Verified Lifting of Stencil Computations

Shoaib Kamil
Adobe, USA
kamil@adobe.com

Alvin Cheung
University of Washington, USA
akcheung@cs.washington.edu

Shachar Itzhaky
Armando Solar-Lezama
Massachusetts Institute of Technology, USA
shachari,asolar@csail.mit.edu

Abstract

This paper demonstrates a novel combination of program synthesis and verification to lift stencil computations from low-level Fortran code to a high-level summary expressed using a predicate language. The technique is sound and mostly automated, and leverages counter-example guided inductive synthesis (CEGIS) to find provably correct translations. Lifting existing code to a high-performance description language has a number of benefits, including maintainability and performance portability. Our experiments show that the lifted summaries allow domain specific compilers to do a better job of parallelization as compared to an off-the-shelf compiler working on the original code, and can even support fully automatic migration to hardware accelerators such as GPUs. We have implemented verified lifting in a system called STNG and have evaluated it using microbenchmarks, mini-apps, and real-world applications. We demonstrate the benefits of verified lifting by first automatically summarizing Fortran source code into a high-level predicate language, and subsequently translating the lifted summaries into Halide, with the translated code achieving median performance speedups of $4.1\times$ and up to $24\times$ as compared to the original implementation.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Automatic Programming*]: Program Synthesis; I.2.2 [*Automatic Programming*]: Program Transformation

Keywords Verified Lifting, Program Synthesis, Domain-specific Languages

1. Introduction

Until recently, writing high-performance code meant programming in a low-level language like C or Fortran in order to have fine-grain control over all aspects of the execution. The downside of this kind of performance programming was that platform specific optimization tricks would become entwined with the application logic itself, making the resulting code difficult to read and reason about. Furthermore, after the original developers have moved on, maintaining the heavily-tuned code became a nightmare—porting such code to new architectures and runtimes is extremely tedious and is an invitation for introducing new bugs.

In the last few years, high-performance domain specific languages (DSLs) have shown that starting from a high-level program, it is possible to generate implementations that are significantly more efficient than what a state of the art compiler can produce starting from carefully hand-optimized low-level code [1, 13, 47, 55, 56, 64]. High-level domain specific code can be a better starting point for optimization because the compiler is not tied down by low-level implementation decisions embedded in the program, and is able to explore a broad implementation space much more efficiently and systematically than a human programmer.

Stencil computations represent one domain where high-performance DSLs have been especially successful. Stencil computations are nearest-neighbor computations on a multi-dimensional grid; each point in the grid is updated as a function of its value and the value of its neighbors. This kind of computation is commonly found in machine learning, scientific computing and image processing, and quite often is the computational bottleneck of such applications. As a result, a number of high performance domain-specific languages for stencils have been developed to make it possible to obtain good performance with relatively low programmer effort [14, 16, 30, 37, 47, 56], and many DSLs also leverage customized hardware (e.g., GPUs and Many-Integrated Cores accelerators) for additional performance gains. Unfortunately, the performance gains obtained by these high-performance DSLs cannot benefit existing applications written in general purpose languages, so their impact is limited to new code developed in the DSL or to existing code that is manually and ex-

pensively rewritten. For rewriting existing highly-optimized code, the main hurdle is understanding the original programmer intent, since hand-optimization conflates the high-level operations with low-level optimizations such as vectorization and parallelization. Deciphering high-level intent and re-optimizing can take many months of developer effort [43].

This paper demonstrates a technique called *verified lifting* that can address this problem for stencil computations. Verified lifting takes as input a block of potentially optimized code written in an imperative general-purpose language, and infers a summary expressed in a high-level predicate language that is provably equivalent to the semantics of the original program. The lifted summaries expressed using the predicate language are found automatically using inductive program synthesis. Once found, such summaries can be translated to different high-performance DSLs, and subsequently retargeted to execute on different architectures as needed.

Verified lifting can be seen as a form of de-compilation, but is quite different from traditional de-compilation both in terms of scope and in terms of underlying techniques. A traditional de-compiler is not that different from a traditional compiler; its goal is to translate from a low-level general purpose notation (usually assembly language) to another low-level notation like C while applying transformations that improve the readability of the resulting code, for example by introducing structured control flow when possible. In contrast, verified lifting is actually solving an *algorithm identification problem*, where it needs to discover, for example, that a complex sequence of nested loops is actually implementing a simple five point stencil. Recovering the original algorithm behind a hand-optimized piece of low-level code is difficult to do with traditional compiler technology. Instead, verified lifting leverages *inductive synthesis* technology. The system makes hypotheses by generalizing from observed behavior of the code, and then tests the hypotheses by attempting to verify them. By leveraging recent advances in inductive synthesis and verification based on SAT/SMT solving, our system is able to synthesize high-level representations from complex real-world stencils in a matter of minutes in most cases.

The general technique of verified lifting was first proposed by Cheung et al. in the context of database-backed applications [15]. In that context, the goal was to lift imperative code performing data manipulations up to a high-level SQL query that could then be implemented efficiently in the database. Adapting the technique to stencil computations, however, poses a number of new challenges which are addressed in this paper. These challenges arise from the fact that the low-level code we are targeting requires much more complex loop invariants, making the synthesis and verification problems fundamentally more difficult, and requiring new techniques to solve them. For example, the largest stencil computation for which we applied our system required it to automatically infer five loop invariants each with five universally quantified variables and 457 AST nodes.

We have implemented verified lifting for stencil computations in a prototype called STNG, which can automatically identify and find high-level summaries of stencils from general-purpose code written in Fortran. The summaries are expressed using a predicate language based on the theory of arrays [11]. To demonstrate how the lifted summaries can be used, STNG translates them into a high-performance stencil DSL called Halide [47]. Translating the summaries to Halide allows the stencil implementations to be automatically parallelized and even ported to execute on GPUs.

Overall, this paper makes the following contributions:

- We describe verified lifting of stencil computations, a technique that combines recent advances in inductive synthesis and verification to lift code in a low-level language to its equivalent in a high-level predicate language.
- We describe *inductive template generation*, a new technique that uses combined concrete and symbolic execution of the original implementation to guide the search for the high-level summary of the stencil and the invariants necessary to prove it correct.
- We demonstrate the use of Skolemization and partial Skolemization to make the synthesis of universally quantified invariants tractable.
- We present our implementation of verified lifting in STNG, which identifies stencils from Fortran code and lifts them to summaries expressed in a predicate language. To show the benefits of lifting, STNG translates the summaries into Halide, a high-performance DSL. By running the Halide code on microbenchmarks, mini-apps, and real-world applications, our results demonstrate that, combined with autotuning, the translated Halide code can improve performance by up to 17× on non-trivial real-world applications.

2. Overview

The input to the STNG compiler is Fortran code with loop nests that implement stencil kernels[1], and the output is Halide code for each of the stencils, together with a new version of the original Fortran code that has been modified to invoke the generated Halide implementations in place of the original kernels. The process of lifting occurs in two steps: first, we discover a *summary* of the loop nest that captures, in a mathematical formula, the changes to the output arrays after the kernel has executed. We then use this formula to generate code in the backend language (Halide in this case).

To demonstrate STNG, we show how it lifts the simple stencil shown in Figure 1(a). Though this example is simpler than stencils used in real world code, it allows us to show the issues that arise in lifting low-level code to a high-level predicate language.

[1] We use the term *stencil kernel* to denote a single loop nest that writes to one or more multidimensional arrays, computing each output point as a function of other multidimensional input arrays.

```
procedure sten(imin,imax,jmin,jmax,a,b)
  real (kind=8), dimension(imin:imax,jmin:jmax) :: a
  real (kind=8), dimension(imin:imax,jmin:jmax) :: b
  do j=jmin,jmax
    t = b(imin, j)
    do i=imin+1,imax
      q = b(i,j)
      a(i,j) = q + t
      t = q
    enddo
  enddo
end procedure
```

(a)

$post(\texttt{a},\texttt{b}) \equiv \forall\, \texttt{imin+1} \leq \texttt{i} \leq \texttt{imax},\ \texttt{jmin} \leq \texttt{j} \leq \texttt{jmax}.$
$\quad \texttt{a(i,j)} = \texttt{b(i-1,j)} + \texttt{b(i,j)}$

(b)

$invariant(\texttt{a},\texttt{b},\texttt{j}) \equiv \texttt{j} \leq \texttt{jmax} + 1 \wedge$
$\quad \forall\, \texttt{imin+1} \leq \texttt{i} \leq \texttt{imax},\ \texttt{jmin} \leq j' < \texttt{j}.$
$\quad \texttt{a(i,}j'\texttt{)} = \texttt{b(i-1,}j'\texttt{)} + \texttt{b(i,}j'\texttt{)}$

(c)

```
int main() {
  ImageParam b(type_of<double>(),2);
  Func func; Var i, j;
  func(i,j) = b(i-1,j) + b(i,j);
  func.compile_to_file("ex1", b);
  return 0; }
```

(d)

Figure 1. Simplified stencil example. (a) Original Fortran stencil function. (b) Synthesized postcondition. (c) Synthesized outer loop invariant. (d) Halide program to create stencil object file and header.

2.1 Checking Source to DSL Equivalence

To understand verified lifting, it is important to first understand how one would go about *verifying* that a given solution is correct. Specifically, given the code in Figure 1(a), suppose somebody tells us that the code behaves according to the summary shown in Figure 1(b).

Formally speaking, determining whether Figure 1(b) is a correct summary of the computation from Figure 1(a) corresponds to deciding whether Figure 1(b) is a valid *postcondition* for the code. A postcondition is a predicate that will be true at the end of a block of code under all possible executions as long as the inputs to the block of code satisfy some given *precondition*. For STNG, the postcondition is also the lifted summary that we are looking for.

There is extensive literature on how to verify that a block of code is valid with respect to a given pre- and postcondition (e.g., [42, 61]). The idea is to construct a *verification condition*—a formula that if true, implies that the code is valid with respect to its pre- and postconditions. There is a standard approach for constructing verification conditions for a given block of code based on Hoare logic [29], but the approach requires a *loop invariant*, an inductive hypothesis that allows us to prove the postcondition will hold regardless of how many times the loop iterates.

Outer Loop	
initialization	$I_j(\texttt{a},\texttt{b},\texttt{jmin})$
loop exit	$I_j(\texttt{a},\texttt{b},\texttt{j}) \wedge (\texttt{j} > \texttt{jmax}) \rightarrow post(\texttt{a},\texttt{b})$
preservation	(by inner loop initialization & exit)
Inner Loop	
initialization	$I_j(\texttt{a},\texttt{b},\texttt{j}) \wedge (\texttt{j} \leq \texttt{jmax}) \rightarrow I_i(\texttt{a},\texttt{b},\texttt{j},\texttt{imin}+1)$
loop exit	$I_i(\texttt{a},\texttt{b},\texttt{j},\texttt{i}) \wedge (i > \texttt{imax}) \rightarrow I_j(\texttt{a},\texttt{b},\texttt{j}+1)$
preservation	$I_i(\texttt{a},\texttt{b},\texttt{j},\texttt{i}) \wedge (i \leq \texttt{imax}) \rightarrow$ $I_i\big(\texttt{a}\big[(\texttt{i},\texttt{j}) \mapsto \texttt{b(i-1,j)+b(i,j)}\big],\texttt{b},\texttt{j},\texttt{i}+1\big)$

Figure 2. Verification conditions for the running example. I_i,I_j are the unknown loop invariants for the loops over i and j, respectively, and *post* is the unknown postcondition. The expression v[i↦e] means that v contains the same value as before except that the i-th entry is set to e. The program variables a, b, i, j, etc. are implicitly universally quantified.

For example, consider the outer loop from Figure 1(a). The loop invariant for that loop is shown in Figure 1(c). To prove the postcondition, we need to prove three statements:

(1) $\forall s.\ pre(s) \rightarrow invariant(s)$
(2) $\forall s.\ invariant(s) \wedge cond(s) \rightarrow invariant(body(s))$
(3) $\forall s.\ invariant(s) \wedge \neg cond(s) \rightarrow post(s)$

First, we must prove that if the precondition is true for state s, then the invariant is also true (1). Then we must prove that for all states, if the invariant holds and the loop condition is true for some state s, then the invariant should hold for the state obtained after executing the body of the loop once on s (2). Notice that *body* includes the counter increment operation j=j+1, which in Fortran is implicit. Finally, for all states where the invariant holds but which do not satisfy the loop condition, the postcondition should hold as well (3). The conjunction of these three conditions is the verification condition for the loop. If it holds, as is the case with the invariant in Figure 1(c) and the postcondition in Figure 1(b), then the postcondition is sound for the given code.

Figure 2 shows the constructed verification conditions for the running example; they are written in terms of the loop invariants for the inner loop (I_i), the outer loop (I_j), and the postcondition of the block of code ($post$). In general, verification conditions like these can be generated completely mechanically by a syntax-driven algorithm [61] and their validity can be checked using an off-the-shelf theorem prover or SMT solver. This, however, is only possible if the postcondition is known and somebody has provided loop invariants for every loop in the program. In our setting, though, the whole point is that we do not have a postcondition, let alone any loop invariants, so the challenge is to discover postconditions and invariants that make these verification conditions hold.

2.2 From Verification to Synthesis

STNG uses *syntax guided synthesis* [3, 52] to search a large space of possible invariants and postconditions, with the goal of finding an invariant for each loop and a postcondition

that together lead to a verification condition that an off-the-shelf theorem prover can certify as valid. Moreover, because our end goal is to generate a program in a DSL, we have an additional restriction on our postcondition: like the postcondition in Figure 1(b), the postconditions generated by our system should be of a form from which we can trivially extract a target DSL program.

The key concept behind syntax-guided synthesis is that given a space of possible functions and a formula that uses those functions, the synthesizer should be able to efficiently discover the function (or predicate) that will cause the formula to be valid. The syntax guidance in syntax-guided synthesis comes from the fact that the space of programs is defined syntactically, for example via a grammar. The problem of finding invariants and a postcondition such that the verification condition is valid fits nicely into this framework. In particular, the requirement that the postcondition should be translatable to a DSL program is easy to express as a syntactic constraint on the form of the solution.

The problem of syntax-guided synthesis has been suitably formalized into a format called SyGuS with an annual competition for SyGuS solvers [2]. Unfortunately, the problems that arise in our problem domain are difficult beyond the scope of any of these solvers. In particular, there are three features that make our synthesis problems especially challenging.

First, the goal of the synthesizer is to discover postconditions like the one in Figure 1(b), and invariants like the one in Figure 1(c). One aspect that makes these predicates harder to synthesize relative to what existing solvers can handle is the presence of universal quantifiers ($\forall$) in all of them. The presence of quantifiers makes the synthesis problem significantly more challenging and puts it beyond the scope of any SyGuS solver. Secondly, the use of floating point arithmetic in many stencils poses a challenge since modeling floating point or real numbers is more complex than simple integer arithmetic. Additionally, the space of possible invariants and postconditions is astronomically large, so a naïve encoding of the space inevitably leads to an intractable synthesis problem. These constitute the main technical challenges addressed by this paper. §4 explains the key ideas used to solve these problems, but at a high-level, the idea is to leverage what we can learn from observing the execution of the original problem to narrow the search space, and to rely on a combination of abstraction and heuristics to make the problem tractable.

2.3 From an Algorithm to a System

The approach outlined above, of generating a verification condition and then synthesizing postconditions that describe its behavior, is the core algorithm behind verified lifting, but making the algorithm work in the context of a real system poses some additional challenges. Figure 3 shows the overall design of STNG based on the approach described.

Preprocessing To simplify later steps in the process, we begin by preprocessing the code using a frontend built on top

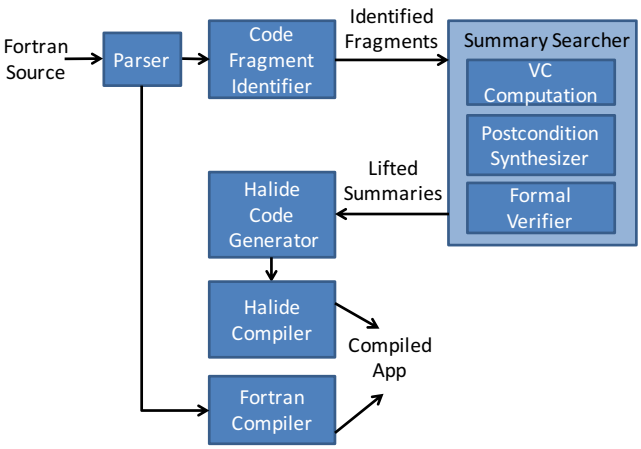

Figure 3. The STNG compiler toolchain

of the ROSE [19] compiler framework. The preprocessing step, described in §5.1, includes finding candidate loop nests that could be stencils, translating each identified loop nest into a simpler intermediate language for synthesis, and generating code to interface with the optimized implementations after each stencil has been converted into DSL code. The intermediate language elides a number of complicated Fortran constructs, canonicalizing them into simpler forms.

Generating DSL Code Once a valid postcondition (i.e., lifted summary) has been found, it is straightforward to translate it into code in the Halide domain specific language. STNG produces a C++ source file that, when compiled and executed, produces an object file that can be linked with the original program. Along with the object file, STNG also produces "glue code" for interfacing between native Fortran arrays and Halide's C++ representation. Thus, the user only needs to modify the build to define a single preprocessor macro; the rest of the build process remains the same as the original code. The generated Halide code can be optimized using empirical autotuning [9] with a Halide autotuner built on top of the OpenTuner [5] program autotuning framework, which automates the search over possible execution schedules for each stencil kernel. The generated Halide code for our running example is shown in Figure 1(d), and §5.3 describes the code generation process in more detail.

3. Synthesis Primer

STNG relies on syntax guided synthesis to discover the unknown invariants and postconditions and to ensure that the resulting postconditions are in a form that can be readily converted into a DSL program.

The key idea behind syntax guided synthesis is to make the synthesis problem more tractable by imposing syntactic restrictions on the space of programs that the synthesizer will consider [3]. Besides making synthesis tractable, the use of syntactic restrictions has the additional benefit of providing

control over the kind of solution that the system will produce. This is crucial in our setting, because there is a large number of possible postconditions that can apply to a given piece of code—i.e., predicates that will be true at the end of its execution—but we are interested in a very specific kind of predicate, one of the form:

$$\forall \vec{x} \in D.\ out[\vec{x}] = expr(\vec{x})$$

where $\vec{x}$ is the vector of all relevant indices for out, D is the domain of indices of out, and $expr$ is an expression that can be converted to the target DSL. If the code under consideration is a stencil, a postcondition like this will allow us to immediately extract what that stencil is. When a loop writes to multiple output arrays, the postcondition is written as a conjunction, and each conjunct is handled separately.

A number of different techniques have been proposed to solve syntax guided synthesis problems, including techniques based on explicit enumeration with pruning [57], stochastic search based on the Markov Chain Monte Carlo method [51], and symbolic search based on constraint solving [54]. For this paper, we rely on the open source SKETCH synthesis infrastructure, which uses constraint-based techniques and includes a full-featured language that makes it possible to efficiently describe complex spaces of programs [52].

The key challenge in synthesis is that the correctness of the desired function is often defined in terms of its behavior under all possible values of some universally quantified variables. In other words, synthesis is an $\exists.\forall$ problem where the goal is to check that there is a function such that for all inputs a predicate involving the unknown function and the inputs is true. For example, in the case of the verification conditions presented in §2, the unknown invariants $invariant$ and postcondition $post$ must make the verification condition valid for all possible states s of the program.

$$\exists post, invariant.\ \forall s.\ VC(post, invariant, s)$$

Synthesizers, including SKETCH, cope with this challenge by relying on Counter-Example Guided Inductive Synthesis (CEGIS) [52]. The key idea behind CEGIS is that instead of searching for a function that satisfies the desired predicate for all possible values of the universally quantified variables, we can search for functions that work correctly under a set of concrete scenarios. For example, instead of checking that the verification condition holds under all possible program states, we can focus on the behavior over some specific state and make sure the invariants and postconditions work for that state. By focusing on concrete states, CEGIS will generate a hypothesis that can then be checked to see if it generalizes to all possible states. If so, we are done. Otherwise, the checking procedure produces a counterexample that is added to the set of concrete states used to produce a new hypothesis.

3.1 Checking Candidate Solutions

At every iteration of CEGIS, the synthesizer must check that the candidate invariants and postcondition satisfy the verification condition for all possible values of the universally quantified variables.

In the case of our system, the checking problem is particularly challenging because the invariants themselves also involve universal quantification. This can be problematic given that the checking procedure has to be run a large number of times until the algorithm converges on a correct solution. To address this, SKETCH uses a hierarchy of checking procedures, starting with purely random search, which finds counterexamples very efficiently in the early stages of the algorithm, and then moving to bounded symbolic checking where it considers inputs up to a small bound. When bounded checking succeeds, our system relies on an SMT solver (Z3 [20]) to do sound validation of the resulting formula.

This final verification step leads to a problem of quantified SMT, which is undecidable in general; however, the number of quantifiers is typically small, ranging over array indices, with the number of array elements updated by any loop-free code block being bounded. In practice, this means that the quantifier instantiation mechanisms built into state-of-the-art SMT solvers, such as Z3 and CVC4, prove very effective. As we will see later in the paper, being able to rely on the SMT solver to do full verification of the quantified formula means that the synthesizer can perform even unsound approximations because if those lead to incorrect solutions, such solutions will be caught in the verification step.

4. Summarizing Stencil Computations

In this section, we describe in detail several new ideas that allow STNG to summarize complex stencil requiring large quantified loop invariants. The overall approach is to construct a synthesis problem that solves a Hoare-style verification problem by searching over a restricted space of possible invariants and postconditions. Once the synthesizer, which uses bounded verification, finds candidate postconditions and invariants, the candidates are used to construct a verification problem that will be checked by a full verifier; it is this latter check that guarantees soundness.

4.1 Syntactic Restrictions

Following the syntax guided synthesis paradigm, the space of possible invariants and post-conditions is constrained by exploiting the assumption that the code is a stencil. The predicate language, shown in stylized form in Figure 4, is based on the theory of arrays [11] but is restricted to stencil-like operations on multidimensional arrays. The predicate language can represent more than just affine operations, including arbitrary function calls (as long as the functions are pure) and indirect array accesses. Values in the output arrays are assumed to be functions of a neighborhood of points around the output point, for some contiguous subset of the overall multidimensional output array.

In the figure and throughout this paper we show multi-dimensional arrays for simplicity, but STNG actually oper-

ates on flattened arrays, which complicate the reasoning. We choose to use this representation based on examining highly-optimized stencil codes in C/C++ in several domains, including high performance computing and image processing; in all of the codes we examined, the stencils operate on flat arrays and use custom indexing macros to access the appropriate points in the array. We make our representation as general as possible in order to facilitate extending the system to C/C++ stencils in the future.

As stated earlier the syntactic restrictions are not just to improve the scalability of synthesis; they also force the synthesizer to avoid trivial postconditions and to ensure that the generated postconditions accurately describe the stencil computation. To this end STNG imposes some additional restrictions on top of those imposed by the grammar. Specifically,

- the range of the index variables used to index output arrays must match the range of the locations that are modified by the kernel;
- each output array is expressed as a single $outEq$ constraint;
- the postcondition is a conjunction of universally quantified $outEq$ constraints;
- and each $outEq$ constraint must have at least one non-output term on the right hand side (to avoid trivial invariants).

These restrictions are too strong to support finding postconditions with boundary conditions, but this limitation is not fundamental to the technique. We explore the potential impact of supporting conditionals in §6.6.

STNG also places restrictions on the structure of the loop invariants. In particular, the invariants are quantified over different subsets of loop variables depending on the nesting structure of the loops and the locations of operations within the loops. The restricted structure of the invariants prevents us from discovering loop invariants for arbitrary loops, but is sufficient to construct valid loop invariants for many important stencil problems.

The syntactic restrictions ensure that the postconditions can be lifted to a DSL, but not all DSLs are complete for the domain of stencils. For example, inputs in Halide are currently restricted to a maximum of four dimensions. Similarly, until the most recent versions of Halide, it was impossible to construct a single Halide function with multiple outputs of different dimensionalities. Since Halide is one of our DSL targets, we work around this by generating code that consists of multiple calls to Halide functions, one for each dimensionality. It is important to point out, however, that conversion to Halide is just one of many possible applications for verified lifting.

4.2 Inductive Template Generation

Even after the restrictions above, the space of possible invariants and postconditions represented by the predicate language is extremely large, resulting in a potentially intractable search space, especially for kernels with multiple input and output arrays. In order to further narrow the space, we analyze each stencil kernel to discover its overall structure and use this to restrict the structure of postconditions and invariants. Our technique is inspired by *concrete dependency tree* creation in Helium [43], which converts from binary traces to image processing code. The algorithm works in two steps.

Symbolic Execution First, STNG performs a combined concrete-symbolic execution on the kernel. We first set loop bounds and array sizes to small, random concrete values, while setting all other inputs to symbolic values, including the values of array elements. Then, the loop nest for the kernel is executed by a simple interpreter that utilizes the SymPy [35] computer algebra system to handle any symbolic values. In this way, values in the output arrays become formulas that consist of a mixture of symbolic and concrete values, since the values in the input arrays are symbolic. For example, when the kernel from Figure 1 is run through this interpreter, the output $a(6,3)$ will have the symbolic value $b[5,3]+b[6,3]$ and, similarly, output $a(4,2)$ equals the symbolic value $b[3,2]+b[4,2]$.

Template Generation The next step is to find templates that are sufficiently general to capture all the observed expressions, but narrow enough that the search space is still tractable. Our system errs on the side of simplicity for this stage. Our approach is to compute the intersection of all the expressions for a given array. More specifically, given two symbolic expressions e_1 and e_2 where each expression is either a terminal or a recursive expression of the form $e_x = (op, e_{x1}, \ldots e_{xn})$ we can compute $\sqcap(e_1, e_2)$ as follows:

$$\sqcap(e_1,e_2) := \begin{cases} e_1 & \begin{array}{l} e_1 = e_2 \\ leaf\,(e_1) \end{array} \\ \left(op\ \{\sqcap(e_{1i},e_{2i})\}_i\right) & \begin{array}{l} e_1 = \left(op\ \{e_{1i}\}_i\right) \\ e_2 = \left(op\ \{e_{2i}\}_i\right) \end{array} \\ MakeHole(e_1,e_2) & (otherwise) \end{cases}$$

where $leaf(e_1)$ means that e_1 is a leaf in the expression tree. Whenever two subexpressions do not agree, they are replaced by a hole created by $MakeHole$ that is able to generate the appropriate set of expressions. In the case of the running example, this process will produce a template of the form `b[pt()] + b[pt()]`, where where `pt()` denotes a *hole*—an expression to be discovered during synthesis. This structure doesn't include the exact accesses, but does encode that each point in the output array is the sum of exactly two distinct points in array `b`.

The process of generalizing from concrete instances to infer a general structure is known as anti-unification in

$$
\begin{array}{lcl}
post & := & \bigwedge_i \; \forall\, lb_1 < v_1 < ub_1,\ \ldots,\ lb_N < v_N < ub_N.\ outEq_i \\
invariant & := & \bigwedge_i \; ineq_i \wedge \forall\, v_1, \ldots v_N . \big(\bigwedge_k bound_{i,k} \big) \rightarrow outEq_i \\
bound & := & lb < v_i < ub \;\big|\; lb < v_i < bndExp \\
outEq & := & out[v_1, \ldots, v_N] = exp \\
ineq & := & \bigwedge_i \big(v_i \; (< \mid \leq) \; bndExp \big) \\
bndExp & := & intvar \;\big|\; c \;\big|\; intvar \; op \; bndExp \;\big| \\
 & & \max(bndExp, bndExp) \;\big|\; \min(bndExp, bndExp) \\
exp & := & term \;\; op \;\; exp \\
term & := & w * in[inIndex_1, \ldots] \;\big|\; floatvar \;\big|\; f(term) \\
inIndex & := & v_i + c \;\big|\; intvar \;\big|\; c \;\big|\; in[inIndex_1, \ldots] \\
op & := & + \;\big|\; - \;\big|\; / \;\big|\; \times
\end{array}
$$

$out \in$ output arrays
$in \in$ input arrays
$lb \in$ loop lower bounds
$ub \in$ loop upper bounds
$v_i \in$ index variable symbols
$f \in$ mathematical (pure) functions
$w \in$ floating point constants
$c \in$ integer constants
$intvar \in$ integer variables and loop counters
$floatvar \in$ floating point variables

Examples of candidate inner loop invariants:

$\forall\, \texttt{imin} + 1 < v_i < \texttt{i},\ \texttt{jmin} \leq v_j < \texttt{j}\ .\ \texttt{a}[v_i, v_j] = \texttt{b}[v_i - 1, v_j] + \texttt{b}[v_i - 1, v_j]$

$\forall\, \texttt{imin} + 1 < v_i < \texttt{i},\ \texttt{jmin} \leq v_j < \texttt{j}\ .\ \texttt{a}[v_j, v_i] = \texttt{b}[v_i + 1, v_j] + \texttt{b}[v_j, v_i - 3] \times \texttt{b}[v_i, v_j] + 2.0$

Examples of candidate postconditions:

$\forall\, \texttt{imin} + 1 < v_i < \texttt{imax},\ \texttt{jmin} \leq v_j \leq \texttt{jmax}\ .\ \texttt{a}[v_i, v_j] = \texttt{b}[v_i - 1, v_j + 1] + \texttt{b}[v_i + 2, v_j - 1]$

$\forall\, \texttt{imin} + 1 < v_i < \texttt{imax},\ \texttt{jmin} \leq v_j \leq \texttt{jmax}\ .\ \texttt{a}[v_i, v_j] = \texttt{b}[v_i - 1, v_i + 2] + 2.0$

Figure 4. Predicate language for expressing loop invariants and postconditions in stylized form (top), with potential loop invariant and postcondition candidates for the running example (middle and bottom). This stylized form elides details such as operator precedence.

the inductive programming community and there are many sophisticated algorithms for it [12, 46, 48]. The procedure above is very simple relative to the state of the art in inductive learning, but it works quite well, especially because we do not consider stencils with conditionals or boundary conditions. In order to generalize to conditionals and boundary conditions we would need to either include conditionals in the symbolic structure or we would need to aggregate trees into multiple "classes" of computations. Even for the uniform case, a more aggressive anti-unification might discover, for example, that the second parameter is always the same for both array accesses, or that the first one is always an offset by one. There is a tradeoff, however, between how aggressively we reduce the search space through anti-unification and the risk that we specialize too much and end up ruling out the correct solution. We did not explore this tradeoff aggressively because we were able to get good results with the simple procedure above.

4.3 Quantifier Elimination With Partial Skolemization

As discussed in §3, synthesis is naturally formulated as a $\exists.\forall$ problem. However, the presence of universal quantification in the invariants introduces an additional quantifier alternation into the formula. To illustrate this problem, consider the encoding of one of the clauses of the verification conditions from our running example in Figure 2. Focusing on the loop exit clause, we want to know if $\exists I_j, post$ such that

$$\forall \texttt{a}, \texttt{b}, \texttt{j}, \texttt{jmax}.\ \neg I_j(\texttt{a}, \texttt{b}, \texttt{j}) \ \vee\ \neg(\texttt{j} > \texttt{jmax}) \ \vee\ post(\texttt{a}, \texttt{b})$$

Inside the invariant I_j, though, is another quantification of the form $\forall j' < \texttt{j}.\ e$, which, because it occurs in negative context, normalizes into $\exists j'$ and gives the overall constraint to be solved the form $\exists.\forall.\exists$ rather than the required $\exists.\forall$. As a result, we cannot apply existing constraint-based synthesis techniques directly to this type of problem.

The standard approach for eliminating these quantifiers is Skolemization [62], which takes a statement of the form $\forall x \exists y$ and replaces the existentially quantified variable y with a new *Skolem function* $f(x)$, where the input to f is the universally quantified variable x. In other words, Skolemization replaces y with a function that determines y based on x. This can be applied repeatedly to eliminate multiple existential quantifiers.

For our SKETCH instance, Skolemization requires finding this function f. One possible approach would be to synthesize the function by building a template based on the loop structure and control flow of the stencil computation, but such a function potentially would need to include conditionals to select its value, based on the current values of the inputs. Instead, we use *partial Skolemization*, which replaces the quantifier $\exists y$ with $\exists y \in f_S(x)$ for small set $f_S(x)$ that depends on x. The idea is that if y sometimes has to be equal to some value $x + i$ and sometimes has to equal some value $x + j$, with full Skolemization, the Skolem function needs to know exactly when to return which, but with partial Skolemization, it can return a set containing $x + i$ and $x + j$, and then $\exists y.P(x, y)$ just becomes $P(x, x + i) \vee P(x, x + j)$. This makes the check a little more complicated than with full Skolemization but saves complexity in synthesizing the Skolem function.

4.4 Dealing With Floating Point Arithmetic

Many stencils work on floating point data and use complicated math functions such as exponentiation. Floating point types complicate synthesis and verification because the data require many bits to represent and because, in general, reordering floating point operations results in slightly different values.

To simplify synthesis with floating point data, we model floating point numbers as an integer field modulo 7, which eliminates dealing with floating point inaccuracies and in practice generates good code. For final verification, we model floating point numbers as real numbers, which guarantees that the summarization is correct with respect to reals. Aiming for a higher level of fidelity with respect to floating point numbers would be futile and unnecesarily restrictive, given that Halide only guarantees correctness relative to real numbers and performs aggressive reordering of operations.

STNG handles math functions, which are side-effect-free, by modeling them as uninterpreted functions. The combination of these techniques allows us to deal with computations over floating point data in a tractable manner.

4.5 Other Optimizations

For each stencil, we generate multiple synthesis problems, each with different optimization strategies, to speed up the search. In particular, some generated problems try to take advantage of common stencil patterns (such as "cross" or box patterns over the input arrays) and other simplifications, such as assuming perfect loop nests, to make the synthesis problem simpler. We run all of the generated problems and for each one that successfully completes, we check the generated postconditions and loop variants using a full verifier.

By default, SKETCH employs bounds checks on arrays for each access. For general synthesis problems, this often speeds up solving time by introducing additional restrictions on the search space, but for our problems, due to the large number of array accesses, these additional assertions make synthesis more difficult. For most of our synthesis templates, we remove bounds checks on arrays, as the quantifiers over the arrays are sufficient to avoid solutions that would result in incorrect accesses.

Because the array accessor functions are synthesized, we consider grid dimensions to be inputs. Our sketches incrementally increase the number of bits for these inputs; in practice, we find 2–4 bits to be sufficient to construct correct invariants and postconditions. Lower numbers of bits sometimes lead to incorrect invariants, but these are caught using the full verifier.

Our overall strategy involves running multiple synthesis problems in parallel, but for each of these problems, we also use a recent feature in SKETCH that allows us to parallelize each synthesis problem [33]. SKETCH does this by choosing some small set of important control bits to concretize, and, for each possible concretization, runs a separate search. If these important values are concretized to the correct value, synthesis proceeds extremely quickly.

5. Architecture and Implementation

In this section, we describe how STNG preprocesses input Fortran code by compiling it into an intermediate representation, and how STNG generates glue code that invokes the converted stencil implementation in Halide from Fortran. We also show how annotations can be added to the original source code to eliminate some corner cases that hinder conversion.

5.1 Code Preprocessing

As discussed in §2, STNG performs a series of analyses to find candidate loops for translation. In this section we describe the steps involved in this process.

Identifying Candidate Loops STNG first iterates through the entire source code to identify intraprocedural loop nests and their outermost enclosing loop construct. For each such loop construct, STNG performs the following lightweight analysis to determine which are candidates for transformation:

- **Array uses.** STNG checks if the loop nest uses arrays, and filters out those that do not along with those whose indices are indirect array accesses or return values from function calls. Any loop that contains such accesses is not considered a candidate.
- **Pointer uses.** STNG supports pointers to arrays. The only complication is that their bounds need to be determined using runtime mechanisms (to be discussed below). Otherwise they are treated the same way as non-pointer arrays.
- **Conditionals, function calls, and unstructured control flow.** STNG currently does not handle loop nests that contain conditional statements, make calls to Fortran procedures (except for pure functions and intrinsics that are modeled as uninterpreted functions in our intermediate language, as described in §4), or contain unstructured control flow statements such as `break` and `continue`. There are no fundamental reasons for not handling such constructs except for engineering effort; for instance, rewriting unstructured control flow using standard algorithms [65] would allow us to handle them, and conditionals pose no difficulties for axiomatic semantics.

If consecutive (in terms of control flow) loop nests satisfy the criteria listed above, they are combined into a single code fragment to be considered for conversion. An example of this would be two consecutive loop statements.

Processing Selected Loops Each loop construct that passes the checks listed above is compiled to an intermediate representation, which is a simplified version of the original Fortran code (e.g., all loops are rewritten as `while` loops, complex expressions are broken down into binary ones, etc). To allow easy integration of the translated code, STNG extracts each candidate loop into a separate Fortran function with appropriate parameters that are passed in and returned. If the loop can be converted into Halide, it is passed to the backend code generator. In addition, the original loop is replaced by a call to the generated Halide code, with appropriate setup parameters passed to it. These include starting and ending indices for the stencil computation, the array contents to be operated on, and any other program variables that are needed

for the computation. For those loops that fail translation, the original Fortran code is retained instead. The entire toolchain is automatic and does not involve any developer intervention.

5.2 Annotations

While the process described above is completely automatic, there are certain cases where STNG fails to convert the input code due to conservative assumptions. In such cases, STNG allows users to provide additional preconditions about the source code using annotations. For example, one construct we observed in real-world code is an array accessor of the form `a[i*(sz0-sz1)]`, where `i` is a loop counter. If `sz0` = `sz1`, these accesses all refer to the same element, which may make it impossible to find a loop invariant for this array without running expensive pointer analysis.

Instead, users can provide assumptions on the input code using Fortran comments of the form `STNG: assume(e)`. Here, `e` is a boolean-valued expression that the user specifies and is assumed to be true. For the example mentioned, an annotation `STNG: assume(sz0 != sz1)` can be used to indicate distinct accesses to the array. We do not anticipate users needing to provide many annotations. In fact, of the 77 loops that were translated, only 6 required annotations in order to be lifted.

5.3 Code Generation

Once STNG synthesizes the postcondition for a code fragment, it constructs a Z3 script to verify the postcondition is correct. If this succeeds, the synthesized fragment is transformed into backend code in Halide. The backend code consists of a small C++ program that specifies both the algorithm and the *schedule*, which describes how to execute the algorithm. When executed, this program generates an object file and header that STNG uses to replace the original code.

We use Halide's autotuner, based on the OpenTuner [5] framework, to tune the schedules for the backend code. The autotuner uses an ensemble of techniques combined with a multi-armed bandit to search over the large space of candidate schedules. In general, the space of possible schedules is far too large to explore exhaustively, so the combination of multiple machine learning techniques is essential to discovering best schedules while searching only a fraction of the space.

Two complications make the transformation from postconditions to Halide non-trivial: in Halide, all loop bounds are implicit and come from the logical sizes of the output grids. Unlike our intermediate representation, which uses flattened single-dimensional infinite arrays, Halide operations occur on logical multidimensional grids.

To resolve this issue, STNG synthesizes accessor functions for input and output arrays in terms of the constants that are live at the entry to the code fragment. We use symbolic interpretation to convert these accessors back into multidimensional grid accesses, by considering a neighborhood of points and matching points with the resulting one-dimensional array access. This symbolic interpretation mechanism is built on top of SymPy [35], the symbolic math library for Python.

In addition to using symbolic simplification for array accesses, STNG also simplifies the postcondition to make the Halide code shorter and more readable. As an illustration, the generated Halide code from our running example is shown in Figure 1(d), with the schedule elided due to space.

Loop bounds are calculated from the original code. As part of the conversion process described in §5.1, STNG outputs "glue code" that converts loop bounds and array sizes into Halide data structures with logical bounds such that the Halide code operates on the same array elements as the original code.

5.4 Limitations

Our current prototype cannot handle code fragments that contain conditional statements. It also only handles loops with monotonically increasing loop variables. Handling these aspects requires more engineering effort and we do not believe they represent any fundamental limitations of our approach.

6. Experimental Results

We evaluate STNG by using it to lift stencil kernels from microbenchmarks, mini-apps, and full applications in Fortran, followed by converting the lifted summaries into Halide. In this section, we describe the experiments we conducted and evaluate the effectiveness of STNG.

6.1 Experimental Setup

All experiments are run on a 24-node cluster of dual-socket Intel Xeon E5-2695v2 machines running at 2.4GHz. Each socket has 12 cores, and each node has 128 GB of memory. The cluster runs Ubuntu Linux 14.04 LTS (kernel version 3.13.0-52-generic, GCC 4.8.4, LLVM 3.4, Intel Compiler 16.0.0). Halide code is autotuned for 3600 minutes to find optimal schedules, using the OpenTuner framework [5]. We test STNG on the following examples.

The **StencilMark** microbenchmarks [36] are a set of stencil microbenchmarks for performance and compiler experimentation. We choose the four 3D kernels in the suite and port them manually to Fortran.

NAS MG [6] is a standard benchmark that implements the multigrid algorithm in 3D to solve a discrete Poisson equation using multigrid V-cycles and multiple levels of refinement. It is a challenging problem for STNG due to existing optimizations in the code.

CloverLeaf [40] implements a Lagrangian-Eulerian hydrodynamics code on a Cartesian grid using an explicit second-order method. Part of the Mantevo [28] mini-apps project, it solves the compressible Euler equations on a 2D staggered grid.

TERRA [8] simulates mantle convection/circulation using a spherical shell model. The code uses five- and six-dimensional arrays to represent simulation quantities. We

Benchmark	Kernel	Halide Speedup	icc Before Speedup	icc After Speedup	Halide GPU Speedup	Halide GPU Speedup (no transfer)	Sketch Time (s)	Control Bits	Postcon AST Nodes
Cloverleaf	akl81	7.44	1.73	4.53	4.09	10.88	14944	105	143
	akl83	4.57	0.95	0.95	3.29	8.27	683	97	129
	akl84	4.51	0.71	0.94	2.94	7.88	631	97	129
	akl85	4.05	0.95	0.77	2.68	6.85	662	97	129
	akl86	4.04	0.97	0.79	2.27	6.00	919	97	129
	ackl95	4.19	1.00	1.00	1.94	7.47	1088	80	82
	amkl100	3.84	0.93	0.89	1.57	6.72	1512	80	82
	amkl101	3.64	1.01	1.00	1.52	5.44	1273	62	83
	amkl103	3.42	0.93	0.93	1.99	6.77	176	39	57
	amkl105	3.37	0.98	0.98	1.50	5.17	707	70	83
	amkl107	3.95	0.94	0.96	2.31	8.71	133	39	57
	amkl97	4.19	1.00	1.01	1.96	6.98	4099	115	136
	amkl98	5.41	1.31	1.32	1.86	7.12	4191	115	136
	amkl99	4.15	0.99	0.98	1.78	7.44	1736	80	82
	fckl89	4.75	0.96	0.96	3.07	9.47	425	40	87
	fckl90	3.55	0.96	0.93	2.31	6.60	131	56	87
	gckl77	2.53	1.00	1.00	1.09	6.53	7	10	23
	gckl77	3.65	0.99	0.99	1.02	5.65	8	10	23
	gckl79	3.77	0.98	1.00	1.04	6.59	6	10	23
	gckl80	2.57	0.99	1.00	1.11	5.78	7	10	23
	ickl10	3.37	0.99	0.75	1276.63	3226.12	2	4	25
	ickl11	4.09	0.98	1.00	569.34	2090.13	1	8	13
	ickl12	6.12	0.99	0.99	924.28	5895.51	1	4	25
	ickl13	3.89	1.00	1.00	1462.36	5928.43	1	8	13
	ickl14	3.10	1.00	1.00	1.04	5.59	5	5	23
	ickl15	2.67	1.00	1.00	1.04	6.67	13	10	23
	ickl16	2.82	1.12	1.12	1.04	6.78	7	10	23
	ickl8	4.10	1.01	1.00	1344.48	4902.77	1	4	13
	ickl9	4.11	1.21	1.22	719.55	2704.02	1	4	13
	rfkl109	3.47	0.94	0.95	1.59	6.80	6	10	31
	rfkl110	2.85	0.95	0.93	1.22	6.96	5	10	31
	rfkl111	3.00	0.94	0.94	1.56	6.78	6	10	31
	rfkl112	2.85	0.93	0.92	1.56	7.71	7	10	31
	ackl91	5.19	0.91	1.12	2.13	8.24	6361	115	136
	ackl92	4.79	0.91	0.90	1.60	7.84	1542	80	82
	ackl94	4.77	0.97	0.96	1.95	7.20	4273	115	136
	ackl102	5.27	1.02	1.00	2.30	6.16	9153	105	139
	ackl106	4.69	1.02	1.00	7.83	24.90	5546	105	139
	rkl87	2.99	0.94	0.94	1.55	7.54	5	10	31
	rkl88	3.04	0.96	0.95	1.18	6.67	5	10	31
NFFS-FVM	calcph0	6.32	1.08	0.19	1.45	5.96	17	112	90
	calcph1	13.35	0.81	0.65	2.47	6.47	69	748	314
	geomet0	3.56	0.97	0.98	5.57	6.11	6	4	21
	geomet1	3.45	0.99	0.49	6.13	6.13	9	4	13
	geomet10	3.75	0.99	0.90	6.75	6.81	8	4	17
	geomet11	3.94	0.99	1.00	4.66	4.79	5	4	9
	geomet12	6.94	1.18	0.95	13.69	14.37	4	4	11
	geomet13	3.44	0.97	0.52	4.47	4.53	8	4	22
	geomet14	3.09	1.00	0.92	6.78	6.80	10	4	38
	geomet15	5.05	1.00	4.30	4.51	4.54	16	4	38
	geomet16	6.47	0.99	4.33	4.51	4.52	13	4	38
	geomet17	4.86	1.00	4.29	4.54	4.54	20	4	38
	geomet2	3.71	0.99	0.51	6.13	6.11	5	4	9
	geomet3	4.09	0.99	0.81	10.80	11.02	13	4	9
	geomet4	5.89	0.99	0.78	13.55	13.76	14	4	11
	geomet5	3.99	0.99	0.51	4.45	4.52	8	4	22
	geomet6	2.96	0.98	0.87	15.93	16.11	8	4	9
	geomet7	4.14	0.99	0.81	11.10	11.51	8	4	9
	geomet8	5.49	0.99	0.80	13.58	13.58	12	4	11
	geomet9	3.70	0.98	0.53	4.50	4.54	6	4	22
	initial0	24.14	19.56	1.27	12.82	75.96	32004	4900	537
	initial1	4.40	2.52	2.57	1.91	8.11	75	4	33
	meclfu0	10.04	8.63	0.89	2.48	11.34	94	1104	246
	simple0	5.69	4.72	1.11	2.00	12.54	494	24	54
	simple2	11.84	8.27	0.07	1.92	4.23	38	424	171
NAS MG	mgl15	5.69	5.73	1.44	3.69	15.95	8	17	25
	mgl18	5.89	6.08	1.47	3.74	13.37	7	17	25
	mgl5	17.51	1.52	5.23	2.38	4.30	46422	1100	455
StencilMark	div0	9.69	4.80	0.63	1.54	3.74	6590	224	131
	heat0	11.30	1.45	1.10	1.43	3.82	4668	172	131
	grad0	6.49	1.18	1.10	4.46	10.32	18557	680	177
Challenge	heat27	1.84	1.35	1.35	0.28	0.30	49692	2812	457
	heat27u	12.27	0.00	9.04	1.10	1.20	9162	2812	457
	heat27b1	12.23	0.00	9.01	1.10	1.20	57074	3514	451
	heat27b2	12.26	0.00	9.03	0.97	1.06	64439	2110	455
	heat27pl	1.84	1.35	1.35	0.28	0.30	49692	2812	457

Table 1. Overall lifting results. Speedups are compared to `gfortran` on the original code.

were provided one of the computational kernels from this application to test our methodology by one of the authors of the code.

NFFS-FVM[58, 59] simulates 3D fluid mechanics and heat transfer for viscoplastic non-Newtonian flows using the finite volume method. This full application consists of many stencil kernels and has been used to study OpenMP parallelization for finite volume fluid simulation. We were provided a copy of this code by one of the authors.

We manually constructed **Challenge Problems**, a set of 27-point 3D stencils based on optimizations used in real-world stencils found in prior work [18, 37]. These stencils use a combination of loop tiling and unrolling to improve cache utilization. The resulting code is complex, with non-affine loop bounds and deeply-nested loops.

6.2 Lifting Effectiveness

We first address the generality of the approach and its ability to lift stencil kernels from a variety of applications. Table 2 summarizes the results of applying STNG to our suite of test codes. The *Candidates* column in Table 2 states how many loops were flagged as candidates for translation, and the *Translated* column indicates how many out of those were actually stencils and are translated by STNG. As described in §5.1, the criteria for flagging a loop for analysis is quite liberal, so not everything that is flagged as a candidate stencil is actually a stencil; the column *Untranslated Stencils* in Table 2 indicates how many of the kernels that were flagged but not translated were actually stencils. Table 1 shows the full array of stencils translated and their details. Although we show the number of AST nodes just for the postcondition, the sizes of the invariants are almost exactly the same.

	Candidates	Translated	Untranslated Stencils	Non Stencils
StencilMark	4	3	1	0
NAS MG	9	3	5	1
CloverLeaf	45	40	4	1
TERRA	1	1	0	0
NFFS-FVM	29	25	1	3
Challenge	5	5	0	0
Total	**93**	**77**	**11**	**5**

Table 2. Summary of lifted kernels. Because our front-end processing liberally marks loop nests as potential stencils, we also show how many actual stencils STNG was not able to translate.

A few of the kernels in our sample were relatively simple and could have been translated with a less sophisticated technique based, for example, on pattern matching. However, out of the kernels we translate there were at least 23 that involved some form of manual optimization and required non-trivial reasoning from the synthesizer.

For example, the 3D kernels in NAS MG, NFFS-FVM, StencilMark, and the challenge suite, as well as the 5D kernels in TERRA, are among the most challenging to lift and represent the longer synthesis times shown in Table 1. Our verified lifting methodology generates more complex synthesis problems in higher dimensions and when many points from the same input kernels are used. The former is due to needing to represent larger flattened arrays, while the latter is due to the increased search space even after applying inductive template generation. The complexity of the synthesis problems can be seen from looking at the control bits in Table 1, which range from 4 bits for simple kernels to thousands for the high-dimensional complex stencils. In addition, the size of the synthesized postcondition (in terms of the number of syntax tree nodes) is shown in the rightmost column. From these two measures, it is apparent that STNG is able to synthesize complex postconditions and invariants for complex stencils that result in difficult synthesis problems.

Of all the translated kernels, 6 require programmer annotation to enable correct invariant and postcondition construction. The programmer annotation required in each case specified that the dimensions of the array were at least as large as loop iteration space, meaning that no out of bounds accesses occurred and not all writes were to the same array location.

Of the 11 kernels we could not translate, 9 failed due to engineering issues that are not fundamental to our algorithm. For example, our system is currently restricted to incrementing loops, so 4 kernels failed because they used decrementing loops, but would have otherwise succeeded. Only 2 timed out for scalability reasons.

The most challenging kernel in our suite is a 27-point kernel with tiling, resulting in a 4-deep loop nest with each invariant requiring an expression of 455 AST nodes. This produces a synthesis problem with over 3500 nodes. STNG is able to find the summary for this extremely large problem after 17 hours.

6.3 Performance of Lifted Code

We next consider whether lifting allows us to take advantage of domain-specific languages and their specialized compilers. The first column of Table 1 shows the performance of the lifted code after the summary is translated to Halide and autotuned using OpenTuner [5] to run on a single 24-core node, relative to the original code compiled with `gfortran`. The median speedup is $4.1\times$ across the 77 kernels, ranging up to $24\times$, with a minimum speedup of $1.84\times$. Thus, for every kernel, lifting and translating to Halide plus autotuning increases performance. For reference, we also show the performance of the original code using `ifort -parallel` in the second column. In the vast majority of cases, lifting and translating to a DSL outperforms auto-parallelization; in fact, the median speedup with that approach is $1.0\times$.

6.4 Lifted Code Portability

Lifting also makes it possible to execute code on new platforms, by taking advantage of DSL backends. We execute the lifted code on a GPU by leveraging the ability of Halide to

target multiple platforms. In this case, we construct a naïve GPU version in Halide by changing a single line of code in our code generation. Thus, using STNG, we can make the kernels execute on the GPU in a fully-automatic way. Table 1 shows the speed of running the generated Halide on the GPU both with and without the time to transfer results back. Note that we had to reduce the sizes of the data in order to get it to fit on our GPU (Nvidia K80) compared to the CPU version. Even with the cost of transfering data between the CPU and GPU, several kernels execute far faster on the GPU; many of these compute reductions, so have little data to commuinicate. Others, such as `ackl106`, which gets a $7.8\times$ speedup, are traditional 2D stencils with lots of computation. The autotuning system provided by OpenTuner does not tune for the GPU, so there is room to further increase performance. Nevertheless, this shows the ability of lifting to take advantage of code generation for new architectures, increasing portability.

6.5 Lifting as Deoptimization

Finally, we look at a last use of lifting. It is difficult for compilers to easily reason about and transform hand-optimized code, because these optimizations often introduce artificial complexity such as non-affine loop bounds caused by tiling. When STNG creates a summary for a kernel, it doesn't contain the complex control flow and transformed code that was present in the original hand-optimized version. Thus, the summary can be seen as a clean deoptimized version of the input code.

To test this hypothesis, we implement a simple serial code generator from STNG summaries to C++, and look at the suite of challenge problems (the last 5 rows of Table 1). For the ones with the most complicated transformations, the Intel compiler is unable to obtain any speedup— in fact, the codes perform four orders of magnitude worse, as seen in the second column of the table, which shows the speedup of running the Intel compiler with auto-parallelization on the original code. The hand optimizations trigger pathological behavior in the compiler's auto-parallelization and optimization. The third column, which is the performance of running the auto-parallelizing compiler on the generated code, however, shows up to $9\times$ speedups on these codes. This is because it essentially gets the performance of `heat27`, the non-hand-optimized version of the code. Note that this doesn't apply in all cases; for some codes, the Intel compiler is unable to auto-parallelize the code. Clearly, however, in some cases deoptimization through lifting makes the job of applying transformations in the compiler easier.

6.6 Impact of Conditionals

The current implementation of STNG does not support conditionals, and therefore cannot translate stencils with boundary conditions or output values dependent on conditional expressions. To explore the potential impact of conditionals on the scalability of synthesis when lifting stencils, we hand-modify the SKETCH problem for one of our benchmark problems

```
do k = y_min, y_max+1
do j = x_min, x_max+1
  if (cond) then
   xvel1(j,k) = expr1
  else
   xvel1(j,k) = expr2
  endif
enddo
enddo
```

(a)

$$cond \quad := \quad arr_{in}(j+?, k+?) \quad (\leq \mid \geq \mid < \mid > \mid = \mid \neq) \quad (R \mid in_{float})$$

$arr_{in} \in$ input arrays $\quad ? \in$ integers
$R \in$ floats $\quad in_{float} \in$ float inputs

(b)

$$cond \quad := \quad (j \mid k) \quad (\leq \mid \geq \mid < \mid > \mid = \mid \neq) \quad (? \mid in_{int})$$

$? \in$ integers $\quad in_{int} \in$ integer inputs

(c)

Figure 5. (a) Fortran stencil with conditional used in our experiment to measure the impact of conditionals on synthesis time. (b) Grammar for data-dependent conditional. (c) Grammar for location-dependent conditional.

(`akl83`) and measure the impact on synthesis time. We consider two different kinds of conditionals: data-dependent and location-dependent.

Data-dependent conditionals are conditionals that branch on the value of an input point. For this experiment, we attempt to lift a stencil of the form in Figure 5(a), where the conditional `cond` is given by the stylized grammar in Figure 5(b). Currently, SKETCH does not support boolean comparisons with floating point values, so the comparisons use uninterpreted functions. The resulting SKETCH problem contains 160 control bits (compared to 97 for the original problem) and takes 4445 seconds to synthesize, a slowdown of $6.5\times$ compared to the original.

Location-dependent conditionals are commonly used for boundary conditions, where the cells at the grid borders are given by a different expression than those in the interior. The grammar for our location-dependent experiment is shown in Figure 5(c), and is simpler than the data-dependent grammar, requiring 154 control bits. The synthesis time is 736 seconds, which is only $1.1\times$ slower.

These experiments demonstrate that conditionals for boundary conditions are less difficult to deal with for our approach, while data-dependent conditionals do increase the complexity of the synthesis problems and synthesis time. Speeding up such problems, as well as adding support for conditionals in the complete toolchain, is a goal for future work.

In summary, we have shown that STNG can successfully lift a large set of stencils from real-world codes, benchmarks, and hand-optimized implementations. The sizes of invariants, postconditions, and the resulting synthesis problems are very large, reflecting the true complexity of the stencils we lift. Lifting enables leveraging domain-specific compilers for performance, enables portability, and simplifies code that has been hand-optimized.

7. Related Work

Stencil DSLs Domain-specific languages for stencil computations are a rich area of recent work, due to their importance and the poor performance obtained through most stencil libraries. PATUS [16] and Kamil et al. [37] build auto-tuned systems that optimize Fortran code using a library of transformations, but neither has any soundness guarantees. PolyMage [44] is a DSL similar to Halide that uses polyhedral compilation instead of user-guided optimization. Pochoir [56] uses a cache-oblivious parallel approach to stencils with a DSL embedded in C++ for defining them. It is particularly successful at optimizing multi-timestep stencils. The corresponding cache-aware approach is called time-skewing [63]. In both cases, developers need to rewrite their existing code to take advantage of the specialized implementations. A mixed polyhedral/SIMD code generation approach [39] for optimizing stencils can take a very restricted inner loop and create an optimized SIMD code generator.

In program synthesis, the earliest implementations of SKETCH only worked on small finite programs. Solar-Lezama et al. [53] describes a reduction algorithm that transforms stencils written in SKETCH, which usually have unbounded input sizes, into bounded circuits, which can then be used in the synthesis algorithm to fill any holes in the reduced program. The synthesized values for the holes can be directly plugged into the original stencil program, since the reduction is lossless.

Compiler Optimizations There has been a lot of work done in optimizing the kinds of loop nests that describe stencils, both in the programming systems and high-performance computing research communities. Much of this work has concentrated on polyhedral optimization [4, 22, 26, 27, 31], where loop nests are optimized by treating the iteration space as a polyhedron and determining an efficient traversal of the space. The polyhedral method applies to more than just stencil computations, and we see it as a complementary approach: STNG can rewrite optimized stencils to simpler forms amenable for polyhedral optimization. Recent work has also applied synthesis to convert code to utilize SIMD instructions [7].

Aside from DSLs, there is a rich history of stencil-specific optimization techniques [24, 49] and this continues to be an important area of research. Most recently, Helium [43] uses brute-force analysis of binary traces to attempt to derive equivalent Halide kernels from executions of image processing code. The technique is dynamic and unsound, while STNG is static, based on source code, and is sound.

Superoptimization [41] is another technique used to improve code performance by finding more efficient implementations. While there have been systems constructed to optimize general-purpose code [34, 50], including a recent system to generate code for specialized architectures [45], we are not aware of any such systems that target stencil-specific codes.

Inferring Invariants Inferring loop invariants has been an active area of research. Classical approaches include using predicate refinement [23], and dynamic detection [21]. Unlike most prior work, STNG does not aim to discover the strongest loop invariant and postconditions. Instead, we aim to find postconditions that are strong enough for the purpose of lifting the given loop. As is commonly known in verification, the required strength of the invariant depends on the desired postcondition—stronger postconditions usually require richer invariants. IC3 [10] takes advantage of this by directing the search towards an invariant that is sufficient to prove the postcondition. It was originally developed as a hardware model-checking technique, but proved useful for software as well [17, 38]. STNG focuses on a certain family of postconditions, namely those that admit a semantically equivalent implementation in Halide, which naturally implies a corresponding family of adequate invariants. This makes inferring the invariant more tractable while not limiting the results from synthesis. In principle, STNG can use strong invariants from an external source if they are available, and conversely, an invariant generated by STNG can also be used as a starting point for other inference tools to refine (e.g., IC3). Recent work for inferring numerical invariants [25] constructs invariants for code snippets without the semantic restrictions of STNG (i.e., it is not restricted to stencils), but those invariants are far simpler than what are needed to validate our inferred postconditions.

In the database research community, there has been work in inferring invariants and postconditions in transforming imperative code into database queries [15, 32, 60]. We follow QBS [15] in using constraint-based program synthesis to come up with invariants and postconditions. Unlike QBS, the language of invariants and postconditions in STNG is based on the theory of arrays, and the different application domain requires several advances in the verified lifting approach: (1) because invariants in this domain are universally quantified, STNG uses Skolemization to make synthesis tractable; (2) the presence of floating-point arithmetic complicates reasoning, requiring a simplified representation to make analysis tractable; and (3) STNG must use novel techniques, including inductive template generation via mixed concrete-symbolic execution, to enable scalability in the presence of much larger invariants.

8. Conclusion

In this paper we presented STNG, a novel system for lifting stencil computations. Rather than using traditional syntax-driven techniques to optimize input programs, STNG uses verified lifting to convert the input code written in a general-purpose language into a high-level representation, which can then be compiled to a DSL and leverage its specialized optimization techniques. We have implemented a prototype of STNG and evaluated using real world benchmarks, showing STNG's ability to lift complicated kernels from real-world code without boundary conditions, including hand optimizations used by stencil programmers. Generating DSL code from the STNG summaries results in median performance improvements of $4.1\times$ and as much as as $24\times$.

Acknowledgements

This work was partially supported by DOE Office of Science Awards DE-SC0005288 and DE-SC0008923.

References

[1] Apache Hive. `http://hive.apache.org`.

[2] Syntax-Guided Synthesis Competition. `http://www.sygus.org`.

[3] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17, 2013.

[4] Saman P Amarasinghe, Jennifer-Ann M Anderson, Monica S Lam, and Chau-Wen Tseng. An overview of the SUIF compiler for scalable parallel machines. In *PPSC*, pages 662–667, 1995.

[5] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[7] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 123–134, 2013.

[8] John R. Baumgardner. Three-dimensional treatment of convective flow in the earth's mantle. *Journal of Statistical Physics*, 39(5-6):501–511, 1985.

[9] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[10] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, 2011.

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] Peter E Bulychev, Egor V Kostylev, and Vladimir A Zakharov. Anti-unification algorithms and their applications in program analysis. In *Perspectives of Systems Informatics*, pages 413–423. Springer, 2010.

[13] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programmable Models for Emerging Architecture (PMEA)*, 2009.

[14] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal High Performance Computing Applications*, 21(3):291–312, 2007.

[15] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[16] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *International Parallel Distributed Processing Symposium (IPDPS)*, pages 676–687, May 2011.

[17] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, 2012.

[18] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] Kei Davis and Daniel J. Quinlan. ROSE: An optimizing transformation system for C++ array-class libraries. In *Workshop on Object-Oriented Technology*, ECOOP '98, pages 452–453, London, UK, UK, 1998. Springer-Verlag.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[22] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[23] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 191–202, New York, NY, USA, 2002. ACM.

[24] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[25] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer International Publishing, 2014.

[26] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[27] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014.

[28] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratory, 2009.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[30] Intel. The X10 parallel programming language. `http://x10-lang.org`.

[31] Francois Irigoin and Remi Triolet. Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA, January 1988.

[32] Ming-Yee Iu and Willy Zwaenepoel. HadoopToSQL: A MapReduce query optimizer. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 251–264, New York, NY, USA, 2010. ACM.

[33] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. Adaptive concretization for parallel program synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 377–394, 2015.

[34] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. pages 304–314, 2002.

[35] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E. Granger. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra*, 45(3/4):225–234, January 2012.

[36] Shoaib Kamil. A cross-domain stencil benchmark suite. In *Workshop on Optimizing Stencil Computations*, 2013.

[37] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, 2010.

[38] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 583–602, 2015.

[39] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[40] A.C. Mallinson, D.A. Beckingsale, W.P. Gaudin, J.A. Herdman, J.M. Levesque, and S.A. Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. In *Cray User Group*, 2013.

[41] Henry Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[42] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'06, pages 362–376, 2006.

[43] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland, OR, June 2015.

[44] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[45] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM.

[46] Gordon D Plotkin. A note on inductive generalization. *Machine Intelligence*, 5(1):153–163, 1970.

[47] 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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[48] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 135–151. Edinburgh University Press, Edinburgh, Scotland, 1969.

[49] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[50] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *SIGPLAN Notices*, 48(4):305–316, March 2013.

[51] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 9, 2014.

[52] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[53] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodk, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 167–178. ACM, 2007.

[54] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[55] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions in Embedded Computer Systems*, 13(4s):134:1–134:25, April 2014.

[56] Yuan Tang, Rezaul Chowdhury, Bradley C. Kuszmaul, Chi keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *ACM Symposium on Parallelism in Algorithms and Architectures, (SPAA)*, 2011.

[57] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296, 2013.

[58] Diego Vasco, Nelson Moraga, and Gundolf Haase. Parallel finite volume method simulation of three-dimensional fluid flow and convective heat transfer for viscoplastic non-newtonian fluids. *J. Numerical Heat Transfer, Part A: Applications*, 66(2):990–1019, 2014.

[59] Diego Vasco, Nelson Moraga, Aurel Neic, and Gundolf Haase. OpenMP parallel acceleration of a 3D finite volume method based code for simulation of non-Newtonian fluid flows. In Sergio Gutiérrez, Daniel Hurtado, and Esteban Sáez, editors, *Cuadernos de Mecánica Computacional*, pages 108–117, Concepción, Chile, 2011. Sociedad Chilena de Mecánica Computacional.

[60] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 19–36, 2008.

[61] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[62] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.

[63] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[64] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[65] Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, 2004.