# Interactive Compiler Scheduling with Strong Guarantees

Evan Williams
Unaffiliated
USA

Justin Lubin
University of California, Berkeley
USA

Rubens Lacouture
Stanford University
USA

Olivia Hsu
Stanford University/Carnegie Mellon University
USA

## ABSTRACT

Scheduling languages have become increasingly common in optimizing compilers for hardware accelerators. However, achieving high performance often requires significant manual scheduling effort from users. On the other hand, auto-schedulers reduce users' ability to tune compiler schedules for desired properties and can lead to suboptimal results. Instead, what if we could retain the ability for users to write any valid schedule while also providing them a scaffolded program authoring experience?

To explore this vision, we propose using interactive program synthesis with strong guarantees on the interaction and resulting schedule. As a case study, we use Programming by Navigation (an interactive synthesis approach that ensures users always remain en route to a valid schedule while retaining the ability to reach any valid schedule) to generate schedules for FuseFlow (a sparse machine learning compiler that targets multiple hardware backends). Through this work, we argue for scheduling systems that leverage interactive program synthesis with strong guarantees to scaffold the program authoring process, allowing users to focus on the parts of the schedule they deem most important.

## 1 INTRODUCTION

Scheduling languages are a way to design and program domain-specific accelerators and their compilers [7, 14, 24, 25] by letting users explicitly optimize performance around key hardware constructs through, *e.g.*, tiling, memory hierarchy management, and data pinning [5, 10, 12, 16, 20]. Commonly, users can either create schedules manually, which is powerful but time-consuming and error-prone, or rely on auto-schedulers that provide efficient search procedures at the cost of control [1, 2, 4, 23, 31]. This dichotomy cannot easily incorporate user input without writing the full schedule by hand.

These problems are particularly important for data-dependent and dynamic accelerators, like sparse accelerators where irregular iteration patterns, compressed storage formats, and fixed-function hardware units make scheduling highly workload-dependent [2, 11]. Such computations often lack a single optimal schedule across inputs and machines, since performance is tightly coupled to both the data and the target [5, 18, 26–28]. For example, two matrices with the same sparsity percentage but with different sparsity patterns could require different optimal schedules. In addition, the scheduling space is large and shaped by non-obvious validity constraints, making systematic exploration difficult [9]. In hardware-aware schedules, many of the most important scheduling decisions are ones domain experts understand well, but that auto-schedulers cannot reliably infer. As designers of scheduling languages, we may therefore want a system that can scaffold users in writing compiler schedules *without* removing access to any particular schedule.

We are not the first to argue for interactivity in optimizing compilers. Koehler et al. [15]'s *guided equality saturation* queries the user at particularly challenging points in an equality saturation workload; their goal is to be mostly-automated while occasionally involving the user and consequently does not ensure that user choices are guaranteed to succeed. Ikarashi et al. [13]'s *guided optimization* system Roly-poly enables users to follow a fixed-order workflow of i) choosing function scheduling locations and ii) tiling parameters for a dense image processing pipeline. Their system does not enable users to reach any possible program in the scheduling language, but, unlike guided equality saturation, it does guarantee that every option it shows is valid. Roly-poly users produced significantly improved schedules, but they were still outperformed by the auto-scheduler. These systems highlight a design space for interactive optimization and raise the question of what kinds of systems should exist beyond the manual and auto-scheduling paradigms.

In this paper, we ask what the interaction between the programmer and the scheduling system should look like. In particular, what guarantees can we provide on the programming process? *What guarantees do we even want?*

To study these questions in practice, we apply Programming by Navigation [22], an interactive synthesis framework that ensures reachability of all valid programs and prevents users from taking invalid steps, to the problem of scheduling in FuseFlow [19], a compiler for sparse machine-learning accelerators. While we demonstrate this idea using Programming by Navigation and FuseFlow, the core technique is general. Our prototype starts with a subset of FuseFlow's scheduling operations and guarantees that users can construct all valid schedules, and *only* the valid schedules, in this subset. This system illustrates how such guarantees can be instantiated in a real compiler for sparse accelerators. More broadly, we invite discussion about what interactive mechanisms and guarantees are most important for authoring schedules.

## 2 BACKGROUND

Programming by Navigation models synthesis problems as multi-round interactions in which a user incrementally refines a target


*LATTE '26, March 23, 2026, Pittsburgh, Pennsylvania, USA*

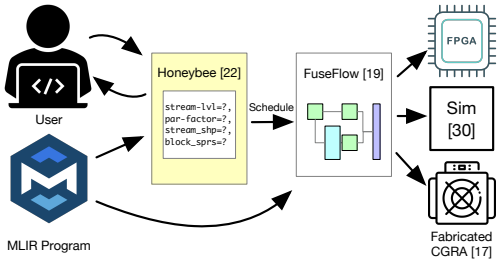


**Figure 1: Overview of Honeybee–FuseFlow scheduling.** The user provides an MLIR program to Honeybee and interactively fills the holes in the schedule. The resulting schedule and input program are passed to FuseFlow, which targets different hardware backends.

program [22]. The synthesizer maintains an in-progress sketch with holes as placeholders. Crucially, Programming by Navigation provides **Strong Completeness** and **Strong Soundness**, which guarantee that all valid programs are reachable and only valid next steps are shown, respectively. Together, these properties prevent common failure modes such as rendering valid programs inaccessible or leading users down rabbit holes that may never result in the desired expression. Honeybee is an implementation of Programming by Navigation for instantiating synthesis problems from a user-defined library of functions and validity constraints (which, for us, will be the building blocks of an interactive scheduling system).

FuseFlow is an end-to-end compilation system from PyTorch [3] with sparse annotations [6, 8] to streaming dataflow graphs that run on sparse accelerators [11] in MLIR [21]. These dataflow graphs can target fabricated hardware accelerators [17], a cycle-approximate hardware simulator [30], and map to FPGA backends. FuseFlow provides a scheduling language that lets users choose fusion granularity, dataflow ordering, parallelization, and sparse data-structure annotations for tensor expressions. Users specify fusion via `Fuse{}` regions in MLIR, select dataflow order by rewriting MLIR `Linalg` affine maps, and set parallelization and vectorization schedule parameters through a command-line interface. Parallelization tells FuseFlow which loops to parallelize and by how much, which controls the number of parallel lanes for each loop level, and vectorization tells the compiler the stream vectorization width and optionally whether to use a block-sparse vectorization mode.

## 3 HONEYBEE FOR FUSEFLOW SCHEDULING

We implement a Honeybee library for FuseFlow scheduling that produces stream parallelization and vectorization operators. Interactive Honeybee sessions require two inputs (defined via TOML files): a **program** and a **library**. The program contains the inputs and goal to solve. The library is the domain-specific language definition that describes the schedule and includes the steps that construct types and enforce constraints throughout the synthesis session.

Honeybee programs are generated by parsing the MLIR program before invoking FuseFlow. We extract the MLIR-level function names (or IDs) and the loop depth of the `Linalg` ops in the program. The loop depth is inferred by parsing the `iterator_types` list or falling back to the default provided by `linalg.matmul`. This information is used to produce constraints that are provided to the synthesizer. For example, the stream level must be less than or equal to the loop nest (you cannot pick a stream level that refers to a loop level that does not exist). Honeybee ensures that the user cannot construct a program where these constraints are violated, while retaining the ability to navigate to any program that does satisfy the constraints.

Passing the program and library to Honeybee and starting the synthesis session prompts the user to select scheduling options. The user fills the holes sequentially until they reach a completed schedule. The output of the synthesis session is a Python program that generates a JSON containing fields corresponding to the scheduling operators in the FuseFlow compiler.

## 4 RESULTS AND FUTURE WORK

We show our interactive synthesis tool on an MLIR program that computes a nested matrix multiplication (and more generally, on sparse kernels like SpMM where some loop dimensions are compressed by the sparse format).

As intuition, consider an SpMM loop nest where one dimension is sparse (e.g., CSR-format). Many plausible scheduling choices, such as parallelizing the compressed loop, are invalid or counterproductive. Manual scheduling becomes tedious and error-prone, and auto-scheduling can miss the specific decisions experts care about. Instead, we focus on demonstrating an interaction model that (i) never proposes invalid steps and (ii) still reaches any valid schedule expressible by the library.

In the interaction, Honeybee maintains an in-progress schedule sketch with holes denoted by ?N. The user fills holes by choosing from a menu of *valid next steps*. The user is presented with the ability to build a schedule, or to select a default:

```
Possible next steps:
  1) ?0 -> default_schedule(_metadata={})
  2) ?0 -> build_schedule(order=?1, _metadata={})
```

The default schedule produces scheduling primitives corresponding to no-ops in FuseFlow (i.e. no parallelization or vectorization enabled). If the user selects `build_schedule`, they advance to a loop-ordering choice.

We use FuseFlow's built-in dataflow ordering enumerator to show loop-order options:

```
Possible next steps:
  1) ?1 -> choose_loop_order(pass=?2,
         _metadata={order="i0 i1 i2 i4 i5 i6 i7 i8 i9 i3"})
  2) ?1 -> choose_loop_order(pass=?2,
         _metadata={order="i0 i1 i2 i4 i5 i6 i8 i7 i9 i3"})
  ...
```

The user then selects the parallelization factor for each loop in the computation, one at a time. They can optionally select default values for each stream level, allowing them to focus solely on the stream levels that are most important to parallelize.

```
Possible next steps:
  1) ?3 -> choose_default_par_factor(level=?4,
         _metadata={par_factor=1, stream_level=9})
  2) ?3 -> choose_par_factor_2(level=?4,
         _metadata={par_factor=2, stream_level=9})
  3) ?3 -> choose_par_factor_4(level=?4,
         _metadata={par_factor=4, stream_level=9})
  ...
```

One sparse-aware change we are encoding in Honeybee is to hide parallelization-factor choices for stream levels that correspond to sparse or compressed dimensions, inferred from encodings like CSR/DCSR. Honeybee's constraint system ensures that users do not parallelize these loops.

After repeating this process for all eligible loops in the program, the user selects vectorization scheduling parameters:

```
Possible next steps:
  1) ?24 -> choose_default_block_sparse(shape=?25,
          _metadata={block_sparse=False})
  2) ?24 -> choose_block_sparse_true(shape=?25,
          _metadata={block_sparse=True})
```

The scheduling flow outputs a Python program that can be executed to generate a JSON file containing the full schedule. The JSON file can be provided to the FuseFlow compiler directly to invoke the compiler with the described scheduling operations.

```
{
  "dataflow-ordering": "i0 i1 i2 i4 i5 i6 i7 i8 i9 i3",
  "stream-parallelizer": {
    "par_factors": [1, 1, 1, 1, 1, 4, 1, 1, 1, 4]
  },
  "stream-vectorizer": {
    "enable-block-sparse": true,
    "stream-shape": 1
  }
}
```

The output schedule is guaranteed to be a valid FuseFlow schedule in the sense that all parallelization factors and vector widths are bounded by the dimensions of the input tensors. More importantly, structuring the interaction as a sequence of decisions does not weaken the guarantees we care about. Within the subset of FuseFlow schedules captured by our Honeybee library, users can still reach every valid schedule, and every intermediate choice remains on a path to a valid completion. Thus, we can impose an ordering on schedule construction without giving up expressiveness.

This allows us to show smaller, more constrained sets of choices at each step, rather than forcing the user to reason about the full schedule all at once. To make this tradeoff concrete, we can quantify the size of the scheduling space induced by the subset of decisions currently exposed interactively in our Honeybee library:

$$|\mathcal{S}| \;=\; O! \cdot \prod_{\ell \in \mathcal{P}} Par_\ell \cdot V \cdot 2, \tag{1}$$

where $O!$ is the number of dataflow orders, $Par_\ell$ is the number of parallelization factors available at loop level $\ell$, $V$ is the number of vector shapes, and 2 is for the block sparse format option.

For the fused matmul example in this preliminary work, this restricted interactive space comprises

$$4! \times 8 \times 8 \times 2 \;=\; 3072 \tag{2}$$

possible schedules. Here, the factors correspond to the exposed choices for dataflow ordering, parallelization, vectorization shape, and the block-sparse toggle. Although 3072 is already large enough to make manual exploration cumbersome, it reflects only a small subset of the scheduling parameters available in practice. FuseFlow exposes many additional tunable knobs, and a broader interactive system would ideally allow users to navigate richer fusion decisions and lower-level scheduling parameters as well. To accomplish this, we would need to include support for fusion regions and sparse format selection, which are both currently works in progress. We therefore view this 3072-count space not as the full FuseFlow scheduling space, but as a preliminary interactive subspace that already demonstrates the value of interactive navigation.

Even in this restricted setting, the benefit of interaction is that the system does not present thousands of complete schedules. Instead, it forces the space into a sequence of small, local choices, where every choice is guaranteed valid and every step has a default choice.

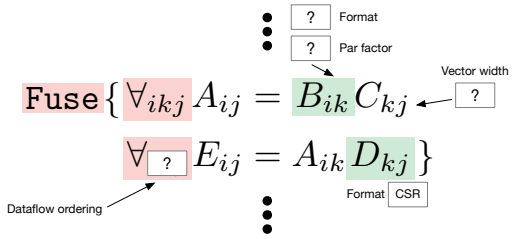


**Figure 2: Einsum expressions with an autoscheduler-selected Fuse region and filled in partial schedule annotated with holes for scheduling parameters to be filled with Honeybee.** User schedules are highlighted in red and sparse tensors in green. Each expression can have a distinct schedule.

Thus, users can focus on the handful of decisions they think are important without needing to fully understand the scheduling API.

Honeybee's type system allows us to carefully design the interaction between the user and the scheduling system. Schedule authors may have a better understanding of the local behavior of a program than its global behavior. We take this into account by imposing an ordering of scheduling decisions on the user that allows them to reason one level at a time.

At the same time, the exact interaction is not fixed. The same Programming by Navigation machinery could be used to formulate other interactions. For example, we could use Honeybee to show the user *all* possible schedules in one step, where each selection is a schedule with all holes filled out. We can also combine steps that are commonly selected together, i.e. by choosing to parallelize an inner loop while leaving the outer loops sequential.

We are currently exploring interoperability between the interactive synthesizer and the wealth of existing sparse auto-schedulers [2, 9, 18, 29] by allowing the user to invoke an auto-scheduler for holes of their choice while using interactive refinement to fill the others. This approach would provide fine-grained control over decisions users care about, while allowing them to defer other decisions to differing well-tested, automatic systems. More broadly, this is how we see interactive scheduling interoperating with existing auto-schedulers (including LLM-guided ones [18]). We further hope to enable more successful multi-objective optimization by letting users freely experiment with different combinations of scheduling refinements to navigate the Pareto frontier of optimizations.

## 5 CONCLUSION

As scheduling languages evolve beyond the manual scheduling vs. auto-scheduling dichotomy, we argue for further exploration of techniques for interactive compiler scheduling that balance automation with control. Within this space, the central questions lie in what these interactive systems should look like in practice and what user experience best serves the needs of accelerator designers and programmers. A key aspect of the design is the guarantees the system provides while authoring schedules and the notions of validity it enforces. The foundation of a good user experience for programming systems for scheduling languages will rest on the choices for these guarantees and the extent to which these guarantees match users' needs and prevent the pitfalls they are likely to run into.

# REFERENCES


[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[2] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. doi:10.1145/3519939.3523442

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[5] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proc. ACM Program. Lang.* 7, PLDI, Article 122 (June 2023), 26 pages. doi:10.1145/3591236

[6] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (Sept. 2022), 25 pages. doi:10.1145/3544559

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[8] Google. 2021. MLIR Sparsifier.

[9] Erik Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. 2023. BaCO: A Fast and Portable Bayesian Compiler Optimization Framework. *International Conference on Architectural Support for Programming Languages and Operating Systems* (April 2023).

[10] Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. 2025. Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 628–643. doi:10.1145/3696443.3708918

[11] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. doi:10.1145/3582016.3582051

[12] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[13] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. doi:10.1109/VL/HCC51201.2021.9576341

[14] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. *International Symposium on Code Generation and Optimization.*

[15] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL, Article 58, 32 pages. doi:10.1145/3632900

[16] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3192366.3192379

[17] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications. *IEEE Symposium on VLSI Technology and Circuits (VLSI)* (June 2024).

[18] Rubens Lacouture, Genghan Zhang, Konstantin Hossfeld, Tian Zhao, and Kunle Olukotun. 2025. LLM-Guided Autoscheduling for Large-Scale Sparse Machine Learning. In *Machine Learning for Systems 2025*. https://openreview.net/forum?id=7H9qWe8lLO

[19] Rubens Lacouture, Nathan Zhang, Ritvik Sharma, Marco Siracusa, Fredrik Kjolstad, Kunle Olukotun, and Olivia Hsu. 2025. FuseFlow: A Fusion-Centric Compilation Framework for Sparse Deep Learning on Streaming Dataflow. arXiv:2511.04768 [cs.LG] https://arxiv.org/abs/2511.04768

[20] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. doi:10.1145/3289602.3293910

[21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[22] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. doi:10.1145/3729264

[23] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. doi:10.1145/2897824.2925952

[24] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2023. *Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines* (1 ed.). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3596711.3596751

[25] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[26] Ryan Swann, Muhammad Osama, Karthik Sangaiah, and Jalal Mahmud. 2024. Seer: Predictive runtime kernel selection for irregular problems. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 133–142.

[27] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices* 50, 6 (2015), 521–532.

[28] Jaeyeon Won, Charith Mendis, Joel S Emer, and Saman Amarasinghe. 2023. Waco: learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 920–934.

[29] Bobby Yan, Alexander J Root, Trevor Gale, David Broman, and Fredrik Kjolstad. 2026. Fast Autoscheduling for Sparse ML Frameworks. *International Symposium on Code Generation and Optimization* (February 2026).

[30] Nathan Zhang, Rubens Lacouture, Gina Sohn, Paul Mure, Qizheng Zhang, Fredrik Kjolstad, and Kunle Olukotun. 2024. The Dataflow Abstract Machine Simulator Framework. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 532–547. doi:10.1109/ISCA59077.2024.00046

[31] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.