PROGRAM SYNTHESIS FOR SCIENTIFIC COMPUTING

HAL FINKEL
Argonne National Laboratory

IGNACIO LAGUNA
Lawrence Livermore National Laboratory

August 4–5, 2020

Report of the Workshop on Program Synthesis for Scientific Computing

August 4–5, 2020
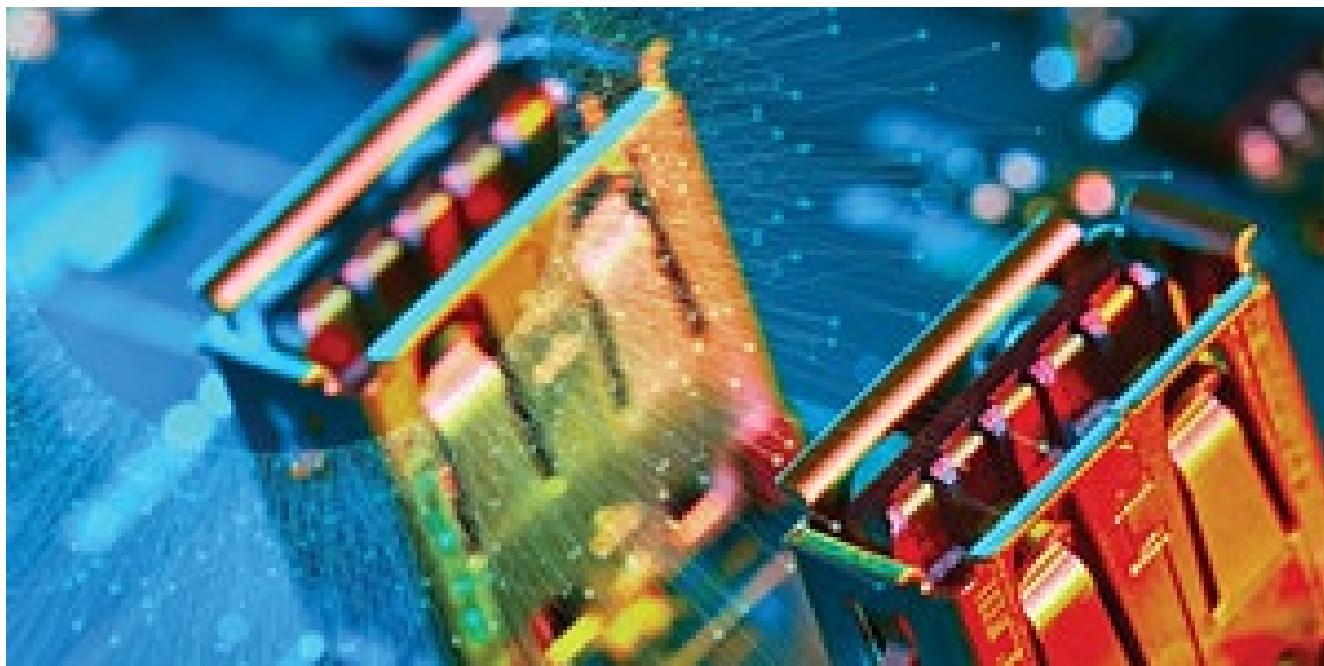
Organizers:
Hal Finkel, Argonne National Laboratory
Ignacio Laguna, Lawrence Livermore National Laboratory

Participants:

Maaz Ahmad (University of Washington)
Alex Aiken (Stanford University)
Aws Albarghouthi (University of Wisconsin - Madison)
Farhana Aleen (NVIDIA Corporation)
Francis Alexander (Brookhaven National Laboratory)
Rajeev Alur (University of Pennsylvania)
Saman Amarasinghe (Massachusetts Institute of Technology)
Todd Anderson (Intel Corporation)
Pavan Balaji (Argonne National Laboratory)
Prasanna Balaprakash (Argonne National Laboratory)
Ras Bodik (University of Washington)
James Bornholt (University of Texas at Austin)
Bill Carlson (Center for Computing Sciences)
Barbara Chapman (BNL & Stony Brook University)
François Charton (Facebook)
Swarat Chaudhuri (University of Texas at Austin)
Estee Chen (University of Pennsylvania)
Alvin Cheung (University of California, Berkeley)
Amazon & University of Chicago (spertus.com)
Taylor Childers (Argonne National Laboratory)
Ravi Chugh (University of Chicago)
Valentin Clement (Oak Ridge National Laboratory)
Loris D'Antoni (University of Wisconsin - Madison)
Arnab Das (University of Utah)
Isil Dillig (University of Texas at Austin)
Johannes Doerfert (Argonne National Laboratory)
Krzysztof Drewniak (University of Washington)
Anshu Dubey (Argonne National Laboratory)
Souradeep Dutta (University of Pennsylvania)
Markus Eisenbach (Oak Ridge National Laboratory)
Nur Aiman Fadel (Swiss National Supercomputing Centre)
Kyle Felker (Argonne National Laboratory)
Ian Foster (Argonne National Laboratory)
Franz Franchetti (Carnegie Mellon University)
Milos Gligoric (University of Texas at Austin)
Cindy Rubio Gonzalez (University of California, Davis)
Ganesh Gopalakrishnan (University of Utah)
Justin Gottschlich (Intel Labs & University of Pennsylvania)
Vinod Grover (NVIDIA Corporation)
Mary Hall (University of Utah)
Niranjan Hasabnis (Intel Corporation)
Amaury Hayat (Rutgers University)
Thomas Helmuth (Hamilton College)
Michael Heroux (Sandia National Laboratories)
Paul Hovland (Argonne National Laboratory)
Justin Hsu (University of Wisconsin - Madison)
Jan Hueckelheim (Argonne National Laboratory)
Roshni Iyer (University of California, Los Angeles)
John Jacobson (University of Utah)
Xiao-Yong Jin (Argonne National Laboratory)
Beau Johnston (Oak Ridge National Laboratory)
Vinu Joseph (University of Utah)
Ian Karlin (Lawrence Livermore National Laboratory)
Vineeth Kashyap (GrammaTech, Inc.)
Samuel Kaufman (University of Washington)
Sarfraz Khurshid (University of Texas at Austin)
Jungwon Kim (Oak Ridge National Laboratory)
Martin Kong (University of Oklahoma)
Jaehoon Koo (Northwestern University)
Siddharth Krishna (Microsoft Corporation)
Michael Kruse (Argonne National Laboratory)
Ignacio Laguna (Lawrence Livermore National Laboratory)
Jacob Lambert (University of Oregon)
Edward Lee (University of California, Berkeley)
Seyong Lee (Oak Ridge National Laboratory)
Richard Lethin (Reservoir Labs)
Dmitry Liakh (Oak Ridge National Laboratory)
Nevin Liber (Argonne National Laboratory)

Zhengchun Liu (Argonne National Laboratory)
Alexey Loginov (GrammaTech, Inc.)
Nuno Lopes (Microsoft Corporation)
Abid Malik (Brookhaven National Laboratory)
Ruben Martins (Carnegie Mellon University)
James McDonald (Kestrel Institute)
Thirimadura Mendis (Massachusetts Institute of Technology)
Abdullah Muzahid (Texas A&M University)
Mayur Naik (University of Pennsylvania)
Sri Hari Krishna Narayanan (Argonne National Laboratory)
Brandon Neth (University of Arizona)
Matthew Norman (Oak Ridge National Laboratory)
Boyana Norris (University of Oregon)
Cathie Olschanowsky (Boise State Univeristy)
Cyrus Omar (University of Michigan)
Peter-Michael Osera (Grinnell College)
Pavel Panchekha (University of Utah)
Sheena Panthaplackel (University of Texas at Austin)
Eun Jung Park (EJ) (Los Alamos National Laboratory)
Gilchan Park (Brookhaven National Laboratory)
Madhusudan Parthasarathy (UIUC)
Tharindu Patabandi (University of Utah)
Hila Peleg (University of California, San Diego)
Ruzica Piskac (Yale University)
Nadia Polikarpova (University of California, San Diego)
Tobi Popoola (Boise State Univeristy)
Xiaokang Qiu (Purdue University)
Arjun Radhakrishna (Microsoft Corporation)
Jonathan Ragan-Kelley (Massachusetts Institute of Technology)
Krishnan Raghavan (Argonne National Laboratory)
Yihui Ren (Brookhaven National Laboratory)
Thomas Reps (University of Wisconsin - Madison)
Kamil Rocki (Cerebras Systems)
Jose Rodriguez (Intel Corporation)
Tiark Rompf (Purdue University)
Randi Rost (Intel Corporation)
Roopsha Samanta (Purdue University)
Mark Santolucito (Barnard College)
Vivek Sarkar (Georgia Institute of Technology)
Markus Schordan (Lawrence Livermore National Laboratory)
Eric Schulte (GrammaTech, Inc.)
Koushik Sen (University of California, Berkeley)
Srinivasan Sengamedu (Amazon)
Dolores Shaffer (Science and Technology Associates, Inc.)
Min Si (Argonne National Laboratory)
Douglas Smith (Kestrel Institute)
Armando Solar-Lezama (Massachusetts Institute of Technology)
George Stelle (Los Alamos National Laboratory)
Michelle Strout (University of Arizona)
Yizhou Sun (University of California, Los Angeles)
Joseph Tarango (Intel Corporation)
Zachary Tatlock (University of Washington)
Valerie Taylor (Argonne National Laboratory)
Supun Tennakoon (Purdue University)
Aditya Thakur (University of California, Davis)
Rajeev Thakur (Argonne National Laboratory)
Jesmin Jahan Tithi (Intel Corporation)
Jeffrey Vetter (Oak Ridge National Laboratory)
Brice Videau (Argonne National Laboratory)
Ashwin Vijayakumar (Intel Corporation)
Fei Wang (Purdue University)
Wei Wang (University of California, Los Angeles)
Xinyu Wang (University of Michigan)
Fangke Ye (Georgia Institute of Technology)
Jisheng Zhao (Georgia Institute of Technology)
Shengtian Zhou (Intel Corporation)
Tong Zhou (Georgia Institute of Technology)

The Workshop on Program Synthesis for Scientific Computing was held virtually on August 4–5 2020.

INTRODUCTION

Computational methods have become a cornerstone of scientific progress, with nearly every scientific discipline relying on computers for data collection, data analysis, and simulation. As a result, significant opportunities exist to accelerate scientific discovery by accelerating the development and execution of scientific software.

The first priority research direction of the U.S. Department of Energy's (DOE's) report on extreme heterogeneity [101] and the ninth section of the DOE community's *AI for Science* report [93], among other sources, highlight the need for research in AI-driven methods for creating scientific software. This workshop expands on these reports by exploring how *program synthesis*, an approach to automatically generating software programs based on some user intent [35, 36]—along with other high-level, AI-integrated programming methods—can be applied to scientific applications in order to accelerate scientific discovery.

We believe that the promise of program synthesis (also referred to as *machine programming* [35]) for the scientific programming domain is at least the following:

- Significantly reducing the temporal overhead associated with software development. We anticipate increases in productivity of scientific programming by orders of magnitude by making all parts of the software life cycle more efficient, including reducing the time spent tracking down software quality defects, such as those concerned with correctness, performance, security, and portability [3, 4, 39]

- Enabling scientists, engineers, technicians, and students to produce and maintain high-quality software as part of their problem-solving process without requiring specialized software development skills [77]

Program synthesis is an active research field in academia, national labs, and industry. Yet, work directly applicable to scientific computing, while having some impressive successes, has been limited. This report reviews the relevant areas of pro-
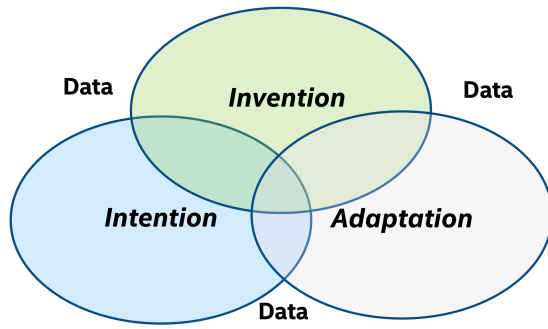
- **Intention**: Discover intent of programmer; lift meaning from software
- **Invention**: Create new algorithms and data structures
- **Adaptation**: Evolve in a changing hardware/software world

Figure 1: The Three Pillars of Machine Programming (credit: Gottschlich et al. [35]).

gram synthesis work, discusses successes to date, and outlines opportunities for future work.

Background on Program Synthesis

Program synthesis represents a wide array of machine programming [35] techniques that can greatly enhance programmer productivity and software quality characteristics, such as program correctness, performance, and security. Specifically, program synthesis incorporates techniques whereby the following may occur:

- The desired program behavior is specified, but the (complete) implementation is not. The synthesis tool determines how to produce an executable implementation.

- The desired program behavior or any partial implementation is ambiguously specified. Iteration with the programmer, data-driven techniques, or both are used to construct a likely correct solution.

In *"The Three Pillars of Machine Programming"* nomenclature, program synthesis is classified in the space of *intention* [35] (see Figure 1). The goal of intention is to provide programmers (and non-programmers) with ways to communicate their ideas to the machine. Once these goals have been expressed, the machine constructs (*invents*) a higher-order representation of the user's intention. Next, the machine programming system is

free to *adapt* the higher-order invented software representation to a lower-order representation that is specific to the user's unique software and hardware environment. This process tends to be necessary to ensure robust software quality characteristics such as performance and security.

When the implementation—the *how* of the program—is unambiguously specified, the process of translating that specified implementation into an executable program is called compilation.

Good compiler technology is essential to scientific programming; and as discussed later in this report, opportunities exist for enhancing compiler technology to better enable program synthesis technology. These two areas, program synthesis and compilers, inform each other; and as we look toward the future, state-of-the-art programming environments are likely to contain both synthesis and compilation capabilities.

> Program synthesis is expected to reduce software development overhead and increase confidence in the correctness of programs. Compiler technology will be essential to enable this technology.

The first program-synthesis systems focused on the automated translation of mathematically precise program specifications into executable code. These systems performed what is sometimes called *deductive synthesis* and often functioned by trying to match parts of the specification to a library of relevant implementation techniques.

With the advent of strong satisfiability modulo theories (SMT) solvers, program synthesis had an important new technology on which to build. An SMT solver can naturally produce counterexamples to an inconsistent set of mathematical assertions or produce an assertion that no counterexamples exist, which is useful for several kinds of verification and synthesis tasks. Writing mathematical specification is often subtle, however, and considerable attention in the synthesis community has focused on *inductive synthesis*: the synthesis of programs based on behavioral examples [37].

Of course, systems can be both deductive and

inductive, which is useful because sometimes a programmer knows some of the desired properties but wishes to fill in the remaining information necessary to construct the program using examples. For example, techniques for *type-and-example-directed synthesis* deductively use types provided by the programmer to guide an inductive search for missing program fragments that satisfy the given examples [71, 60]. Combining inductive synthesis with SMT solver iteration, using solver-generated counterexamples to guide the synthesis process, has also been a fruitful area: counterexample-guided inductive synthesis (CEGIS) has produced exciting results over the past decade.

CEGIS and related techniques are good at refining potential solutions that are structurally close to a correct answer, but the techniques have more difficulty in searching the unbounded space of potential program structures. Evolutionary algorithms have made important contributions to this problem, especially those that encourage the selection of specialists (potential solutions that work for some, but not necessarily all, solution objectives).

The deep learning revolution has led to significant advancements in program synthesis as well. The space of potential program structures can be explored by using differentiable programming, reinforcement learning, and other machine learning techniques. Deep learning has also made practical the incorporation of natural language processing into the synthesis process and the generation of natural language comments as part of the synthesis output. Machine learning techniques now power advanced autocomplete features in various development environments, and looking toward the future, advanced tools can explore more than single-statement completions.

As program-synthesis technology, driven by advanced deep learning, evolutionary, and verification techniques, moves toward tackling real-world programming problems, imparting the resulting productivity gains to scientific programming will require techniques and capabilities that might not be required for other domains. In this report, we explore the state of and challenges in scientific programming and how research in program synthesis technology and synergistic research in

compiler technology might be directed to apply to scientific-programming tasks.

CURRENT STATE OF SCIENTIFIC APPLICATION DEVELOPMENT

Scientific application development is essential to scientific progress; and yet, while advances in both scientific techniques and programming tools continue to improve programmer productivity, creating state-of-the-art scientific programs remains challenging.

Software complexity has increased over the past decades at an astounding rate, with many popular applications and libraries containing tens of millions of lines of code. As shown in Figure 2, scientific development has not been immune from this increase in complexity.

General software infrastructures, along with algorithmic and mathematical techniques, have become increasingly sophisticated. Hardware architectures and the techniques necessary to exploit them have also become increasingly sophisticated, as has the science itself. Managing the resulting complexity is difficult even for the most experienced scientific programmers. Moreover, a lot of scientific programming is not done by experienced programmers but by scientific-domain students and recent graduates with only a few years of experience [63]. As a result, new programmers on a project have difficulty reaching high levels of productivity.

The Largest Challenges

Code development itself is labor intensive; and in order to develop scientific applications, significant portions of the development require direct input from domain experts. These applications often contain critical components that are mathematically complex; and hence developers require advanced mathematical abilities, strong programming skills, and a good understanding of the science problem being solved.

Separations of concerns are common, and not every scientific-software developer implements every mathematical technique from scratch. Nevertheless, developers need sufficient knowledge of
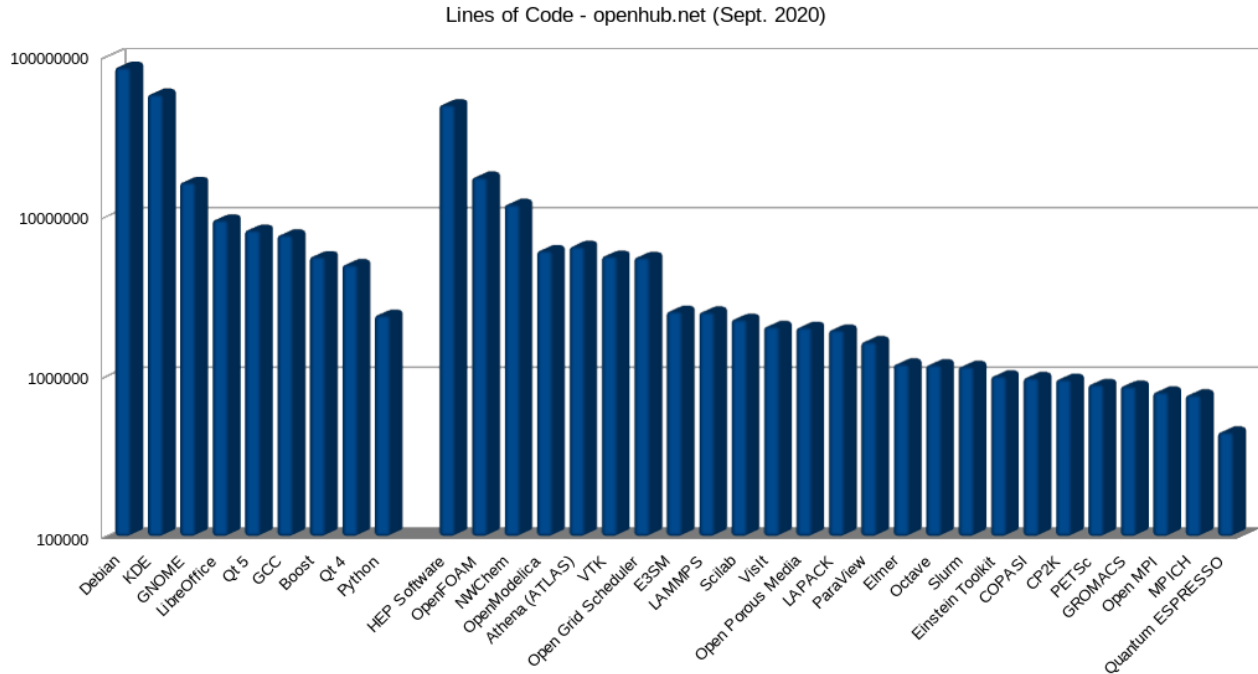
Figure 2: Number of source lines of code in various packages: general packages on the left, scientific packages on the right. Nearly all data from openhub.net (Sept. 2020).

the relevant mathematical techniques to select and use applicable libraries.

> Scientific software has become increasingly complex over the years. Tools and techniques to improve productivity in this space are desperately needed.

Moreover, often the complete design for the application cannot be specified up front. One may not know ahead of time what grid resolutions, discretization techniques, solvers, and so on will work best for the target problems. Instead, development is an iterative process that is part of the scientific investigatory process. As the science evolves, the target problem might change as well, necessitating significant changes to the application design.

While application development for dynamic consumer markets also faces challenges with evolving requirements, scientific software must often meet tight performance and mathematical requirements, where large changes in design must

be implemented quickly by small development teams, imparting unique needs for productivity-enhancing tools in this space.

Adapting to New Hardware: Scientific computing applications tend to require high computational performance and, as a result, are structured and tuned to maximize achieved performance on platforms of interest. However, computational performance is maximized on cutting-edge hardware, and cutting-edge hardware has been evolving rapidly. As a result, applications have had to adapt to new CPU features, such as SIMD vector registers, GPU accelerators, and distribution over tens of thousands of independent nodes. These architectures continue to evolve, with corresponding changes to their programming models, and applications must adapt in turn. If significant work is invested in turning for a particular architecture, as is often the case [7], repeating that level of work for many different kinds of systems is likely infeasible.

To maintain developer productivity in the face of a variety of target architectures, the community has placed significant focus on programming environ-

ments that provide some level of *performance portability*. The idea is that reasonable performance, relative to the underlying system capabilities for each system, can be obtained with no, or minimal, source code changes between systems.

What qualifies as reasonable performance and what qualifies as minimal changes to the source code are hotly debated and, in the end, depend on many factors specific to individual development teams. Nevertheless, overall goals are clear, and these motivate the creation of compiler extensions (e.g., OpenMP), C++ abstraction libraries (e.g., Kokkos [32]), and domain-specific languages (e.g., Halide [76]).

Furthermore, the design of these portable abstractions is largely reactionary; and while one can anticipate and incorporate some new hardware features before the hardware is widely available, often the best practices for using a particular hardware architecture are developed only after extensive experimentation on the hardware itself. This situation leads to a natural tension between the productivity gain from using portable programming models and the time-to-solution gain potentially available from application- and hardware-specific optimizations.

The ability to perform autotuning on top of these technologies has been demonstrated to enhance performance significantly [11]. Some systems (e.g., Halide) were specifically designed with this integration in mind. However, autotuning brings with it a separate set of challenges that make deployment difficult. If the autotuning process is part of the software build process, then the build process becomes slow and potentially nondeterministic.

On the other hand, if the autotuning process produces artifacts that are separately stored in the source repository, then the artifacts need to be kept up to date with the primary source code, a requirement made more difficult by the fact that not all developers have access to all of the hardware on which the artifacts were generated. A potential solution to these challenges is to perform the autotuning while the application is running, but then the time spent autotuning must be traded against the potential performance benefits. Nevertheless,

the optimal tuning results sometimes depend on the state of the application's data structures (e.g., matrix sizes), and these can change during the course of a long-running process, providing an additional advantage to during-execution autotuning and autotuning procedures that can make use of detailed profiling data.

High performance can often be obtained by using different implementations of the same algorithm, and the aforementioned frameworks naturally apply to this case. Sometimes, however, especially when accelerators are involved, different algorithms are needed in order to obtain acceptable performance on different kinds of hardware. In recognition of this reality, a number of capabilities have been explored for supporting algorithmic variants that can be substituted in a modular fashion as part of the porting process (e.g., OpenMP metadirectives, PetaBricks [9]). Having multiple available algorithms for tasks within an application, however, makes testing and verification of the application more difficult. Development and maintenance are also more expensive because each algorithmic variant must be updated as the baseline set of required features expands over time and as defects are fixed.

> The ability of scientific codes to quickly adapt to new hardware is increasingly challenging. While performance-portable programming models and autotuning help, more advanced end-to-end capabilities are needed to adapt algorithms and data structures to new environments.

Data Movement Cost: The performance of many scientific applications is dominated by data movement. Customizing temporary storage requirements to architectures and application needs has demonstrated promising performance improvements [67]. Previous work combined scheduling transformations within a sparse polyhedral framework and dataflow graphs to enable human-in-the-loop architecture-specific optimization [29]. Because of large variations in the design of memory subsystems, optimizing the actual layout of data in memory can reduce data movement as

well as reduce its cost. As one example, bricks—mini-subdomains in contiguous memory organized with adjacency lists—are used to represent stencil grids [110, 111]. Bricks reduce the cost of data movement because they are contiguous and therefore decrease the number of pages and cache lines accessed during a stencil application. Through a layout of bricks optimized for communication, we also eliminate the data movement cost of packing/unpacking data to send/receive messages.

Scheduling: A large fraction of scientific applications are still written in legacy code such C/C++ and require mapping to multithreaded code either manually or by autoparalellizing compilers. Performant execution of such legacy codes is dependent on good task scheduling. While most scheduling techniques perform statically, Aleen et al. [6] showed that orchestrating dynamically (by running a lightweight emulator on the input-characterization graph extracted from the program) can better balance workloads and provide further improvement over static scheduling.

Polyhedral Multiobjective Scheduling: As a step toward achieving portable performance, Kong and Pouchet [48] recently proposed a extensible kernel set of integer linear program (ILP) objectives that can be combined and reordered to produce different performance properties. Each ILP objective aims to maximize or minimize some property on the generated code, for instance, minimizing the stride penalty of the innermost loop. However, this work did not address how to select the objectives to embed into the ILP. More recently, Chelini et al. [20] proposed a systematic approach to traverse the space of ILP objectives previously defined by first creating an offline database of transformations. Such a database is constructed from a number of input cases exhibiting different dependence patterns, which are then transformed via all the possible 3-permutations of ILP objectives from the kernel set of transformations. The resulting transformed codes are then analyzed to extract specific code features that are stored together with the input dependencies and the transformations used (see Fig. 3). Later, during the compilation phase, the database is queried to adaptively select ILP objectives to embed. This process is illustrated
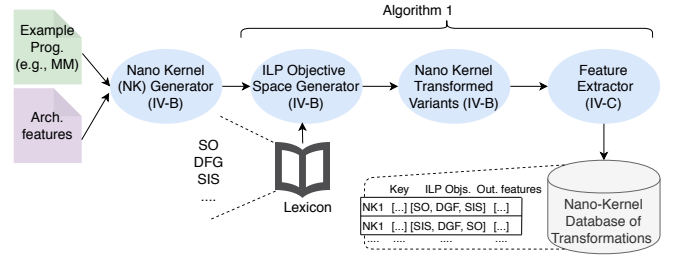
in Fig. 4.

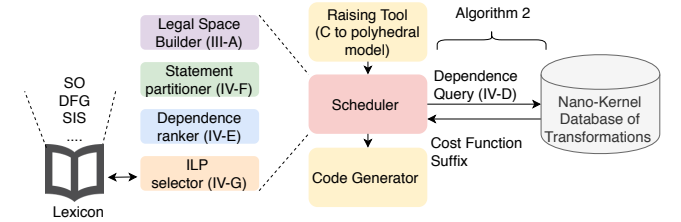

Figure 3: Offline Database Construction

Figure 4: Adaptive Scheduling and Objective Selection (Database Querying).

Testing and Verification: The correctness of scientific-computing applications is critical because the scientific enterprise depends on predictive computational techniques. Especially in areas where the results produced by scientific applications are used to ensure safety or otherwise inform public policy, incorrect results can cause serious problems. In general, science and engineering are competitive fields, and erroneous results put those depending on them at a relative disadvantage. Across the board, productivity can be severely hampered by application crashes and the time spent diagnosing and fixing misbehaving code.

Writing tests for scientific applications is often difficult and time consuming. As in any other software project, individual components should be tested with unit tests, and in addition, application-level tests are essential. If tests represent a fixed set of known input-output pairs for each component or for the application, covering all of the various combinations of allowed features and behaviors generally requires an exponential number of test cases. For tests that, individually, automatically explore more of the allowed state space, verifying the correct behavior is difficult. Programmers of-

ten fall back on verifying only invariants, not the complete output itself. This approach is especially true for physical-simulation results where exact answers are not known. Invariants (where they exist) such as conservation of energy and the application not crashing might be the only things that the tests actually end up checking. Tests providing higher-confidence verification, including comparisons with competing applications, and numerical-convergence analysis are often not automated and hence are performed manually only on irregular occasions (e.g., just before a major release). Problems discovered during these manual tests, as with other problems discovered late in the development cycle, are often difficult to diagnose and correct. The following are some notable challenges.

- **Numerics:** Many physical systems exhibit nontrivial sensitivity to their initial conditions, and thus small differences from truncation error, round-off error, and other small perturbations legitimately cause large differences in the final results. Verification in these cases is subtle, sometimes relying on statistical properties to compare with known solutions and sometimes relying on physical invariants; but even so, picking good thresholds for numerical comparisons is often done by manual experimentation and rules of thumb.

- **Asynchronous Execution:** Modern hardware demands the use of concurrent, and often parallel, execution in order to take advantage of the available computational capabilities. This makes testing and verification difficult because concurrent execution is nearly always nondeterministic. It is difficult to be sure that a particular algorithmic implementation is free from race conditions and other constraint violations under all possible execution scenarios: only a finite number, often a small part of the overall space of possibilities, are exhibited during testing. Pairing that testing with special programs designed to detect race conditions (e.g., valgrind, Thread Sanitizer) can help, but these programs add overheads and thus additional trade-offs into the overall testing process. Asynchronous execution frameworks (e.g., OpenMP tasks) generally depend on programmer-declared de-

pendencies to ensure valid task scheduling, and mistakes in these dependency declarations are sometimes difficult to detect.

- **Large Scale:** Scientific simulations often need to run at large scale in order to exhibit behaviors relevant for testing. Should something go wrong, debugging applications running in parallel on tens of thousands of nodes is difficult. Runs of an application, which are often dispatched from batch queues at unpredictable times, are expensive to recreate; and while interactive debugging sessions can be scheduled, turnaround times for large-scale reservations are not fast, and even state-of-the-art debugging tools for parallel applications have difficulty scaling to large systems. In some cases, state capture and mocking can be used to replicate and test relevant behaviors at smaller scales, but these are time consuming to implement.

- **System Defects:** At leadership scale, systems and their software push the state of the art and, as a result, may themselves have defects that lead to incorrect application behaviors. For the largest machines, vendors are often unable to test their software (e.g., their MPI implementation) at the full scale of the machine until that machine is assembled in the customer's data center. Early users of these machines frequently spend a lot of time helping track down bugs in hardware, system software, and compilers. For these users, the benefit from being able to run unprecedented calculations provides the motivation to continue even in the face of these kinds of issues, but getting things working at large scale is often more difficult than one might imagine.

- **Performance:** It is often desirable for testing to cover not only correctness but also performance. Application performance in scientific computing is often a critical requirement because only a limited number of core-hours are generally available in order to carry out particular calculations of interest. Testing of performance is difficult, however, both because many testing systems run many processes in parallel, making small-scale performance measurements noisy, and because many per-

formance properties can be observed only with large problem sizes at larger scales. Special tests that extrapolate small-scale performance tests can sometimes be constructed, but these are often done manually because the automation would involve even more work.

- **Resource Management:** As scientific software becomes more complex, problems often appear due to unfortunate interactions between different components. A common issue is resource exhaustion, where resources such as processor cores, memory, accelerators, file handles, and disk space, which are adequately managed by the different components in isolation, are not managed well by the composition of the components. For example, different components often allocate system memory with no regard for the needs of other components. Dedicated tests for resource usage can be written, but in practice this kind of testing is also performed manually. Many of these issues surface only when problems are being run at large scale.

Requirements for Automation

Increasing the amount of automation in the scientific application development process can increase programmer productivity, and program synthesis can play a key role. In order to be truly helpful, automated tools need to be integrated into the iterative and uncertain scientific discovery process. Experience has taught us that a number of important factors should be considered.

- **Separation of Concerns:** Tools must be constructed and their input formats designed recognizing that relevant expertise is spread across different users and different organizations. A domain scientist may understand very well the kinds of mathematical equations that need to be solved, an applied mathematician may understand very well what kinds of numerical algorithms best solve different kinds of equations, and a performance engineer may understand very well how to tune different kinds of algorithms for a particular kind of hardware; but these people might not work together directly. As a result, it must be possible to compartmentalize the relevant knowledge provided to the tool, to enable both reuse and independent development progress. To the extent that tools support providing mathematical proofs of correctness, the information needed for these proofs should be providable on component interfaces, so the modularity can improve the efficiency, understandability, and stability of the proof process.

- **Testing, Verification, Debugging:** Tools require specific features in order to assist with verification and debugging. Any human-generated input can, and likely will, contain mistakes. Mistakes can take the form of a mismatch between the programmer's intent [35] and the provided input and can also stem from the programmer's overlooking some important aspect of the overall problem. Moreover, tools themselves can have defects. Thus, tools must be constructed to maximize the ability of programmers to find their own mistakes and isolate tool defects [56]. To this end, tools must generally produce information on what they did, and why, and provide options to embed runtime diagnostics helpful for tracking down problems during execution. FPDetect [28] is an example of a tool providing sophisticated, low-overhead diagnostics that might be integrated with an automation workflow. Another example of synthesizable error detectors that can trap soft errors as well as bugs associated with incorrect indexing transformations is FailAmp [19], which, in effect, makes errors more manifest for easier detection.

- **Community and Support:** State-of-the-art science applications are complex pieces of software, often actively used for decades, and some are developed by large communities. Tools automating this development process need to integrate with existing code bases and support the complexity of production software. Actively used tools need to be supported by an active team. The productivity loss from issues with an undersupported tool, whether defects or missing features, can easily overwhelm the gains from the tool itself. Moreover, tools that generate hardware-

specific code need continual development to support the latest hardware platforms. All tools need to evolve over time to incorporate updated scientific and mathematical techniques. Support and adoption are helped by using tools that integrate with popular languages and programming environments, such as C++ and C++ libraries (e.g., Kokkos). Tools making use of production-quality compiler components (e.g., Clang as a library) tend to be best positioned for success. Sometimes, however, adoption of such tools is hampered by existing code bases in older or less widely used languages (e.g., C, Fortran), and an opportunity exists for automation tools to assist with translating that code to newer languages.

PROGRAM SYNTHESIS

Existing Work in Scientific Programming

In scientific computing, domain-specific languages have a long history, and scientists and engineers have increased the productivity of programming by creating specialized translators from mathematical expressions to code in C, C++, or Fortran. While many of these tools have an implicit implementation specification, they are instructive in the synthesis context as demonstrations of the kinds of high-level descriptions of mathematical calculations that programmers find useful. Claw [24]/GridTools [72] (stencil generation for climate modeling), Kranc [43] (stencil generation for numerical relativity), FireDrake [78]/FEniCS [57] (for the generation of finite-element solvers for partial-differential equations), lbmpy [16] (for the generation of lattice Boltzmann simulations), TCE [41] (for the generation of tensor-contraction expressions), and many others have demonstrated the utility of code generation from physical equations. Likewise, for traditional mathematical kernels, high-level code generators have been produced (e.g., SPIRAL [33], LGen [91], Linnea [14], TACO [47], Devito [54]), and the codelet generator in FFTW [34]). Some of these tools, such as SPIRAL, perform synthesis as well as code generation because they discover new algorithms from the exploration of relevant mathematical properties.

Autotuning is a common technique used to produce high-performance code, and machine learning can be used to dynamically construct surrogate performance models to speed up the search process. Autotuning can be used simply to choose predetermined parameter values or to explore complex implementation spaces and anything between. Stencil code generators such as PATUS [22] and LIFT [92] use autotuning to find the best tile sizes. ATLAS [104] uses this technique to select implementation parameters for linear algebra kernels. Recent work, making use of the ytopt [12] autotuning framework, has explored directly searching complex hierarchical spaces of loop nest transformations, as shown in Figure 5, using a tree search algorithm [49]. Some domain-specific languages, such as Halide, have been designed with a separate scheduling language that can be used by this kind of autotuning search technique.

To synthesize code that examines runtime data to perform optimizations, inspector/executor strategies [64, 84] employ inspector code that at runtime determines data reorderings, communication schedules for parallelism, and computation reorderings that are then used by the transformed executor code [83, 94, 31, 65, 62, 38, 105, 15]. Strout and others have developed inspector/executor strategies such as full sparse tiling that enable better scaling on the node because of reduced memory bandwidth demands [96, 97, 30, 80]. Ravishankar et al. [81] composed their distributed-memory parallelization inspector/executor transformation with affine transformations that enable vector optimization. Compiler support for such techniques consists of program analyses to determine where such parallelization is possible and the compile-time insertion of calls to runtime library routines for performing aspects of the inspector [79].

Some early work applying general-purpose synthesis techniques to scientific computing problems has appeared. For example, AccSynt [25] applied enumerative synthesis to the problem of generating efficient code for GPUs. AutoPandas [17] uses machine-learning-based search to automate the programming of data-table-manipulating programs. Lifting, the process of extracting a high-level model from the source code of an existing
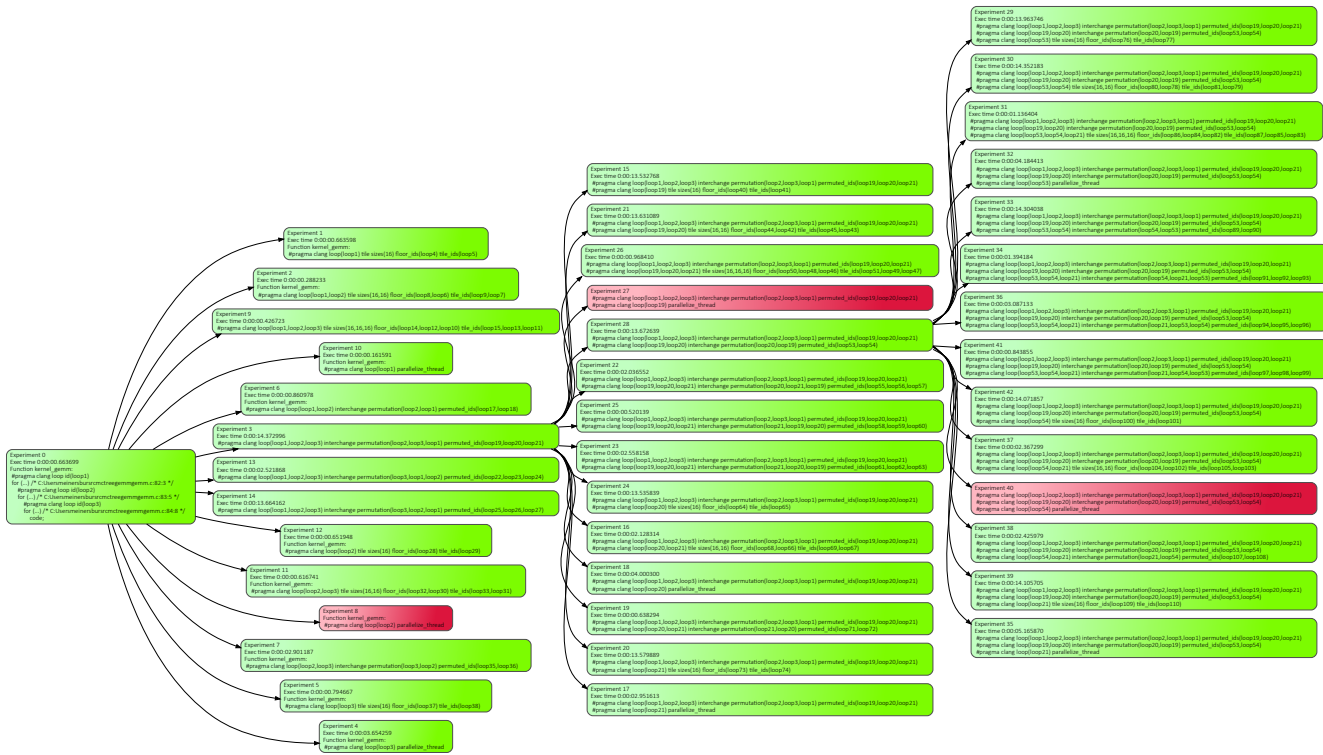
Figure 5: Autotuning search space for composed loop transformations

implementation, is important for handling legacy code and extracting knowledge from existing code bases [46, 21, 1]. Dexter [2], for example, can translate C++ kernel implementations into Halide, and OpenARC [51] can automatically translate existing OpenACC/OpenMP programs to OpenCL specialized for FPGAs.

Existing Work in Other Areas

A lot of existing work on program synthesis has focused on general programming tasks, such as writing functions that operate on common data structures like strings, lists, and trees [71, 60]. Recent work has tended to focus on tools that use natural language, behavioral examples, or both as input. The most widely deployed synthesis system is perhaps the Flash Fill feature in Microsoft Excel 2013+. Flash Fill can synthesize a small program to generate some column of output data given some columns of input data and some filled-in examples. The combination of deep learning with the large amount of publicly available source code (e.g., on GitHub) has provided opportunities for machine learning methods to capture knowledge from existing code bases at a massive scale. The resulting models can then be used for relatively simple tasks, such as autocomplete capabilities in editors, but can also drive complex search techniques in sophisticated synthesis systems. Past academic work in restricted domains, such as SQLizer [106], which translates natural languages to SQL queries, has begun inspiring commercial implementations. One interesting aspect of mining data from version-control repositories is that not only is the code itself available but its development history can also provide critical data. Systems that learn from past commits have been able to learn likely defects and can suggest fixes. Sketch [90], Rosette [98], Bayou [66], Trinity [61], HPAT [99], and many others have explored different aspects of data-driven synthesis.

The autocomplete features in modern development environments are starting to incorporate advanced machine learning technology. Microsoft, for example, has built complex programming-language models for use with its technologies, and both Codota and Tabnine provide intelligent completion for several programming languages. Commercial tools have started to offer more advanced programming assistance, such as suggest-

ing code refactorings to increase the productivity of maintenance tasks. Expanding on these techniques, researchers have demonstrated unsupervised translation between different programming languages [50]. In addition, considerable effort has been put into developing techniques for program repair. In the future, these kinds of technologies can be made available for scientific programmers as well, with models customized for relevant programming languages and libraries.

> Integrating sophisticated example- and type-driven program synthesis and automated program-repair techniques into modern development environments remains an ongoing challenge.

IDE Integration. Integrating sophisticated example- and type-driven program synthesis and automated program-repair techniques into modern development environments remains an ongoing challenge. The Mnemosyne project[1] provides a framework for the integration of program synthesis techniques for code generation (e.g., Trinity [61]), as well as type inference [75, 103] and test generation.[2] In this system independent synthesis *modules* communicate with each other and with a user's IDE using Microsoft's Language Server Protocol.[3] Programmatic communication between modules enables workflows in which multiple modules collaborate in multiphase synthesis processes. For example, the results of an automated test generation process may trigger and serve as input to subsequent automated code synthesis or program repair processes.

Another significant challenge is that program synthesis is often requested when the program is *incomplete*, thst is, when there are missing pieces or errors that the programmer hopes the synthesizer can help with. In other situations, however, standard techniques for parsing, typechecking, and evaluation fail. Some or all of these may be necessary for the synthesizer to proceed. Recent work on formal reasoning about incomplete

programs by using *holes* has started to address this issue [68, 70], and the Hazel programming environment is being designed specifically around this hole-driven development methodology [69]. Programs with holes are also known as program *sketches* in the program synthesis community [89]. Recent work on type-and-example directed program sketching that takes advantage of these modern advances in reasoning about incomplete programs represents a promising future direction for human-in-the-loop program synthesis [60].

Reversible Computation. The reversible computation paradigm extends the traditional forward-only mode of computation with the ability to compute deterministically in both directions, forward and backward. It allows the program to reverse the effects of the forward execution and go backwards to a previous execution state. *Reverse execution* is based on the idea that for many programs there exists an inverse program that can uncompute all results of the (forward) computed program. The inverse program can be obtained automatically either by generating reverse code from a given forward code or by implementing the program in a reversible programming language; and its compiler offers the capability to automatically generate both the forward and the inverse program. Alternatively, an interpreter for a reversible language can execute a program in both directions, forward and backward. For example, the reverse C compiler presented in [74] generates reverse C code for a given C (forward) code. The imperative reversible language Janus [109] allows both the interpretation of a Janus program, where every language construct has standard and inverse semantics, and the generation of forward and reverse C code for a given Janus program.[4]

Over the years, a number of theoretical aspects of reversible computing have been studied, dealing with categorical foundations of reversibility, foundations of programming languages, and term rewriting, considering various models of sequential computations (automata, Turing machines) as well as concurrent computations (cellular automata, process calculi, Petri nets, and membrane

---

[1]https://grammatech.gitlab.io/Mnemosyne/docs/

[2]https://hypothesis.readthedocs.io/en/latest/

[3]https://microsoft.github.io/language-server-protocol/

[4]Online Janus interpreter at https://topps.diku.dk/pirc/?id=janus\protect\leavevmode@ifvmode\kern+.1667em\relax.

computing). An overview of the state of the art and use cases can be found in [100], titled "Reversible Computation – Extending Horizons of Computing." Reversible computation has attracted interest for multiple applications, covering areas as different as low-power computing [53], high-performance computing with optimistic parallel discrete event simulation [85, 23, 86], robotics [55], and reversible debugging [88]. In [87], the generation of forward and reverse C++ code from Janus code, as well as automatically generated code based on incremental state saving, is systematically evaluated. The example discussed in detail is a reversible variant of matrix multiplication and its use in a benchmark for optimistic parallel discrete event simulation.

Automated Machine Learning. Typically, the ML pipeline comprises several components: pre-processing, data balancing, feature engineering, model development, hyperparameter search, and model selection. Each of these components can have multiple algorithmic choices, and each of these algorithms can have different hyperparameter configurations. Configuring the whole pipeline is beyond human experts, who therefore often resort to trial-and-error methods, which are nonrobust and computationally expensive. Automated machine learning (AutoML) [44] is a technique for automating the design and development of an ML pipeline. Several approaches developed for AutoML can be leveraged for program synthesis and autotuning. A related field is programming by optimization [42], where the algorithm designer develops templates and an optimization algorithm is used to compose these templates to find the right algorithm to solve a given problem. These methods have been used to develop stochastic local search methods for solving difficult combinatorial optimization problems.

Expanding on Existing Work

Existing work on program synthesis, while continuing to evolve to address programming challenges in a variety of domains, can be expanded to specifically address the needs of the scientific programming community.

- **Semantic Code Equivalence/Similarity:** The domain of semantic code similarity [73] aims to identify whether two or more code snippets (e.g., functions, classes, whole programs) are attempting to achieve the same goal, even if the way they go about achieving that goal (i.e., the implementation) is largely divergent. We believe this is one of the most critical areas of advancement in the space of machine programming. The reason is that once semantic code similarity systems demonstrate a reliable level of efficacy, they will likely become the backbone as well as enabling a deeper exploration of many auxiliary MP systems (e.g., automatic bug defection, automatic optimizations, repair) [18, 59, 108, 102, 26, 73].

> Program synthesis might address challenges inherent in targeting heterogeneous hardware architectures and generating performance-portable code.

- **Hardware Heterogeneity:** Program synthesis might address challenges inherent in targeting heterogeneous hardware architectures, in cases where specialized code is needed (perhaps "superoptimizations"), where specialized algorithms are needed, and where specialized "scheduling" functions are needed to dynamically direct work and data to the most appropriate hardware. Different data structures, in addition to different loop structures, may be required to achieve acceptable performance on different kinds of hardware.

- **Performance Portability:** Program synthesis might address the challenge of generating code that performs well across a wide variety of relevant hardware architectures [82]. This synthesis procedure might account for later abilities to autotune the implementation for different target architectures.

- **Data Representation Synthesis:** Optimizing data movement for performance portability demands the ability to synthesize the representation of data to take advantage of

hardware features and input data characteristics such as sparsity. Data representation synthesis includes data layout considerations and potentially should be coordinated with storage mappings that specify storage reuse. Complex interactions among algorithms, memory subsystems, available parallelism, and data motivate delaying data structure selection until adequate information is available to make beneficial decisions. Existing data structure synthesis research [40, 58, 107] has focused on more general relational or map-based data structures and must be extended to scientific computing domains.

- **Numerical Properties:** Scientific programs are often characterized by numerical requirements for accuracy and sensitivity. Synthesis techniques might address the challenges in finding concrete implementations of mathematical algorithms meeting these requirements. This problem is made more difficult because these requirements are often not explicitly specified. Instead, requirements might be only indirectly specified by the need for some postprocessing step to meet its requirements. In addition, the properties being extracted from the output of the algorithm might be fundamentally statistical in nature (e.g., a two-point correlation function), making verification of the program properties itself a fundamentally statistical process.

- **Workflows:** Program synthesis might address the challenges in correctly and efficiently composing different analysis and simulation procedures into an overall scientific workflow. This work might include the generation of specialized interfaces to enable coupling otherwise-modular components with low overhead. It might also include helping use existing APIs in order to combine existing components to accomplish new tasks.

- **Translation:** The existing code base of scientific software contains large amounts of code in C, C++, Fortran, Python, and other languages. Much of this code lacks high-level descriptions; moreover, reusing codes together that are written in different programming languages is often difficult. Worse, even within

the same language, different parallel or concurrent programming models do not compose, for example, Kokkos vs. OpenMP vs. SYCL in C++, OpenMP vs. OpenACC in C/-Fortran [52], Dask vs. Ray vs. Parsl [10] in Python. Program synthesis and semantic representations, such as Iyer et al.'s program-derived semantics graph [45], can help by performing lifting and translation of existing code between different languages and representations and by providing better documentation.

Program synthesis can act as an automated facility; but given the often ambiguous nature of expressed programming goals, synthesis tools are expected to interact with programmers in a more iterative manner. A program synthesis tool can act as an intelligent advisor, offering feedback in addition to some automated assistance. Synthesis tools can prompt programmers for additional information. However, how to best interact with scientists regarding different kinds of scientific programming tasks is an open question in need of further exploration.

> An important relationship exists between program synthesis and explainable AI.

An important relationship exists between program synthesis and explainable AI. State-of-the-art program synthesis techniques depend on deep learning and other data-driven approaches. As a result, the extent to which the program synthesis process itself is explainable depends on explainable AI, including machine learning techniques. Program synthesis itself, on the other hand, can distill a data-driven learning and exploration process into a set of symbolic, understandable rules. These rules can form an explainable result from the machine learning process and thus serve as a technique for producing explainable AI processes.

> Tools useful for scientific programming may need to leverage transfer learning techniques in order to apply knowledge from larger programming domains to scientific programming.

What are the challenges in applying program synthesis to scientific computing and HPC problems?

- **Small Sample Sizes:** Scientific programming comprises a small part of the overall programming market. This situation implies that tools useful for scientific programming, regardless of how the tool development is funded, may need to leverage transfer learning techniques in order to apply knowledge from larger programming domains to scientific programming. The challenges associated with the small sample sizes of scientific programs are exacerbated by the fact that scientific programs are effectively optimizing a variety of different objectives (e.g., performance, portability, numerical accuracy), and how the relevant trade-offs were being made by the application developers often is unknown. The challenge of small sample size applies not only to the code itself but also to the set of programmers; and since many synthesis tools are interactive, training data on how humans most effectively interact with the tools is required but challenging to obtain.

- **Lack of Relevant Requirements Languages:** There does not currently exist, either as an official or a de facto standard, an input language capable of expressing the requirements of a scientific application. While work has been done on expressing correctness conditions in general-purpose software, this is often focused on correctness. Covering the performance requirements, numerical requirements, and so on needed for scientific applications has not been captured in a requirements language. Work also has been done on extracting requirements from natural language, unit tests, and other informal artifacts associated with existing code, and these techniques should be applied to scientific applications as well; but in cases where a user might wish to

directly supply requirements, no broadly applicable way to do so currently exists. Significant domain knowledge is involved in constructing scientific applications, but this knowledge is often not explicitly stated, such as how boundary conditions should be handled and how accurate the results need to be. This lack also is manifest in challenges getting different synthesis tools to interoperate effectively.

- **Verification and Validation:** Verifying that a scientific program satisfies its requirements is a complicated process, and validating the application is likewise a complicated but essential part of the scientific method. The lack of a suitable requirements language, the ambiguities inherent in natural language specifications, and other implicit parts of the code requirements make even defining what "correct" means a challenge. For many configurations of interest, no analytic solution exists to which one can compare with certainty. Moreover, the use of randomized and nondeterministic algorithms, limited numerical stability, and the use of low-precision and approximate computing make even the process of comparing with a reference solution difficult. Scale also makes verification difficult: the computational resources required to test an application might not be readily or regularly acquired. Verification can take multiple forms, often all of which are important:

  - A mathematical verification (proof) of correctness

  - The results of randomized testing showing no problems

  - A human-understandable explanation of what the code does and why it is correct

  Verification often assumes that the base system functions correctly; but especially on leading-edge hardware, defects in the hardware and system software can be observed, and narrowing down problems actually caused by system defects is an important goal. How to measure comprehensibility, succinctness, and naturalness and otherwise ensure that code can be, and is, explained is an

open question. For applications that simulate physical processes, the process of validating that the result matches reality sufficiently well can involve expensive and sometimes difficult-to-automate physical experiments.

- **Legacy Code:** Existing scientific code bases have large amounts of code in Fortran and C, generally considered legacy programming languages. Because of a lack of robust, modular components for processing code in these legacy languages and because of the smaller number of samples for training in these legacy languages, it can be challenging to apply program synthesis tools to code that must interact with these code bases. Moreover, even if modern languages such as C++ or Python are used, the libraries used for scientific development are often distinct from the libraries used more generally across domains. Some of these libraries use legacy interface styles (e.g., BLAS) that present some of the same challenges as do legacy programming languages.

- **Integration and Maintenance:** The use of tools that generate source code has long been problematic from an integration perspective, especially if the code generation process is nondeterministic or depends on hardware-specific measurements or on human interaction. With all such tools, difficult questions must be addressed by the development team:

  - Should the tool be run as part of the build process?

  - If run as part of the build process, builds become nondeterministic and perhaps slower, both because of the time consumed by the tool and because the requirements associated with making sufficiently reliable performance measurements may limit build parallelism.

  - If not run as part of the build process, how is the output cached? Is it part of the source repository? How is the cache kept synchronized with the tool's input? Do all developers have access to all of the relevant hardware to update the cached output? Alternatively, can the tool generate performance-portable code? If the

tool is interactive, how are invocation-to-invocation changes minimized?

> Synthesis tools face challenges, not only in interfacing with existing code, but also in interfacing with each other.

- **Composability:** As with other software components and tools, synthesis tools face challenges, not only in interfacing with existing code, but also in interfacing with each other. The properties of code generated by one tool may need to be understood by another synthesis tool, and the tools may need to iterate together in order to find an overall solution to the programming challenge at hand [5]. The Sparse Polyhedral Framework [95] provides a possible foundation for composing data and schedule transformations with compile-time and runtime components in the context of program synthesis.

- **Stability and Support:** Like other parts of the development environment, program synthesis tools require a plan for stability and support in order to mitigate the risk that defects or missing features in the tools do not block future science work. In order to transition from research prototypes to tools useful by the larger scientific community, the tools must be built on robust, well-maintained infrastructures and must be robust and well maintained themselves.

- **Search Space Modeling for Program Synthesis:** The search space of program synthesis for scientific workloads is complex, rendering many search algorithms ineffective. For example, in autotuning, the search can be formulated as a noisy, mixed-integer, nonlinear mathematical optimization problem over a decision space comprising continuous, discrete, and categorical parameters. For LLVM loop optimization, as an example, we are faced with a dynamic search space that changes based on the decisions at the previous step. Modeling the search space of these

problems will significantly reduce the complexity of the problem and will allow us to develop effective problem-specific search methods. Application- and architecture-specific knowledge should be incorporated as models and constraints. Moreover, by considering metrics such as power and energy as objectives rather than constraints, we can obtain a hierarchy of solutions and quantify the sensitivities associated with changing constraint bounds. These can significantly simplify (or even trivialize) online optimization at runtime, when quantities such as resource availability and system state or health are known.

What are aggressive short-term, medium-term, and long-term opportunities and goals?

A number of opportunities exist to apply program synthesis techniques to scientific-computing problems, and these opportunities will expand with time.

3.5.1 Short Term (1–2 Years)

- **Defining Challenge and Benchmark Problems:** The community working to develop new program synthesis technologies can use challenge problems to direct their long-term aims and enable conversations with the scientific programming community. In order to measure progress toward addressing those challenge problems, establish concrete examples, and enable comparison between systems, collections of benchmark problems should also be developed. Separate challenge and benchmark collections can be developed for different classes of problems (e.g., programming-language translation, specification-driven synthesis, and example-driven synthesis).

- **Interactive Synthesis, Repair, and Debugging:** Constrained techniques, such as proposing simple hole fillings [60, 8] and local source-code edits to correct user-identified mismatches between expected and observed application behaviors, can be intcorporated into integrated development environment

software that is being used to develop scientific software.

- **Smarter Code Templates:** Synthesis tools can provide assistance with generating boilerplate code, applying design patterns, and performing other largely repetitive programming tasks. Some of these patterns occur more frequently in scientific computing than in other domains, such as halo exchange in physical simulations.

- **Numerical Precision, Accuracy, and Stability:** Synthesis tools can start addressing situations in which different options exist between algorithmic variants and the precision used to represent numerical quantities. Automatic selection between these options, based on user-supplied metrics, including performance, accuracy, and stability, should be possible. A fundamental requirement is to guard synthesis via rigorous and scalable roundoff error estimation methods [27].

- **Superoptimization and High-Quality Compilation:** Synthesis systems can use exhaustive search techniques effectively in restricted domains to find the best algorithmic compositions and code generation for particular systems. Superoptimization in restricted domains can apply to library call sequences and other high-level interface generation tasks. Compilation improvements will include more helpful and more accurate user feedback regarding where and what code annotations will be helpful.

- **Knowledge Extraction:** Analysis and verification tools will extract ever more precise and relevant high-level specifications from existing implementations, often referred to as lifting, and use these specifications to enable verification, interface generation, and other tasks [45].

- **Intelligent Searching:** Intelligent code search engines can be constructed that account for code structure, naming, comments, associated documents (e.g., academic papers), and other metadata. Finding code examples and other existing solutions to related prob-

lems can significantly increase programmer productivity.

- **Synthesis via Component Assembly:** The large-scale application frameworks in the past two decades have used componentization and assembly for a modest level of high-level program construction. The key idea is to have an infrastructural framework that becomes a backbone that model and algorithmic components of the solution system can plug into as needed. While this methodology by itself would not suffice for the multilevel parallelism that we are facing now, the concept of assembly from components can still provide an easy way of attacking some of the performance portability challenges by incorporating variable granularities in how componentization occurs. Past frameworks assumed components at the level of separate standalone capabilities in a quest to shield the science and core numerical algorithms from the details of infrastructure and assembly. Allowing componentization within such standalone capabilities (in a sense letting some of the infrastructural aspects intrude into the science sections of the code) can be helpful in reducing replicated code for different platforms. For example, one can view any function as having a declaration block, blocks of arithmetic expressions, blocks of logic and control, and blocks of explicit data movements that can each become a component. A subset of these components can have multiple implementations if needed or, better still, be synthesized from higher-level expressions. A tool that does not care whether the implementations exist as alternatives or are synthesized will be relatively simple and general purpose and can have a huge impact on productivity and maintainability, while at the same time reducing the complexity burden on other synthesis tools in the toolchain.

- **Benchmarks:** Developing a set of easy-to-use benchmarks and well-defined metrics for comparison will be critical for advancing the algorithms for program generation. These benchmarks should reduce or hide the overhead required to let researchers from other areas to develop algorithms. For example,

autotuning can benefit greatly from applied math and optimization researchers, but currently no easy-to-use framework allows these researchers to test algorithms.

3.5.2 Medium Term (3–5 Years)

- **Synthesis of Test Cases:** Based on user-specified priorities, background domain knowledge, and source analysis, synthesis tools can generate test cases for an application and its subcomponents. These span the granularity space from unit tests through whole-application integration tests and, moreover, can include various kinds of randomized testing (e.g., "fuzz testing") and the generation of specific, fixed tests.

- **Parallelization and Programming Models:** Synthesis tools can assist with converting serial code to parallel code and converting between different parallel-programming models. Parallelism exists at multiple levels, including distributed-memory parallelism (e.g., expressed using MPI) and intranode parallelism (e.g., expressed using OpenMP, SYCL, or Kokkos).

- **Optimized Coupling Code:** Complex applications often require different subcomponents to communicate efficiently, and the "glue code" needed between different components is often tedious to write. Synthesis tools can create this kind of code automatically and over the medium term can create customized bindings that limit unnecessary copying and format conversions. Over the longer term, these customized data structure choices can permeate and be optimized over the entire application.

- **Performance Portability:** Generating code that works well on a particular target architecture is challenging, but generating code that performs well across many different target architectures adds an additional layer of complexity. Synthesis should be able to address this combined challenge of generating performance-portable code (using, e.g., OpenMP, SYCL, or Kokkos) that has good

performance across a wide array of different platforms.

- **Autotuning:** Autotuning is becoming a proven technology to achieve high performance and performance portability. Despite several promising results, however, a number of challenges remain that we need to overcome for a wider adoption [13]. Autotuning should be made seamless and easy to use from the application developer's perspective. This task involves a wide range of advancements from automated kernel extraction and large-scale search algorithms, to reducing the computationally expensive nature of the autotuning process. Modeling objectives such as runtime, power, and energy as functions of application and platform characteristics will play a central role. These models will be used to quantify meaningful differences across the decision space and to offer a convenient mechanism for exposing near-optimal spots in the decision space.

3.5.3 Long Term

- **Solving Challenge Problems:** Challenge problems should be solvable by using composable, widely available tools. These tools should be capable of incorporating background knowledge from a wide variety of domains and of producing efficient, verifiable solutions in a reasonable amount of time. Where appropriate, the code will use sensible identifier names and otherwise be readable and maintainable.

- **Intentional Programming:** Synthesis tools can operate using high-level mathematical and natural language specifications, largely automatic but eliciting key feedback from human scientists, working within a common framework that supports the tooling ecosystem.

- **Lifelong Learning:** Synthesis tools will use an iterative refinement scheme, learning from user feedback and automatically improving themselves as time goes on. Synthesis tools will be able to create new abstractions for different domains and evolve them over time. Recent developments in reinforcement learning can be an effective vessel for lifelong learning.

- **Understanding Legacy Code:** Lifting, or the extraction of high-level features from concrete implementations, can work over large bodies of code. Synthesis tools will use lifting processes to understand legacy code, including any necessary use of background knowledge, and can interface with it or translate it to other forms.

- **Full-Application Verification:** Full applications, including library dependencies, can be verified, symbolically, statistically, or otherwise, with high confidence. The verification procedures can be driven automatically based on user-provided priorities and intelligent source-code analysis.

- **Proxy Generation:** Proxy applications, representing user-specified aspects of a full application, can be automatically generated by synthesis tools. These proxies will include appropriate source-code comments and documentation.

- **End-to-End Automation for Autotuning:** Autotuning needs to be part of the compiler tool chain. The process of autotuning should not involve any manual intervention. We need to develop autonomous, intelligent autotuning technology that can surpass human capability to accelerate computational science on extreme-scale computing platforms.

COMPILER TECHNOLOGY

Synthesis technology and compiler technology are intimately tied. In the end, compilers are an essential consumer of the output of synthesis tools. Moreover, synthesis tools often reuse programming language processing and analysis infrastructure from compiler infrastructures in order to process inputs and mine existing code bases for patterns and other background knowledge. Improvements in synthesis technology are expected to

go hand in hand with improvements in compiler technology.

> Improvements in synthesis technology are expected to go hand in hand with improvements in compiler technology.

What opportunities exist for improving compilers and program analysis tools to better support scientific applications and HPC?

Compilers can better leverage machine learning techniques to drive heuristics and other algorithmic tuning parameters. Examples include the following:

- Optimizing the order in which transformations are applied

- Optimizing the thresholds and other parameters used by transformation and analysis routines, including choices between algorithms, to balance cost vs. benefit tradeoffs.

- Optimizing the sets of features used to drive the aforementioned tuning decisions.

- Building cheap surrogate models for performance modeling in autotuning, which can be used to prune the large search space and identify promising regions in a short time.

Compilers and other analysis tools can produce more information, both about the code being compiled and about the tool's modeling results and decisions. This information can be used by human engineers and by synthesis tools as part of an iterative development process.

As hardware architectures become more complex, with multiple levels in memory and storage hierarchies and different computational accelerators, compilers can offer additional support for mapping applications into the underlying hardware. This may require more information regarding data sizes, task dependencies, and so on than what is traditionally provided to a compiler for a programming language such as C++ or Fortran. Data sizes, for example, affect how code is optimized:

the optimal code structure for data that fits into the L1 cache may be different from that for data that fits into the L2 cache. These code structure differences include loop structures, data structures and layouts, and accelerator targeting.

How might higher-level information be leveraged to capture those opportunities?

A significant challenge in implementing compilers for Fortran, C++, and other high-performance languages used in scientific computing is the lack of higher-level information in the source code. The compiler generally does not know the size of data arrays, the likely location of data within the memory subsystem, and the number of threads being used. This lack of information forces compiler developers to fall back on heuristics that are likely to work across a larger number of use cases instead of creating models that can optimize specific use cases.

When DSLs are used, some of this information is contained in the input, and where it is not, the semantics of the DSL can constrain the number of specialized versions that the compiler might generate to a practical number. Similarly, known library interfaces can effectively form a DSL and have similar properties that can be extracted by an intelligent compiler to leverage in driving the compilation process. Separate compilation, the general case where an application's source code is split across multiple separately compiled source files, is helpful in enabling parallelism in the build process and other separations of concerns but limits the higher-level information that can be usefully extracted. Tools that build application-scoped databases from source-analysis processes can potentially mitigate that loss of information. These tools, along with associated source annotations and known library interfaces, can enable high-level compiler optimizations while remaining transparent to the programmer.

How might compiler technology be improved to better integrate with program synthesis systems?

Synthesis systems are generally iterative, combining techniques to search the space of potential solutions with techniques for evaluating particular candidate solutions. Compiler technology

can be enhanced for better participation in both parts of this process. Information extracted from code compilation or attempted compilation of partial solutions can be used to constrain the search. In addition, information from compilation can be used to evaluate potential solutions in terms of both correctness and performance; however, performance models from the compiler can be used to inform a wider set of metrics.

Recent advances in compiler frameworks, such as LLVM and its multilevel IR (MLIR), can be leveraged to extract suitable information from applications. Examples of features that can be easily extracted are properties of loop-based computations that exhibit regularity, such as loop depth, array access patterns, shape and size of multidimensional arrays, and dependence structures (e.g., dependence polyhedra in affine programs). As these features evolve and change within and among intermediate representations of a compiler, metadata must be collected describing the sequence of transformations applied. Similarly, for more irregular application such as those in arising in sparse linear systems, the sparsity pattern and structure can be used as synthesis features and can be either collected during the system assembly process or provided by the end user via compiler directives.

Integrating with an iterative synthesis process may also require new schemes to be implemented in compilers that allow for the caching of partial compilation and analysis results so that each variant evaluated during the synthesis process does not incur the full overhead of performing the necessary analysis and transformations each time.

> When both a synthesis tool and the underlying compiler are responsible for code optimization, it is an open question how this responsibility is best divided and how the cooperation will be arranged.

When both a synthesis tool and the underlying compiler are responsible for code optimization, it is an open question how this responsibility is best divided and how the cooperation will be arranged. Synthesis systems might be responsible for higher-level transformations that are difficult to prove

correct in the absence of higher-level information, for example, or for the use of hand-tuned code fragments that a machine-programming system is unlikely to discover automatically. As work continues, open interfaces that are adopted by multiple compilers and synthesis tools will likely enable a diverse, vibrant ecosystem of programming-environment capabilities.

REFERENCES

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging MapReduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.

[2] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to Halide. *ACM Transactions on Graphics (TOG)*, 38(6):1–13, 2019.

[3] Mejbah Alam, Justin Gottschlich, Nesime Tatbul, Javier S Turek, Tim Mattson, and Abdullah Muzahid. A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlchBuc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, NeurIPS 2019, pages 11623–11635. Curran Associates, Inc., 2019.

[4] Mohammad Mejbah ul Alam and Abdullah Muzahid. Production-run software failure diagnosis via adaptive communication tracking. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 354–366, 2016. ISBN 9781467389471. doi: 10.1109/ISCA.2016.39.

[5] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. *SIGARCH Comput. Archit. News*, 44(1):241–252, March 2009. ISSN 0163-5964. doi: 10.1145/2528521.1508273. URL `https://doi.org/10.1145/2528521.1508273`.

[6] Farhana Aleen, Monirul Sharif, and Santosh Pande. Input-driven dynamic execution prediction of streaming applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 315–324, 2010. URL `https://doi.org/10.1145/1693453.1693494`.

[7] Farhana Aleen, Vyacheslav P. Zakharin, Rakesh Krishaniyer, Garima Gupta, David Kreitzer, and Chang-Sun Lin. Automated compiler optimization of multiple vector loads/stores. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 82—91, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341288. doi:

10.1145/2903150.2903169. URL https://doi.org/10.1145/2903150.2903169.

[8] S. An, R. Singh, S. Misailovic, and R. Samanta. Augmented Example-Based Synthesis using Relational Perturbation Properties. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(POPL):56:1–56:24, 2020. doi: 10.1145/3371124.

[9] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. *ACM Sigplan Notices*, 44(6):38–49, 2009.

[10] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M Wozniak, Ian Foster, et al. Parsl: Pervasive parallel programming in Python. In *28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 25–36, 2019.

[11] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018. doi: 10.1109/JPROC.2018.2841200.

[12] Prasanna Balaprakash, Romain Egele, and Paul Hovland. ytopt. GitHub repository. URL https://github.com/ytopt-team/ytopt.

[13] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106 (11):2068–2083, 2018.

[14] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. Automatic generation of efficient linear algebra programs. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, 2020.

[15] Ayon Basumallik and Rudolf Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, New York, NY, USA, 2006. ACM Press.

[16] Martin Bauer, Harald Köstler, and Ulrich Rüde. lbmpy: A flexible code generation toolkit for highly efficient lattice boltzmann simulations. *arXiv preprint arXiv:2001.11806*, 2020.

[17] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. AutoPandas: Neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[18] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3585–3597. Curran Associates, Inc., 2018.

[19] Ian Briggs, Arnab Das, Mark Baranowski, Vishal Chandra Sharma, Sriram Krishnamoorthy, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. FailAmp: Relativization transformation for soft error detection in structured address generation. *ACM Trans. Archit. Code Optim.*, 16(4):50:1–50:21, 2020. doi: 10.1145/3369381. URL https://doi.org/10.1145/3369381.

[20] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 83–96, 2020.

[21] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013.

[22] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *International Parallel Distributed Processing Symposium (IPDPS'11)*, pages 676–687. IEEE, 2011. doi: 10.1109/IPDPS.2011.70.

[23] Davide Cingolani, Alessandro Pellegrini, Markus Schordan, Francesco Quaglia, and David R. Jefferson. Dealing with reversibility of shared libraries in PDES. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, page 41–52, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344890. doi: 10.1145/3064911.3064927. URL https://doi.org/10.1145/3064911.3064927.

[24] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E Osuna, Robert Pincus, Jon Rood, and William Sawyer. The CLAW DSL: Abstractions for performance portable weather and climate models. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–10, 2018.

[25] Bruce Collie. Program synthesis for heterogenous accelerators. URL https://baltoli.github.io/static/msc.pdf.

[26] L. D'Antoni, R. Samanta, and R. Singh. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification (CAV)*, pages 383–401, 2016. doi: 10.1007/978-3-319-41540-6_21.

[27] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. Scalable yet rigorous floating-point error analysis. In *Proceedings*

of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[28] Arnab Das, Sriram Krishnamoorthy, Ian Briggs, Ganesh Gopalakrishnan, and Ramakrishna Tipireddy. FPDetect: Efficient reasoning about stencil programs using selective direct evaluation. *ACM Trans. Archit. Code Optim.*, 17(3):19:1–19:27, 2020. URL `https://dl.acm.org/doi/10.1145/3402451`.

[29] Eddie C Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 265–277, 2018.

[30] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[31] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–241, New York, NY, USA, May 1999. ACM.

[32] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74 (12):3202–3216, 2014.

[33] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. SPIRAL: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018.

[34] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pages 1381–1384. IEEE, 1998.

[35] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B. Tenenbaum, and Tim Mattson. The Three Pillars of Machine Programming. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211355. URL `http://doi.acm.org/10.1145/3211346.3211355`.

[36] S. Gulwani, O. Polozov, and R. Singh. *Program Synthesis*. Foundations and Trends(r) in Programming Languages Series. Now Publishers, 2017. ISBN 9781680832921. URL `https://books.google.com/books?id=mK5ctAEACAAJ`.

[37] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[38] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006.

[39] Niranjan Hasabnis and Justin Gottschlich. ControlFlag: A self-supervised idiosyncratic pattern detection system for software control structures. In *34th Conference on Neural Information Processing Systems (NeurIPS 2020), ML for Systems Workshop*. 2020.

[40] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proceedings of Programming Languages Design and Implementation (PLDI)*, 2011.

[41] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107 (46):9887–9897, 2003.

[42] Holger H Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[43] Sascha Husa, Ian Hinder, and Christiane Lechner. Kranc: A Mathematica package to generate numerical codes for tensorial evolution equations. *Computer Physics Communications*, 174(12):983–1004, 2006.

[44] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

[45] Roshni G. Iyer, Yizhou Sun, Wei Wang, and Justin Gottschlich. Software language comprehension using a program-derived semantics graph. In *34th Conference on Neural Information Processing Systems (NeurIPS 2020), Computer-Assisted Programming Workshop*. 2020.

[46] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 711–726.

[47] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[48] Martin Kong and Louis-Noël Pouchet. Model-driven transformations for multi- and many-core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI

2019, page 469–484, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314653. URL https://doi.org/10.1145/3314221.3314653.

[49] Michael Kruse, Xingfu Wu, and Hal Finkel. Autotuning Search Space for Loop Transformations, September 2020.

[50] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*, 2020.

[51] Jacob Lambert, Seyong Lee, Jungwon Kim, Jeffrey S. Vetter, and Allen D. Malony. Directive-based, high-level programming and optimizations for high-performance computing with FPGAs. In *ACM International Conference on Supercomputing (ICS18)*, June 2018.

[52] Jacob Lambert, Seyong Lee, Jeffrey Vetter, and Allen Malony. CCAMP: An integrated translation and optimization framework for OpenACC and OpenMP. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1387–1400, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society. doi: 10.1109/SC41405.2020.00102. URL https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00102.

[53] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[54] Michael Lange, Navjot Kukreja, Mathias Louboutin, Fabio Luporini, Felippe Vieira, Vincenzo Pandolfo, Paulius Velesko, Paulius Kazakas, and Gerard Gorman. Devito: Towards a generic finite difference DSL using symbolic python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 67–75. IEEE, 2016.

[55] J. S. Laursen, U. P. Schultz, and L. Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1785–1792, 2015.

[56] Seyong Lee, Dong Li, and Jeffrey S. Vetter. Interactive program debugging and optimization for directive-based GPU computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014.

[57] Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. FFC: The FEniCS form compiler. In *Automated Solution of Differential Equations by the Finite Element Method*, pages 227–238. Springer, 2012.

[58] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 958–968, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3 (OOPSLA), October 2019. doi: 10.1145/3360578. URL https://doi.org/10.1145/3360578.

[60] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020. doi: 10.1145/3408991. URL https://doi.org/10.1145/3408991.

[61] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. Trinity: An extensible synthesis framework for data science. *Proceedings of the VLDB Endowment*, 12 (12):1914–1917, 2019. Publisher: VLDB Endowment.

[62] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[63] Reed Milewicz, Gustavo Pinto, and Paige Rodeghero. Characterizing the roles of contributors in open-source scientific software projects. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 421–432. IEEE, 2019.

[64] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 140–152, 1988.

[65] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–202, Los Alamitos, CA, USA, October 1999. IEEE Computer Society.

[66] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

[67] Catherine Olschanowsky, Stephen Guzik, John Loffeld, Jeffrey Hittinger, and Michelle M. Strout. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 793–804. IEEE Press, 2014.

[68] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: a bidirectionally typed structure editor calculus. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 86–99. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009900.

[69] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward semantic foundations for program editors. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 11:1–11:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.SNAPL.2017.11. URL `https://doi.org/10.4230/LIPIcs.SNAPL.2017.11`.

[70] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proc. ACM Program. Lang.*, 3(POPL):14:1–14:32, 2019. doi: 10.1145/3290327. URL `https://doi.org/10.1145/3290327`.

[71] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015. doi: 10.1145/2737924.2738007. URL `https://doi.org/10.1145/2737924.2738007`.

[72] Carlos Osuna. Report on the performance portability demonstrated for the relevant weather & climate dwarfs. *arXiv preprint arXiv:1908.06094*, 2019.

[73] D. M. Perry, D. Kim, R. Samanta, and X. Zhang. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Programming Language Design and Implementation (PLDI)*, pages 860–873, 2019. doi: 10.1145/3314221.3314629.

[74] Kalyan S. Perumalla. *Introduction to Reversible Computing*. CRC Press Book, 2013.

[75] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. *arXiv preprint arXiv:1912.03768*, 2019.

[76] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[77] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530, 2013. doi: 10.1145/2462156.2462176. URL `http://doi.acm.org/10.1145/2462156.2462176`.

[78] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):1–27, 2016.

[79] Lawrence Rauchwerger, Nancy M. Amato, and David A. Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 23(6):537–576, 1995.

[80] Mahesh Ravishankar, John Eisenlohr, Louis-Noel Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, November 2012.

[81] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 65–75, New York, NY, USA, 2015. ACM.

[82] Amit Sabne, Putt Sakdhnagool, Seyong Lee, and Jeffrey S. Vetter. Understanding portability of a high-level programming model on contemporary heterogeneous architectures. In *IEEE Micro*, volume 35. IEEE, July 2015. doi: http://dx.doi.org/10.1109/MM.2015.73.

[83] Joel Saltz, Chialin Chang, Guy Edjlali, Yuan-Shin Hwang, Bongki Moon, Ravi Ponnusamy, Shamik Sharma, Alan Sussman, Mustafa Uysal, Gagan Agrawal, Raja Das, and Paul Havlak. Programming irregular applications: Runtime support, compilation and tools. *Advances in Computers*, 45:105–153, 1997.

[84] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[85] Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, and Daniel Quinlan. Reverse code generation for parallel discrete event simulation. In Jean Krivine and Jean-Bernard Stefani, editors, *Reversible Computation*, pages 95–110, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20860-2.

[86] Markus Schordan, Tomas Oppelstrup, David R. Jefferson, and Peter D. Barnes Jr. Generation of reversible C++ code for optimistic parallel discrete event simulation. *New Gener. Comput.*, 36(3):257–280, 2018. doi: 10.1007/s00354-018-0038-2. URL `https://doi.org/10.1007/s00354-018-0038-2`.

[87] Markus Schordan, Tomas Oppelstrup, Michael Kirkedal Thomsen, and Robert Glück. Reversible languages and incremental state saving

in optimistic parallel discrete event simulation. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz, and Carla Ferreira, editors, *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405*, pages 187–207. Springer International Publishing, Cham, 2020. ISBN 978-3-030-47361-7. doi: 10.1007/978-3-030-47361-7_9. URL https://doi.org/10.1007/978-3-030-47361-7_9.

[88] Shyh-Kwei Chen, W. K. Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, 2001.

[89] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009. doi: 10.1007/978-3-642-10672-9_3. URL https://doi.org/10.1007/978-3-642-10672-9_3.

[90] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 167–178, 2007.

[91] Daniele G Spampinato and Markus Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 23–32, 2014.

[92] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix multiplication beyond auto-tuning: Rewrite-based GPU code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16. ACM, 2016. doi: 10.1145/2968455.2968521.

[93] Rick Stevens, Valerie Taylor, Jeff Nichols, Arthur Barney Maccabe, Katherine Yelick, and David Brown. AI for science. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2020.

[94] James M. Stichnoth and Thomas Gross. Code composition as an implementation language for compilers. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 119–132, Berkeley, October 15–17 1997. USENIX Association.

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

[96] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. Combining performance aspects of irregular Gauss–Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Berlin / Heidelberg, July 2002. Springer.

[97] Michelle Mills Strout, Fabio Luporini, Christopher D. Krieger, Carlo Bertolli, Gheorghe-Teodor Bercea, Catherine Olschanowsky, J . Ramanujam, and Paul H.J. Kelly. Generalizing run-time tiling with the loop chain abstraction. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014.

[98] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[99] Ehsan Totoni, Todd A Anderson, and Tatiana Shpeisman. HPAT: high performance analytics with scripting ease-of-use. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[100] I. Ulidowski, I. Lanese, U. Schultz, Carla Ferreira, E. Bertino, and Carla Ferreira Eds. Reversible computation: Extending horizons of computing. *LNCS*, 12070, 2020. doi: 10.1007/978-3-030-47361-7.

[101] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity. Technical report, USDOE Office of Science (SC), Washington, DC (United States), 2018.

[102] Yanjun Wang, Jinwei Liu, Dalin Zhang, and Xiaokang Qiu. Reasoning about recursive tree traversals. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. ACM, 2021.

[103] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161*, 2020.

[104] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel computing*, 27(1-2):3–35, 2001.

[105] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 57–68, New York, NY, USA, 2013. ACM.

[106] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[107] Cong Yan and Alvin Cheung. Generating application-specific data layouts for in-memory databases. *Proc. VLDB Endow.*, 12(11):1513–1525, 2019.

[108] Fangke Ye, Shengtian Zhou, Anand Venkat, Ryan Marcus, Nesime Tatbul, Jesmin Jahan Tithi, Niranjan Hasabnis, Paul Petersen, Timothy Mattson, Tim Kraska, Pradeep Dubey, Vivek Sarkar, and Justin Gottschlich. MISIM: A novel code similarity system, 2020.

[109] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*, pages 144–153. ACM, 2007. URL `https://doi.org/10.1145/1244381.1244404`.

[110] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356210. URL `https://doi.org/10.1145/3295500.3356210`.

[111] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. Improving communication by optimizing on-node data movement with data layout. In *Proceedings of the ACM SIGPLAN/SIGHPC Conference on Principles and Practice of Parallel Programming*, PPoPP'21, 2021.