Speech Signal Process. (ICASSP).*

[45] A. C. Elster and J. C. Meyer. 2009. A super-efficient adaptable bit-reversal algorithm for multithreaded architectures. In *2009 IEEE Int. Parallel and Distrib. Process.* 1–8.

[46] J. A. Fagervik. 2024. *Efficient Processing of Distributed Acoustic Sensing Data: Anomaly Detection Using Autoencoders.* Master's thesis. Norwegian University of Science and Technology.

[47] T. L. Falch and A. C. Elster. 2015. Machine learning based auto-tuning for enhanced OpenCL performance portability. In *2015 IEEE Int. Parallel and Distrib. Process. Symp Work.* 1231–1240.

[48] T. L. Falch and A. C. Elster. 2017. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurr. and Comput.: Pract. and Exp.* 29, 8 (2017), e4029.

[49] T. L. Falch and A. C. Elster. 2018. ImageCL: Language and source-to-source compiler for performance portability, load balancing, and scalability prediction on heterogeneous systems. *Concurr. and Comput.: Pract. and Exp.* 30, 9 (2018), 22. https://onlinelibrary.wiley.com/doi/10.1002/cpe.4384

[50] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. 2006. Sequoia: Programming the memory hierarchy. In *Proc. of the 2006 ACM/IEEE Conf. on Supercomput. (SC).*

[51] P. Fegade, T. Chen, P. Gibbons, and T. Mowry. 2021. Cortex: A compiler for recursive deep learning models. In *Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.). Vol. 3. 38–54. https://proceedings.mlsys.org/paper_files/paper/2021/hash/eca986d585a03890a412587a2f5ccb43-Abstract.html

[52] P. Fegade, T. Chen, P. Gibbons, and T. Mowry. 2022. The CoRa tensor compiler: Compilation for ragged tensors with minimal padding. In *Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.). Vol. 4. 721–747.

[53] Matteo Frigo. 1999. A fast fourier transform compiler. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI).*

[54] S. Ghosh, M. Martonosi, and S. Malik. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 21, 4 (Jul 1999), 703–746.

[55] F. Gieseke, S. Rosca, T. Henriksen, J. Verbesselt, and C. E. Oancea. 2020. Massively-parallel change detection for satellite time series data with missing values. In *Int. Conf. on Data Eng. (ICDE).* 385–396.

[56] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. of Parallel Program.* 34, 3 (2006), 35–44.

[57] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX USENIX Symp. on Operating Sys. Design and Impl. (OSDI 12).*

[58] T. Gysi, T. Grosser, and T. Hoefler. 2015. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Int. Conf. on Supercomputing (ICS).*

[59] T. Gysi, T. Grosser, and T. Hoefler. 2019. Absinthe: Learning an analytical performance model to fuse and tile stencil codes in one shot. In *Procs. Int. Conf. on Par. Arch. and Compilation Tech. (PACT)*.

[60] B. Hagedorn, A. Elliott, H. Barthels, R. Bodik, and V. Grover. 2020. Fireiron: A data-movement-aware scheduling language for GPUs. In *Procs. of Int. Conf. on Par. Arch. and Compilation Tech. (PACT)*.

[61] B. Hagedorn, B. Fan, H. Chen, C. Cecka, M. Garland, and V. Grover. 2023. Graphene: An IR for optimized tensor computations on GPUs. In *Proc. Arch. Support for Prog. Lang. and Sys. (ASPLOS)*.

[62] B. Hagedorn, J. Lenfers, T. Kundefinedhler, X. Qin, S. Gorlatch, and M. 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 (2020), 29. https://dl.acm.org/doi/10.1145/3408974

[63] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. 2018. High performance stencil code generation with lift. In *Int. Symp. on Code Gen. and Optim. (CGO)*.

[64] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Int. Symp. on Code Gen. and Optim. (CGO)*.

[65] A. Haj-Ali, H. Genc, Q. Huang, W. Moses, J. Wawrzynek, K. Asanovi?, and I. Stoica. 2020. ProTuner: Tuning programs with monte carlo tree search.

[66] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. 2010. Loop transformation recipes for code generation and auto-tuning. In *Lang. and Compilers for Par. Computing (PCPC)*, Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li (Eds.). Springer Berlin H., 50–64.

[67] A. Hartono, B. Norris, and P. Sadayappan. 2009. Annotation-based empirical performance tuning using orio. In *IEEE Int. Symp. on Par. & Distrib. Processing (IPDPS)*. 1–11.

[68] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. 2019. Incremental flattening for nested data parallelism. In *Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*. ACM, 53–67.

[69] D. Huff, S. Dai, and P. Hanrahan. 2021. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In *IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 186–194.

[70] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *43rd ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*. 703–718.

[71] E.J. Im, K. Yelick, and R. Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. of High Perform. Comput. Appl.* 18, 1 (2004), 135–158.

[72] M. Jayaweera, M. Kong, Y. Wang, and D. Kaeli. 2024. Energy-aware tile size selection for affine programs on GPUs. In *IEEE/ACM Int. Symp. on Code Gen. and Optimization (CGO)*. 13–27.

[73] R. Jensen, I. Karlin, and A. C. Elster. 2011. Autotuning a matrix routine for high performance. In *NIK 2011*.

[74] S. P. Jones, A. Tolmach, and T. Hoare. 2001. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, Vol. 1. 203–233.

[75] R. Kaleem, A. Venkat, S. Pai, M. Hall, and K. Pingali. 2016. Synchronization trade-offs in GPU implementations of graph algorithms. In *IEEE Int. Par. and Distrib. Processing Symp. (IPDPS)*.

[76] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. 2013. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim. (TACO)* 9, 4, Article 31 (2013), 25 pages.

[77] D. B. Kirk and W. W. Hwu. 2016. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[78] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Int. Conf. on Parallel Arch. and Comp. Tech. (PACT)*. 237–246.

[79] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. 2017. The tensor algebra compiler. *Proc. of the ACM on Prog. Lang.* 1, OOPSLA (2017), 1–29.

[80] T. Koehler, A. Goens, S. Bhat, T. Grosser, P. Trinder, and M. Steuwer. 2024. Guided equality saturation. *Proc. of the ACM on Program. Lang.* 8, POPL (2024), 1727–1758.

[81] M. Kong and L.-N. Pouchet. 2019. Model-driven transformations for multi- and many-core CPUs. In *Procs. Conf. on Prog. Lang. Design and Implem. (PLDI)*. ACM, 469–484.

[82] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. 2007. Optimistic parallelism requires abstractions. In *Int. Conf. on Prog. Lang. Design and Implem. (PLDI)*. 211–222.

[83] S. Kulkarni and J. Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Int. Conf. Obj. Oriented Prog., Systems, Langs. (OOPSLA)*. 147–162.

[84] A. Kyrola, G. Blelloch, and C. Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symp. on Operating Sys. Design and Impl. (OSDI 12)*.

[85] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Int. Symp. on Code Gen. and Opt. (CGO)*. 2–14.

[86] A. Lenharth and K. Pingali. 2015. Scaling runtimes for irregular algorithms to large-scale NUMA systems. *Computer* 48, 8 (2015), 35–44.
[87] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.* 37, 4, Article 139 (2018), 13. https://dl.acm.org/doi/10.1145/3197517.3201383
[88] P. Lipps, U. Möncke, and R. Wilhelm. 1989. OPTRAN - A language/system for the specification of program transformations: System overview and experiences. In *Compiler Compilers and High Speed Compilation*, Dieter Hammer (Ed.). Springer Berlin H.
[89] M. P. Lücke, O. Zinenko, W. S. Moses, M. Steuwer, and A. Cohen. 2025. The MLIR transform dialect: Your compiler is more powerful than you think. In *Procs. of Int. Symp. on Code Generation and Optimization (CGO'25).* ACM, 241–254.
[90] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar. 2016. Automatic data layout generation and kernel mapping for CPU+GPU architectures. In *Procs. Int. Conf. on Compiler Constr. (CC).* ACM, 240–250.
[91] M. Merouani, K. A. Boudaoud, I. N. Aouadj, N. Tchoulak, I. K. Bernou, H. Benyamina, F. Benbouzid-Si Tayeb, K. Benatchba, H. Leather, and R. Baghdadi. 2024. LOOPer: A learned automatic code optimizer for polyhedral compilers.
[92] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. K. Prasanna, M. Püschel, B. Singer, M. Veloso, and J. Xiong. 2001. Generating platform-adapted DSP libraries using SPIRAL. In *High Perf. Extreme Comp. (HPEC).*
[93] R. Mullapudi, V. Vasista, and U. Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Procs. Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS).* ACM, 429–443.
[94] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (Jul 2016), 11 pages.
[95] P. Munksgaard, S. L. Breddam, T. Henriksen, F. C. Gieseke, and C. Oancea. 2021. Dataset sensitive autotuning of multi-versioned code based on monotonic properties. In *Trends in Functional Programming*, Viktória Zsók and John Hughes (Eds.). Springer, 3–23.
[96] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. 2022. Memory optimizations in an array language. In *Procs. Int. Conf. on High Perf. Comp., Networking, Storage and Analysis (SC).* IEEE, Article 31.
[97] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 35, 1, Article 3 (2013), 48. https://dl.acm.org/doi/10.1145/2450136.2450138
[98] K. Narasimhan, A. Acharya, A. Baid, and U. Bondhugula. 2021. A practical tile size selection model for affine loop nests. In *Procs. Int. Conf. on Supercomputing (ICS).* ACM, 27–39.
[99] N. Nayak, T. O. Odemuyiwa, S. Ugare, C. Fletcher, M. Pellauer, and J. Emer. 2023. TeAAL: A declarative framework for modeling sparse tensor accelerators. In *Procs. of IEEE/ACM Int. Symposium on Microarchitecture (MICRO'23).* 1255–1270.
[100] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris. 2015. Generating efficient tensor contractions for GPUs. In *44th Int. Conf. on Par. Proc. (ICPP).* 969–978.
[101] D. Nguyen, A. Lenharth, and K. Pingali. 2013. A lightweight infrastructure for graph analytics. In *Symp. on Operating Systems Principles (SOSP '13).* ACM, 456–471.
[102] Cedric Nugteren. 2018. CLBlast: A tuned OpenCL BLAS library. In *Workshop on OpenCL (IWOCL '18).* ACM, Article 5.
[103] E. Park, S. Kulkarni, and J. Cavazos. 2011. An evaluation of different modeling techniques for iterative compilation. In *14th Int. Conf. on Comp., Arch. and Synth. for Embedded Sys. (CASES '11).* ACM, 65–74.
[104] T. Patabandi, A. Venkat, A. Kulkarni, P. Ratnalikar, M. Hall, and J. Gottschlich. 2021. Predictive data locality optimization for higher-order tensor computations. In *Int. Symp. on Machine Prog. (MAPS 2021).*
[105] T. R. Patabandi and M. Hall. 2023. Efficiently learning locality optimizations by decomposing transformation domains. In *32nd ACM SIGPLAN International Conf. on Compiler Constr. (CC 2023).* 37–49.
[106] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *Procs. of Prog. Lang. Design and Implem. (PLDI).* ACM, 12–25.
[107] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *29th ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI).* 90–100.
[108] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232– 275.
[109] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM Sigplan Notices* 48, 6 (2013), 519–530.
[110] A. Rasch. 2024. (De/Re)-composition of data-parallel computations via multi-dimensional homomorphisms. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 46, 3 (May 2024). 73 pages.

[111] A. Rasch, R. Schulze, W. Gorus, J. Hiller, S. Bartholomäus, and S. Gorlatch. 2019. High-performance probabilistic record linkage via multi-dimensional homomorphisms. In *Symp. on Applied Computing (SAC '19)*. ACM, 526–533.

[112] A. Rasch, R. Schulze, D. Shabalin, A. C. Elster, S. Gorlatch, and M. Hall. 2023. (De/Re)-compositions expressed systematically via MDH-based schedules. In *Int. Conf. on Comp. Constr. (CC)*. ACM, 61–72.

[113] A. Rasch, R. Schulze, M. Steuwer, and S. 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 (2021), 26. https://dl.acm.org/doi/10.1145/3427093

[114] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. 2012. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Proc. of SC'12 (SC '12)*. IEEE, Article 72.

[115] P. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan. 2018. Domain-specific optimization and generation of high-performance GPU code for stencil computations. *Proc. of the IEEE* 106, 11 (2018), 1902–1920.

[116] P. Rawat, M. Vaidya, A. Sukumaran-Rajam, A. Rountev, L. Pouchet, and P. Sadayappan. 2019. On optimizing complex stencils on GPUs. In *IEEE Int. Parallel and Distrib. Processing Symp. (IPDPS)*.

[117] C. Reddy and U. Bondhugula. 2014. Effective automatic computation placement and data allocation for parallelization of regular programs. In *Int. Conf. on Supercomputing (ICS '14)*.

[118] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. 2018. Loop tiling in large-scale stencil codes at run-time with OPS. *IEEE Trans. on Par. and Distr. Sys.* 29, 4 (2018), 873–886.

[119] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. 2016. Performance portable GPU code generation for matrix multiplication. In *W. Gen. Purpose Proc. Using Graphics Processing Unit (GPGPU '16)*. ACM, 22–31.

[120] Even Olsson Rogstadkjærnet. 2018. *ImageCL 3D Extensions Targeting Adative Mesh Refinement Proxy Applications on GPUs*. Master's thesis. Norwegian University of Science and Technology.

[121] R. Schenck, O. Rønning, T. Henriksen, and C. E. Oancea. 2022. AD for an array language with nested parallelism. In *Procs. Int. Conf. on High Perf. Comp., Networking, Storage and Analysis (SC '22)*. IEEE, Article 58.

[122] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proc. of the ACM on Prog. Lang.* 4, OOPSLA (2020), 1–30.

[123] J. Shirako and V. Sarkar. 2020. An affine scheduling framework for integrating data layout and loop transformations. In *Lang. and Comp. for Par. Comp. (LCPC)*. 3–19.

[124] J. Shun and G. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symp. on Princ. and Pract. of Par. Prog. (PPoPP '13)*.

[125] S. Sioutas, S. Stuijk, T. Basten, H. Corporaal, and L. Somers. 2020. Schedule synthesis for halide pipelines on GPUs. *ACM Trans. Archit. Code Optim. (TACO)* 17, 3, Article 23 (Aug 2020), 25 pages.

[126] S. Sioutas, S. Stuijk, L. Waeijen, T. Basten, H. Corporaal, and L. Somers. 2019. Schedule synthesis for halide pipelines through reuse analysis. *ACM Trans. Archit. Code Optim. (TACO)* 16, 2, Article 10 (Apr 2019), 22 pages.

[127] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. 2015. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Procs. ICFP*. ACM, 205–217.

[128] M. Steuwer, T. Koehler, B. Köpcke, and F. Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design.

[129] R. Strandh and A. C. Elster. 1998. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report CNA-288.

[130] M. Strout, M. Hall, and C. Olschanowsky. 2018. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[131] M. Strout and P. Hovland. 2004. Metrics and models for reordering transformations. In *Workshop on Mem. Sys. Perf. (MSP'04)*.

[132] M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky. 2016. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.* 53, C (2016), 32–57.

[133] Q. Sun, Y. Liu, H. Yang, Z. Jiang, Z. Luan, and D. Qian. 2024. Adaptive auto-tuning framework for global exploration of stencil optimization on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 35, 1 (2024), 20–33.

[134] A. M. Tavakkoli, S. Joshi, S. Singh, Y Xu, P. Sadayappan, and M. Hall. 2023. PEAK: Generating high-performance schedules in MLIR. In *Int. Workshop on Lang. and Comp. for Par. Comp. (LCPC)*. Springer Verlag.

[135] O. Temam, C. Fricker, and W. Jalby. 1994. Cache interference phenomena. In *SIGMETRICS*. ACM, 261–271.

[136] O. Temam, E. D. Granston, and W. Jalby. 1993. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Procs. of ACM/IEEE Conf. on Supercomputing (Supercomputing'93)*. ACM, 410–419.

[137] S. F. X. Thiago Teixeira, Corinne Ancourt, David Padua, and William Gropp. 2019. Locus: A system and a language for program optimization. In *IEEE/ACM Int. Symp. on Code Gen. and Optimization (CGO)*. 217–228.

[138] P. Tillet, H. T. Kung, and D. Cox. 2019. Triton: An intermediate language and compiler for tiled neural network computations. In *Workshop on Machine Learning and Prog. Lang. (MAPL 2019).*

[139] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *IEEE Int. Symp. on Parallel & Distrib. Proc. (IPDPS).* 1–12.

[140] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and David I. August. 2003. Compiler optimization-space exploration. In *Int. Symp. on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 204–215.

[141] L. Trümper, T. Ben-Nun, P. Schaad, A. Calotoiu, and T. Hoefler. 2023. Performance embeddings: A similarity-based transfer tuning approach to performance optimization. In *Int. Conf. on Supercomputing (ICS '23).* ACM, 50–62.

[142] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2018. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions.

[143] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2019. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. *ACM Trans. Archit. Code Optim. (TACO)* 16, 4, Article 38 (Oct 2019), 26. https://dl.acm.org/doi/10.1145/3355606

[144] A. Venkat, M. Hall, and M. Strout. 2015. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI).*

[145] A. Venkat, M. Mohammadi, J. Park, H. Rong, R. Barik, M. Strout, and M. Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Procs. Int. Conf. on High Perf. Comp., Networking, Storage and Analysis (SC).*

[146] A. Venkat, T. Rusira, R. Barik, M. Hall, and L. Truong. 2019. SWIRL: High-performance many-core CPU code generation for deep neural networks. *Int. Journal of High Perf. Comp. Appls.* 33, 6 (2019), 1275–1289.

[147] A. Venkat, M. Shantharam, M. Hall, and M. Strout. 2014. Non-affine extensions to polyhedral code generation. In *IEEE/ACM Int. Symp. on Code Gen. and Opt. (CGO).*

[148] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim. (TACO)* 9, 4 (2013), 54:1–54:23.

[149] S. Verdoolaege and G. Janssens. 2017. Scheduling for PPCG. *Report CW* 706 (2017).

[150] E. Visser, Z. Benaissa, and A. Tolmach. 1998. Building program optimizers with rewriting strategies. In *ACM SIGPLAN International Conf. on Func. Prog. (ICFP '98).*

[151] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas. 2017. Extending halide to improve software development for imaging DSPs. *ACM Trans. Archit. Code Optim. (TACO)* 14, 3, Article 21 (Aug 2017), 25. https://dl.acm.org/doi/10.1145/3106343

[152] R. Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels.* Ph.D. Dissertation. University of California at Berkeley.

[153] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically tuned linear algebra software. In *Conf. on Supercomputing (SC).* 1–27.

[154] M. E. Wolf and M. S. Lam. 1991. A data locality optimizing algorithm. In *Prog. Lang. Design and Implem. (PLDI).* 30–44.

[155] J. Won, C. Mendis, J. S. Emer, and S. Amarasinghe. 2023. WACO: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *28th Int. Conf. on Arch. Support for Prog. Lang. and Operating Sys. (ASPLOS).*

[156] R. Yadav, A. Aiken, and F. Kjolstad. 2022. DISTAL: The distributed tensor algebra compiler. In *ACM SIGPLAN Int. Conf. on Prog. Lang. Design and Impl. (PLDI).* 286–300.

[157] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. 2007. POET: Parameterized optimizations for empirical tuning. In *IEEE International Par. and Distr. Proc. Symp. (IPDPS).* 1–8.

[158] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. 2018. Graphit: A high-performance graph DSL. *Proc. of the ACM on Prog. Lang.* 2, OOPSLA (2018), 1–30.

[159] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen. 2019. Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs. In *Procs. Int. Conf. on High Perf. Comp., Netw., Storage and Anal. (SC).*

[160] T. Zhao, T. Popoola, M. Hall, C. Olschanowsky, and M. Strout. 2022. Polyhedral specification and code generation of sparse tensor contraction with co-iteration. *ACM Trans. Archit. Code Optim. (TACO)* 20, 1, Article 16 (Dec 2022), 26. https://dl.acm.org/doi/10.1145/3566054

[161] L. Zheng, C. Jia, M. Sun, Z. Wu, C. Hao Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *USENIX Symp. on Op. Sys. Design and Impl. (OSDI).* 863–879.

[162] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng. 2020. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*, 26.

Received 13 November 2024; revised 19 March 2025; accepted 12 May 2025