Sketching Stencils

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat,* Sanjit Seshia

University of California, Berkeley
{asolar,arnold,tancau,bodik,sseshia}@eecs.berkeley.edu

* IBM T. J. Watson Research Center
vsaraswa@us.ibm.com

Abstract

Performance of stencil computations can be significantly improved through smart implementations that improve memory locality, computation reuse, or parallelize the computation. Unfortunately, efficient implementations are hard to obtain because they often involve non-traditional transformations, which means that they cannot be produced by optimizing the reference stencil with a compiler. In fact, many stencils are produced by code generators that were tediously handcrafted.

In this paper, we show how stencil implementations can be produced with sketching. Sketching is a software synthesis approach where the programmer develops a partial implementation—a sketch—and a separate specification of the desired functionality given by a reference (unoptimized) stencil. The synthesizer then completes the sketch to behave like the specification, filling in code fragments that are difficult to develop manually.

Existing sketching systems work only for small finite programs, *i.e.*, programs that can be represented as small Boolean circuits. In this paper, we develop a sketching synthesizer that works for stencil computations, a large class of programs that, unlike circuits, have unbounded inputs and outputs, as well as an unbounded number of computations. The key contribution is a reduction algorithm that turns a stencil into a circuit, allowing us to synthesize stencils using an existing sketching synthesizer.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Software Architectures, Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design, Performance

Keywords Sketching, Stencil, SAT

1. Introduction

Sketching is a program synthesis paradigm in which a programmer expresses an outline of the implementation, called a *sketch*. The details missing in the sketch are filled in by the compiler such that the result is functionally equivalent to a separately provided specification. Programming by sketching is supported by the SKETCH language [21], an imperative language with a "hole" construct. Holes can be used in place of hard-to-get-right expressions, such as index expressions and loop bounds. The SKETCH synthesizer infers the content of holes using a SAT-based combinatorial search over the space of possible sketch completions. Thanks to the combinatorial search, the synthesizer requires no domain-specific knowledge and can therefore be applied to a wide variety of programming problems, as long as they compute *finite programs*, *i.e.*, functions that take inputs of bounded size and perform a finite computation.

SKETCH has been used to synthesize efficient implementations for non-trivial functions—including ciphers (*e.g.*, DES and AES), error-correction codes, and long integer multiplication—in only a fraction of the time required for coding a full and correct implementation.

Scientific computing codes could equally benefit from sketching. Despite advances in optimizing compilers, these codes are often hand-crafted through a long and error-prone process. The challenges arise in particular when transforming a loop iteration space for better locality and computation reuse. For example, a complex tiling strategy may entail index and loop bound expressions that are hard to write and debug.

Unfortunately, most scientific codes are not finite programs: their input size is unbounded, and hence SKETCH is unable to synthesize them. While we can finitize some scientific kernels by fixing their problem size, the combinatorial problem easily overwhelms the SAT solver. (In fact, SKETCH already fails to synthesize a simple kernel on an 8x8 matrix.)

This paper introduces sketching for stencil-based scientific kernels. Stencil kernels transform an arbitrarily large input grid into an output grid, mostly by applying constant-time, nearest-neighbors computations. This domain includes many widely-used iterative PDE solvers, including the popular Jacobi and MultiGrid [3] methods. These solvers constitute the core of many structured grid applications, such as elasticity and fluid dynamics simulations.

In order to handle the unbounded nature of stencils, we develop a program transformation that reduces a stencil into a finite program. The reduction is enabled by the internal boundedness of a stencil, a property that each output grid element is a function of only a finite (and small) number of input grid elements. The reduction process effectively extracts this function from an imperative program with loop nests. The reduced stencil is then synthesized using the finite SKETCH synthesizer. Because the reduction is lossless, we can plug the synthesized holes back into the original sketch to obtain a complete and correct implementation of the stencil.

We have used sketching to implement a recursive, cache-oblivious implementation of a 2-point stencil. The paper also describes our experience implementing two kernels from the MultiGrid solver. The implementation of two MultiGrid kernels took less than two hours of work by one of the authors, including synthesis time, which did not exceed 5 minutes. Both sketches produced implementations that were 2 to 8-times faster than the corresponding naive stencil.

In summary, this paper makes the following contributions:

PLDI'07 June 11–13, 2007, San Diego, California, USA.

- It presents a sketching synthesizer for the class of stencil computations. It is the first system for sketching programs that go beyond the finite programs of the original SKETCH language and synthesizer [21].
- It develops and evaluates a reduction mechanism for transforming an unbounded stencil into a finite function for which an efficient synthesizer is available.
- It shows that the reduction based approach scales to real-world implementations of kernels of interest to the scientific-computing community, and that it is able to synthesize implementations which are beyond the reach of commercial optimizing compilers.

2. The SKETCH Programming Language

To give necessary background for the sections that follow, this section summarizes the finite sketch synthesis method introduced in [21] and implemented in the SKETCH language and compiler.

The original SKETCH language allows the user to write a clean, behavioral specification for an algorithm, and then sketch the outlines of a more efficient implementation. The following example illustrates some of the key features of the language. The function on the left is a specification that computes the logarithm of an integer by searching for the position of the most significant non-zero element. The specification is executable, so it can be debugged and tested like any other program. The function on the right is a sketch. The **implements** clause in the function header states that sklog2 should be resolved to be behaviorally equivalent to slog2. The **??** operators will be replaced by the synthesizer with suitable constants to satisfy this equivalence.

```
int W = 8, logW = 3;
bit[W] log2(bit[W] in) {
  bit[W] i = W;
  while (ret > 0) {
    i--;
    if (in[i])
      break;
  }
  return i;
}
```

```
bit[W] sklog2(bit[W] in)
       implements log2 {
  bit[W] ret = 0;
  loop (logW)
    if (in & ??) {
      in >>= ??;
      ret |= ??;
    }
  return ret;
}
```

The sketch specifies the key properties of the implementation: in particular, we state that it applies a binary search (with a logarithmic number of steps) to find the most significant non-zero, setting the bits of the result accordingly. The **??** operator is being used in place of tedious details of the implementation, such as the actual masks and shifts. The synthesizer will unroll loop constructs the appropriate number of times, inline all function calls (if such exist), and then substitute all "holes" with actual values to produce a final implementation that matches the behavior of the specification for all inputs.

```
bit[W] sklog2_resolved(bit[W] in) {
  bit[W] ret = 0;
  if (in & 0xf0) { in >>= 4; ret |= 4; }
  if (in & 0xc0) { in >>= 2; ret |= 2; }
  if (in & 0x02) { in >>= 1; ret |= 1; }
  return ret;
}
```

The sketch compiler completes the sketch by reducing the problem to a generalized Boolean satisfiability problem. Having unrolled all loops and inlined all function calls, the compiler then replaces each hole in the resulting straight-line program with a distinct free variable, referred to as a *control variable*. Then it translates both the specification and the sketch into Boolean functions of the inputs and the control variables.

At this point, the sketch resolution problem becomes a 2QBF Boolean satisfiability problem (i.e., a satisfiability problem with two quantifiers) of the form

$$\exists c. \forall x. P(x) = S(x, c) \tag{2.1}$$

where P is a Boolean function representing the spec, and S is a Boolean function representing the sketch. The variables x and c correspond to the program inputs and the control variables respectively. The formula specifies a search for a value c of the control variables that will make the sketch behave identically to the specifications on all inputs x.

The problem is then solved using a counterexample-driven search procedure. The solver expects that there will be a small set of inputs E such that a solution c to the problem

$$\exists c. \forall x \in E. P(x) = S(x, c) \tag{2.2}$$

will also be a solution to Equation (2.1). Equation (2.2) can be expanded and supplied to a SAT solver directly since the universal quantification over the small set E can be expressed as a conjunction. The SKETCH solver starts by solving Equation (2.1) with E containing only a single random input. It then verifies that the synthesized c actually solves Equation (2.1). If the verifier fails, it will produce a counterexample which is added to set E, and the process is repeated until a solution is found or the sketch is shown buggy (*i.e.*, it cannot be completed to behave like the specification).

The synthesis algorithm works remarkably well, but requires that both the spec and the sketch be *finite* programs. This means that the number of iterations of all loops and the sizes of all arrays must be bounded at compile time. In the rest of the paper, we describe how we have overcome this limitation for the domain of stencils.

3. Sketching Stencil Kernels

In the scientific computing literature, a stencil is a nearest-neighbor computation on a grid, where the new value of a grid entry is computed as a function of the old values of some of its neighbors. Stencils are generally classified by the number of neighbors they consider and the dimensions of the grid. For example, a typical four-point stencil in two-dimensions computes a value $a^{\text{new}}_{i,j}$ as a linear combination of the values $a^{\text{old}}_{i+1,j}$, $a^{\text{old}}_{i,j+1}$, $a^{\text{old}}_{i-1,j}$ and $a^{\text{old}}_{i,j-1}$. This paper uses a broader definition of stencils: we define a stencil to be a function that computes each element of an output grid by performing constant time operations on a bounded number of input grid elements.

Stencils form the core of many scientific applications; in particular, most PDE solvers work through repeated applications of different stencils. Stencils are also important in signal processing, image analysis and even compression; for example, the wavelet transform that forms the basis of the JPEG2000 image compression standard is implemented as a sequence of stencil computations. Our broader definition of stencils also covers data permutations, such as matrix transpositions, and even data-dependent permutations such as scatter and gather operations.

As befits such an important class of problems, great efforts have been expended in automatically identifying and optimizing stencils, particularly in languages for high-performance computing, such as HPF [18] and ZPL [20]. Fully automatic optimization approaches, however, are constrained by the set of transformations built into the compiler, as well as the analysis and heuristics used to decide if it is possible and convenient to apply a given one. These constraints are relevant for production codes because stencil optimization is still an area of active research [10–12, 19]. For this reason, many production-level stencil codes are still hand-tuned in Fortran and C++.

Hand-optimized stencil implementations can be fiendishly complicated despite their relatively small size. As one of the authors can

```
int N;
int T = 3;  // manipulate 3 rows at once
void sten1d(float[N] in, float[T,N] X) {
  for (int i = 0; i < N; ++i)
    X[0, i] = in[i];
  for (int t = 1; t < T; ++t)
    for (int i = 1; i < N-1; ++i)
      X[t, i] = X[t-1, i-1] + X[t-1, i+1] ;
}
```

Figure 1. Specification for 1-dimensional 2-point stencil.

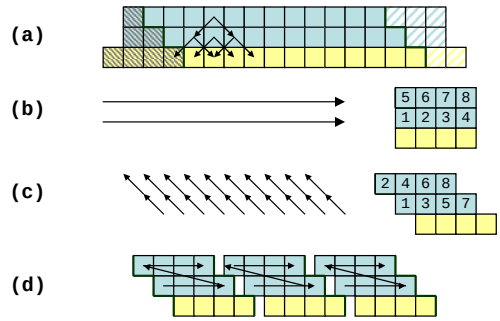

Figure 2. (a) Grid with some of the data dependencies. The regions in the two ends correspond to the corner cases which the time-skewed implementation will have to handle differently. (b) Iteration direction for the spec. (c) Iteration direction for the time-skewed implementation. (d) Iteration direction for the base case of the cache-oblivious scheme.

attest from personal experience, one can easily spend several days hunting for a bug in two hundred lines of this low-level code. The complexity in these implementations arises from a large number of low-level expressions controlling nested looping and multidimensional indexing. These expressions have little intuitive meaning for the programmer because they do not resemble the specification. To make matters worse, programming errors may have subtle effects which are hard to spot. For example, it is possible for iterative algorithms that work by repeated applications of a stencil to produce the correct answer even when coded with a buggy stencil; they algorithm may just take much longer to converge.

For these reasons, the domain of stencils is well suited for sketching: stencil specifications can usually be stated cleanly and concisely, in a few dozen lines of code, and the low-level expressions which complicate the implementations can be efficiently synthesized by the compiler. As the following example illustrates, sketched implementations for stencils allow the developer to provide high-level insights about an implementation without getting bogged down in low-level implementation details.

3.1 Example

The stencil we want to implement is a 2-point 1-dimensional stencil of the form $X_i^t = X_{i-1}^{t-1} + X_{i+1}^{t-1}$. In order to improve temporal locality, the implementation will compute the next two iterations (called timesteps) of the stencil at once, instead of a single one. This is described by the full specification shown in Figure 1. From this specification, we are going to create two implementations, the second one being an improvement of the first one. First, we are going to implement time-skewing, as described in [23]. Then we are going to use this time-skewing implementation as a reference to produce a recursive cache-oblivious implementation like the one described in [10].

The time-skewing transformation requires us to change the iteration pattern from the one in Figure 2(b) to the one in Figure 2(c). Note that in order to preserve the dependencies, we have to traverse along diagonals, which forces us to treat the cells close to the boundary as special cases, as shown in Figure 2(a). It is easy to see that one must write three separate loop nests, two for the corner cases and one for the steady state. However, determining the exact expressions to use for the loop iteration bounds is a challenging task, especially for the corner cases, where the iteration bound for the inner loop will depend on the outer loop. We can express the basic implementation idea in a sketch that leaves all these details unspecified.

To understand the sketch shown below, recall that in SKETCH, functions that contain holes in them are inlined into their call site before the holes are resolved. In effect, we can think of these functions as macros. The inlining semantics allow us to treat functions with holes as *generators*. In the sketch below, `linexpG` is a generator that produces expressions involving sums and differences of its arguments, and the generator `loopNest` can produce loop-nests with arbitrary loop conditions, but which follow the diagonal pattern we desire. In this example, each call to `linexpG` and `loopNest` expands into a separate snippet of code, and the holes in each of these snippets are independent of each other. After the holes are resolved, partial evaluation is applied to clean up the code by eliminating unnecessary operations and conditionals.

```
int linexpG(int a, b, c = 0) {
  rv = ??;
  if (??) rv = (?? ? rv + a : rv - a);
  if (??) rv = (?? ? rv + b : rv - b);
  if (??) rv = (?? ? rv + c : rv - c);
  return rv;
}
void loopNest(float[T,N] X) {
  for (int i = linexpG(N, T); i < linexpG(N, T); ++i)
    for (int t = linexpG(N, T, i);
         t < linexpG(N, T, i); ++t)
      X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
}
void sten1dSK(float[N] in, float[T,N] X)
                    implements sten1d {
  if (N >= 3) {
    for (int i = 0; i < N; ++i) X[0, i] = in[i];
    loopNest(X);  // generate left corner case
    loopNest(X);  // generate steady-state loop
    loopNest(X);  // generate right corner case
  } else
    sten1d(in, X);  // optimization inapplicable
}
```

The key idea of the implementation is expressed in the sketch by stating that $X_{i-t}^t = X_{i-t-1}^{t-1} + X_{i-t+1}^{t-1}$, but the low-level details of the loop iteration bounds are left for the compiler to discover. As an added benefit, the sketch spares the programmer from the error-prone task of having to code the three cases separately. Instead, all three loops are synthesized from the generator `loopNest`.

This sketch resolves to the correct implementation in less than 4 minutes, and produces the code shown below. The synthesized expressions are underlined.

```
void sten1dSK(float[N] in, float[T,N] X)
                    implements sten1d {
  if (N >= 3) {
    for (int i = 0; i < N; ++i)
      X[0, i] = in[i];
```

```
    for (int i = 0; i < T; ++i)
      for (int t = 1; t < i; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
    for (int i = T; i < N; ++i)
      for (int t = 1; t < T; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
    for (int i = N; i < N+T; ++i)
      for (int t = i-N+2; t < T; ++t)
        X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
  } else
    sten1d(in, X);
}
```

This time-skewed implementation will serve to as a reference for a cache-oblivious implementation of the middle loop, which handles the steady state behavior. The first step will be to extract this loop and put it into a separate function as shown below. We will treat this function as a specification.

```
void mainLoop(float[T,N] X, int n1, int n2) {
  for (int i = n1; i < n2; ++i)
    for (int t = 1; t < T; ++t)
      X[t, i-t] = X[t-1, i-t-1] + X[t-1, i-t+1];
}
```

The developer writes another sketch to implement this specification with the recursive partitioning that yields a cache-oblivious behavior. For the base case of the recursion, we are going to reverse the loops, since the block is small enough that we believe we can get better cache behavior this way: it will result in consecutive reads instead of interleaved ones. Since we are not sure how the reversal is going to affect the index expressions, we just replace them with the `linexpG` generator. On the other hand, we wish to retain control of the recursive partitioning and the size of the base case, so that portion of the sketch is fully specified. From the sketch, the synthesizer produces the desired recursive implementation and proves its equivalence to the original spec.

```
void mainLoopSK(float[T,N] X, int n1, int n2)
                        implements mainLoop {
  if (n2-n1 < 4)
    for (int t = 1; t < T; ++t)
      for (int i = n1; i < n2; ++i)
        X[t, i-t] = X[t-1, linexpG(i,t)]
                  + X[t-1, linexpG(i,t)];
  else {
    int m = (n1 + n2) / 2;
    X = mainLoopSK(X, n1, m);
    X = mainLoopSK(X, m, n2);
  }
  return X;
}
```

The cache-oblivious implementation we have produced is known to be difficult to implement by hand, especially for higher dimensions. But using the algorithms presented in this paper, the sketch compiler easily synthesizes the low-level details for this complex implementation. (Note that for a 3-D stencil, we would have to deal with 16 different corner cases, making an optimization by hand extremely difficult [11].) Furthermore, no compiler we know of will generate a recursive cache-oblivious implementation automatically.

4. Overview of Stencil Synthesis

As mentioned before, the compiler resolves the sketches by first reducing the spec and the sketch to finite programs, solving the finite synthesis problem with the SKETCH compiler, and then mapping the results of finite synthesis onto the original sketch. This section presents the key ideas of the reduction process; the reduction algorithm is described in detail in Section 5.

The algorithm exploits domain-specific properties to reduce both the spec and sketch into bounded programs without loss of information. In particular, the reduction uses the fact that each element in the output grid is computed from a finite number of input grid elements. This property allows the compiler to represent the stencil as a function taking only a small number of grid elements as input and producing only a single element of the output. For example, the compiler reduces a 2-point 1-D stencil `s` into a finite program `reduced_s`:

```
float[N] s(float[N] in);
float    reduced_s(float[4] v, int N, int idx);
```

The reduced stencil `reduced_s` computes `s(in)[idx]`, *i.e.*, the value of the `idx`-th element of the grid computed by the original stencil `s`. The finite array `v` represents the elements from the input grid `in` that are needed to compute `s(in)[idx]`. (This section will explain why four elements are needed for a 2-point stencil.) The function is a finite program, and therefore a valid input to the finite SKETCH solver: the size of its input and output are fixed to small constants, and, as we will see shortly, the number of computations performed by the function is constant with respect to the input parameters. In particular, the value of `N` no longer determines the amount of computation; it is only passed because it may be needed to compute the output value, *e.g.*, to identify if the desired output element is close to the grid boundary.

The reduction works in three steps; the same steps are applied to both the spec and the sketch. First, the reduction bounds the size of the program output by focusing on a single element of the output grid. Second, it uses symbolic manipulation to bound the number of computations performed by the stencil. Finally, it uses abstraction to bound the size of the input. To illustrate the process, we use the 2-point 1-D stencil, together with a somewhat contrived sketch which nonetheless hints at how more complex sketches are reduced. For the sake of simplicity, the sketch differs from the specification only slightly: it uses holes to adjust its index expressions to compensate for its loop bounds, which differ from those in the spec. The functions `f`, `g`, and `h` stand for arbitrary index expressions (these functions do not have side effects):

```
float[N] spec(float[N] in) {
  foreach i ∈ [1, N-2]
    out[f(i)] = in[g(i)] + in[h(i)];
}
float[N] sketch(float[N] in) implements spec {
  foreach k ∈ [0, N-??]
    out[f(k+??)] = in[g(k+??)] + in[h(k+??)];
}
```

Bounding the output. We replace the unbounded output grid with a single scalar by making the program return the value of a single element of that output grid. Specifically, a stencil function `s` is transformed to a function `scalar_s` in such a way that `s(in)[idx]=scalar_s(in, idx)`. The transformation does not lose any information, and the behavior of the original program can be obtained by invoking the *scalar function* of `s` as shown below for both the spec and the sketch.

```
float[N] spec(int[N] in) {
  foreach j ∈ [0, N-1] out[j] = scalar_spec(in, j);
}
float[N] sketch(int[N] in) implements spec {
  foreach j ∈ [0, N-1] out[j] = scalar_sketch(in, j);
```

```
}
```

Note that aside from calling a different scalar function, the spec and the sketch are identical (both compute all elements of the output grid out). Hence, we reduced the stencil synthesis problem to the problem of making the scalar functions scalar_sketch and scalar_spec behave identically.

The two scalar functions are shown below. Their goal is to express out[idx] in terms of the input grid. Both functions achieve this with a two-step symbolic back-substitution process. First, they compute the iteration in which the output element out[idx] was most recently assigned. (In our example, each output element was assigned at most once, but as we shall see in the next section this is not required by the algorithm.) The number of this iteration is stored in variable last. Next, the value of out[idx] is computed by evaluating the right-hand-side expression of the most recent assignment. If the right-hand-side expression refers to values other than the input, the two-step process is repeated (not needed in our example).

```
float scalar_spec1(float[N] in, int idx) {
  int last = UNDEFINED;
  foreach i ∈ [1, N-2]
    if (idx == f(i)) last = i;
  if (last == UNDEFINED) return 0;
  return in[g(last)] + in[h(last)];
}
float scalar_sketch1(float[N] in, int idx) {
  int last = UNDEFINED;
  foreach k ∈ [0, N-??]
    if (idx == f(k+??)) last = k;
  if (last == UNDEFINED) return 0;
  return in[g(last+??)] + in[h(last+??)];
}
```

Bounding the number of computations. The scalar functions execute an unbounded number of computations because they contain loops controlled by the free variable N. However, the only purpose of the loops is to identify the latest iteration that wrote to out[idx]. This operation can be expressed declaratively as last = max({[1,N−2]∩f^{-1}(idx)}), where f^{-1} is the inverse of the index expression f.

The advantage of this formulation is that it allows us to use a symbolic solver to reduce the evaluation of the latest iteration expression into a finite sequence of operations. Our current solver is quite rudimentary, but it has already been able to handle all the problems we have tried so far, including several from real-world benchmarks.

For our running example, the declarative formulation allows us to write the scalar functions as follows.

```
float scalar_spec2(float[N] in, int idx) {
  int last = max({[1,N−2] ∩ f^-1(idx)});
  if (last == UNDEFINED) return 0;
  return in[g(last)] + in[h(last)];
}
float scalar_sketch2(float[N] in, int idx) {
  int last = max({k | k ∈ [0,N−??]
                    ∧ k+?? ∈ f^-1(idx)});
  if (last == UNDEFINED) return 0;
  return in[g(last+??)] + in[h(last+??)];
}
```

If f is the identity function, for example, the symbolic solver will replace the last-iteration computation with a few conditional assignments; the one in the spec is shown below.

```
if (idx >= 1 && idx <= N-2) last = idx;
else last = UNDEFINED;
```

After the replacement, the functions are already bounded in terms of computation, but the input is still an unbounded array.

Bounding the input. For programs that do not modify their inputs, the input array can be treated as an uninterpreted function. In other words, the input array is an entity whose only discernible property is that accesses with the same index produce the same value. Programs like the example in Section 3.1 that do modify their input array are modeled by the compiler as receiving an immutable array as input, copying it into a mutable array, and then returning the modified array as an output at the end of the computation.

The next problem is to represent the uninterpreted function finitely. We observe that the scalar functions scalar_spec2 and scalar_sketch2 read only a finite number of input grid elements, which lets us borrow a technique originally proposed by Ackerman [1] and used extensively in hardware verification [4]. The function in_fn below implements the semantics of an uninterpreted function under the restriction that the function is called with at most four different values of the argument. (For this example, we can restrict ourselves to four symbolic values because the scalar functions reduced_spec and reduced_sketch, shown below, will together make no more than four dynamic calls to in_fn for a given value of their input parameter idx.) The function in_fn implements the desired semantics by returning the same symbolic value whenever called with the same value of the argument in_idx.

```
float in_fn(int in_idx) {
  if (in_idx == i_0) return v_0;
  if (in_idx == i_1) return v_1;
  if (in_idx == i_2) return v_2;
  if (in_idx == i_3) return v_3;
}
```

This use of uninterpreted functions is considered a form of abstraction, because we are representing an unbounded grid using only four scalars by throwing away all the information about those cells that were not accessed on a particular invocation of the scalar functions.

The final step is to represent the symbolic values i_k and v_k with constructs from the SKETCH language. (Recall that SKETCH relies on a Boolean satisfiability solver which does not support symbolic manipulations.) The non-symbolic version of in_fn shown below, expressed entirely in SKETCH, implements the symbolic values i_k by remembering the concrete values of arguments in a global array. The symbolic values v_k are implemented as function arguments. This treatment will make the synthesizer carry out its reasoning under all possible values of v_k, which is equivalent to viewing v_k as symbolic values.

```
float in_fn(int in_idx, float[4] v) {
  static int[4] g; static int gi=0;  // globals
  g[gi++] = in_idx;
  if (in_idx == g[0]) return v[0];
  if (in_idx == g[1]) return v[1];
  if (in_idx == g[2]) return v[2];
  if (in_idx == g[3]) return v[3];
}
```

As an optional optimization, our system allows the user to assert that the sketch computes out[idx] using only the input elements used by the spec, as opposed to arbitrary input elements. That is, the array g is set only in the spec; the sketch only reads its values and asserts that one of them matches. This is the case for almost all implementations, because if a sketch used any other entry, its value would have to get canceled out in order for the

spec and the sketch to be equivalent. Using this assumption, we can reduce the number of comparisons on each call to the uninterpreted function, since we only need to compare the index to those indices used by the spec, but not with those used by the sketch. With this optimization, the compiler will ignore any implementation that violates the assumption, while never producing an incorrect implementation.

Putting it all together. Let us assume for the sake of simplicity that `f`, `g`, and `h` are identity functions. Then, the symbolic solver reduces the evaluation of the latest iteration expression to guarded assignments as shown below.

```
float reduced_spec(float[4] v, int N, int idx) {
  if (idx < 1 || idx > N-2) return 0;
  return in_fn(idx, v) + in_fn(idx, v);
}
float reduced_sketch(float v[4], int N, int idx) {
  int last = idx - ??1;
  if (last < 0 || last > N-??2) return 0;
  return in_fn(last+??3, v) + in_fn(last+??4, v);
}
```

These functions define a finite sketch problem which can be solved by the finite SKETCH solver. In this case, the SKETCH solver can easily prove that the abstracted sketch and spec are equivalent for the following value assignments to holes: $??_2 = 3$, $??_1 = ??_3 = ??_4 = 1$.

The control values can then be applied to the original sketch, and the construction will guarantee that the resulting implementation is equivalent to the original spec.

Abstracting integers and floating point values. Here, we focus on complications arising from modeling integer and floating-point values.

The reduced functions produced by the above algorithm lead to intractable SAT problems if translated directly to Boolean circuits, mainly due to the presence of floating point variables in the reduced programs. Additionally, modeling floating point values in their full IEEE glory would make the equivalence criterion overly strict, ruling out many optimizations employed by programmers who often choose to assume associativity of floating point numbers.

There are numerous approaches in the literature to handle floating point arithmetic in the context of model-checking and verification, most of them relying on uninterpreted functions [6, 15]. While it is relatively easy to replace floating point operations with uninterpreted functions, there is a simpler approach that works remarkably well in our domain. The key observation is that the stencils we are generating are often linear functions in their floating point arguments: both the reduced spec and sketch have the property that if you set their integer inputs to any fixed value, the remaining function will be a linear function (it may be a different linear function for different values of the integer inputs). The stencil appears linear to the solver because it performs verification separately for each combination of integer input values. It is trivial to verify this property statically from a DAG representation of the reduced spec and reduced sketch. When this property holds—as is the case with all the benchmarks presented in this paper—the compiler can safely replace all floating point inputs with 1-bit integers without losing soundness, provided that it grows the integer representation whenever necessary to avoid arithmetic overflow. The soundness argument should be obvious from the fact that in order to test the equivalence of two k-dimensional linear functions over the reals, one only needs to test them with k independent vectors.

Floating point constants are treated as free variables, and are also represented with a single bit. This treatment is sound but not complete because it loses algebraic relationships among constants. For example, after we replace 0.5 with a free variable $v_{0.5}$, we can no longer prove that `a*0.5 + a*0.5 == a`. This limitation did not prove to be an issue for any of the benchmarks we studied.

After performing abstraction on the floating point values, the remaining problems involve only integers and Booleans. Our current approach is to translate these problems directly into circuits, which limits our scalability to representing integers with about 6 bits (3 bits for the hardest benchmarks). However, there are known scalable techniques that we can use to remove this limitation [5].

5. Algorithm Details

Here, we presents in detail the stencil reduction algorithm outlined in Section 4, focusing mainly on the first two steps (bounding the output and bounding the computations).

5.1 Preliminaries

We use standard compiler transformations to bring the input programs (*i.e.*, the specification and the sketch) into an intermediate normal form bearing the following properties:

Loops. All loops are normalized to the form `for (int i = e1; i < e2; ++i)`, where `i` is the uniquely named *main induction variable* of the loop. Remaining loop induction variables are expressed as a function of `i`.

Function calls. All calls are inlined. Recursive calls to sketched functions are replaced with calls to their specifications (which must be non-recursive). Proving equivalence of the spec and a recursive sketch after this transformation constitutes a proof by induction, under the assumption that the recursion is well founded (*i.e.*, that all recursive calls eventually terminate).

Normalized programs obey the following syntax:

Expressions	e	$::= n \mid \texttt{true} \mid \texttt{false} \mid \texttt{x} \mid \texttt{x[}e\texttt{]} \mid e_1\ \texttt{op}\ e_2$
		$\mid f(e_1, \ldots, e_k)$
		$\mid \texttt{switch}\ e\ \texttt{case}\ n_1\texttt{:}\ e_1\texttt{;} \ldots \texttt{case}\ n_k\texttt{:}\ e_k$
Statements	c	$::= \texttt{x:=}e \mid \texttt{x[}e_1\texttt{]:=}e_2 \mid \texttt{skip}$
		$\mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid c_1\texttt{;}\ c_2$
		$\mid \texttt{for (i =}\ e_1\texttt{; i <}\ e_2\texttt{; ++i)}\ c$
Functions	f	$::= \texttt{def f(x}_1\texttt{, } \ldots \texttt{,x}_k\texttt{)}\ c\ \texttt{return}\ e$

An additional semantic restriction is that loop bounds must be invariant with respect to the induction variable of their loop. The bounds can, of course, relate to outer induction variables. This restriction could be relaxed, but the complications involved are not justified in the domain of stencils. The domain is also restricted by the power of the algebraic solver used to eliminate the latest assignment expression (called `RD` in this section).

If the program satisfies the aforementioned constraints, the reduction about to be described will be sound, but only under the assumption that there is no integer overflow in either the spec or the sketch. This is because the symbolic solver used to eliminate the latest assignment expression may assume algebraic properties that do not hold in the presence of overflow (*e.g.*, `a-1<N` $\iff$ `a<N+1`).

5.2 Synthesizing Scalar Functions

We describe the algorithm that generates scalar functions and bounds their computation. These two steps are performed in an intertwined fashion and we describe them together.

One way to define a scalar function is to view it as a slice of the original stencil. More precisely, given a stencil computation `s(in)` returning a grid `out`, and an index `idx`, the scalar function computes a slice of `s` with respect to `out[idx]`. While the original computation may read the entire (unbounded) input grid, the slice only reads a *bounded* number of input elements.

The slice is expressed recursively, using a functional language that resembles the one defined by the above syntax, but does not

include statements. Each recursive call corresponds to a def-use edge in the slice of `out[idx]`. Since the slice incurs a bounded computation, the recursion is bounded. The result is thus finite and can be accepted by the finite SKETCH synthesizer.[1].

The slicing algorithm boils down to expressing `out[idx]` symbolically in terms of program inputs. To this end, we recursively substitute non-input variables in the expression `out[idx]` with the right-hand side values of their most recent assignments.

We refer to most recent assignments as *reaching definitions*. In contrast with traditional reaching definitions, which offer a static approximation of the dynamic behavior of the program, our reaching definitions are *concrete*: they are defined on the execution trace where each program point is reached by at most one definition for any program location. Since there is no ambiguity as to which definition most recently assigned the location, we obtain precise back-substitution in the sense that the fully substituted symbolic expression is executable and computes the value of `out[idx]`.

The procedure `RD`, which lies at the heart of the algorithm, computes the concrete reaching definition of a memory location.

$$RD : M \times P \times I \rightarrow P$$

The procedure maps a memory location $m \in M$, an execution point $p \in P$, and a program input $i \in I$ to the most recent execution point that defined the value of m prior to p, under the input i. The set of memory locations M consists of scalar variables and array elements. The set of execution points P is the cross product of static program points with the loop iteration space. The input space I includes the input grid together with any scalar arguments.

We are now ready to describe the abstraction algorithm. For each scalar variable `v` we create a function

$$\texttt{v\_fn} : P \times I \rightarrow \texttt{T}$$

that computes the value n of `v` at execution point $p \in P$ under the input $i \in I$. The type `T` is a primitive type (**`boolean`**, **`int`**, or **`double`**). The function will be expressed in the functional language shown above. Similarly, for an array `a` (for simplicity, we assume that arrays are one-dimensional) we create a function

$$\texttt{a\_fn} : \texttt{Int} \times P \times I \rightarrow \texttt{T}$$

that computes the value of `a[idx]` for an index `idx` $\in$ `Int` at point $p \in P$ under the input $i \in I$. These two functions are called v-functions. The reduced function for a stencil `s` returning a grid `out` now becomes

```
double reduced_s(int idx, double[N] in) {
  return out_fn(idx, P_e, in);
}
```

where P_e is the end point of the program execution trace.

v-functions are constructed via syntactic translation of the original program. A v-function first obtains the reaching definition and then (recursively) replaces all array and variable references on the right-hand side of the reaching definition with calls to appropriate v-functions. A v-function for variable `v` (or array access `a[idx]`) at execution point p under input i, looks as follows.

1. Obtain the most recent execution point p' where `v` (respectively, `a[idx]`) was defined prior to p under input i.
2. Extract the static program statement s executed at p', and the right-hand side expression e in s.
3. Return a valuation of a transformed expression $F(e)$ where
   (a) each variable sub-expression `v'` is replaced with `v'_fn`(p', i);
   (b) each array access sub-expression `a'[`e'`]` is replaced with `a'_fn(`$F(e')$`,`p'`,`i`)`.

To illustrate the process, consider the following example. The example is acyclic so that the reader need not be concerned with execution point representation for now.

```
     int[N] f(int[N] in, int a) {
s1:    int[N] out = 0;
s2:    int[N] A = in;
s3:    if (in[3] > in[4]) {
s4:      out[3] = A[3];
s5:      A[a] = in[5];
       }
s6:    if (A[5] > 0)
s7:      out[a] = A[out[a]];
       return out;
     }
```

The v-function of `out` for this example needs to handle three assignments to `out`:

```
out_fn(idx, p, i)) {
  p' = RD((out,idx), p, i);
  switch (P_s(p')) {  // extract the statement at p'
    case s1: return 0;
    case s4: return A_fn(3, p', i);
    case s7: return A_fn(out_fn(a, p', i), p', i);
  }
}
```

It remains to show how the function `RD` computes the concrete reaching definitions. First, we need to define the *execution point* $p \in P$. As mentioned in passing above, an execution point p is a pair (s, t), where s is a (static instance of a) program statement and t is a point from the iteration space T of the program. The iteration point t is defined as a valuation of loop induction variables that are in scope at the statement s (these are exactly the induction variables of loops that enclose s). When $p = (s', t')$, we define `P_s`$(p) = s'$ and `P_t`$(p) = t'$. In the following, we use `T_map(`t`,'j',`n`)` to denote binding of an induction variable `j` to some value n in iteration point t, and `T_get(`t`,'j')` to extract the currently bound value for `j`. Similarly, `I_get(`i`,'x')` extracts the value associated with (non-induction) variablex at input state i.

The trace of a program execution is a sequence of execution points. We define a total order $<_P$ on P such that $p_1 <_P p_2$ iff p_1 executed before p_2. The *execution order* $p_1 <_P p_2$ is determined by the lexicographic order of the iteration points `P_t`(p_1) and `P_t`(p_2); if there is a tie, then p_1 and p_2 must be in the same loop iteration and their execution order is determined by their position in the program. Internally, we represent the execution point such that the $<_P$-test can be performed as a single lexicographic test. We define two execution point constants: P_b is the beginning of the execution and P_e is the end of the execution.

As statement guarded by conditionals may not execute in every iteration, the execution order $<_P$ alone is insufficient for determining the most recent definition. To reflect control conditions under which the statement executes, we define the predicate $q(p, i)$, which holds iff the statement `P_s`(p) executes at iteration point `P_t`(p) under the input i. Formally speaking, $q(p, i)$ is the disjunction of the path constraints for all paths that reach the execution

[1] If the input program violates the boundedness assumption, the algorithm described in this section will still produce a correct recursive representation of the slice, but the recursion will be unbounded and will depend on the inputs. When this program is fed to the SKETCH synthesizer, the synthesizer will attempt to inline the recursive calls an increasing number of times, to no avail; after a few tries, it will reach a predefined threshold for function inlining, and will inform the user that the sketch can not be resolved. Crucially, though, a violation of the assumption can not lead to a buggy implementation.

point $q(p, i)$. Programs in our domain have structured control flow and loop bounds that are invariant with respect to their loop's induction variable. Therefore, the predicate $q(p, i)$ can be constructed syntactically as a conjunction of all the conditionals (including loop conditions) enclosing the statement P_s(p).

For example, consider statement s_4 in the code below. Let $p = (s_4, t)$ be an execution point associated with s_4 for some iteration point t. We form the predicate $q(p, i)$ for this execution point under some input state i: the constraint corresponding to the **if** statement in s_3 is A_fn(T_get(t, 'j'), p, i) > 0; the constraint corresponding to the loop statement in s_0 and s_1 is T_get(t, 'j') $\geq 0 \wedge$ T_get(t, 'j') $<$ I_get(i, 'N').

```
    double[N] f(double[N] in) {
      int [N] A, out;
s0: for (int j=0;
s1:              j<N; ++j) {
s2:   A[j] = in[j];
s3:   if (A[j] > 0)
s4:     out[j] = in[j];
      else
s5:     out[j] = -in[j];

s6:   out[j] = sqrt(out[j]);
    }
  }
```

We are now ready to describe RD, the procedure for computing reaching definitions, for this case of an array access. Given a program location v[idx], an execution point p, and input i, the procedure considers all execution points p' that precede p, execute under the input i, and assign into location v[e] for some index expression e such that the value of e at execution point p' equals idx. Among all such execution points, it selects the most recent one; if none meets all criteria, there is no reaching definition, and RD returns P_b.

```
RD((v,idx), p, i) {
  return max({P_b} ∪ {p' | p' <_P p ∧ q(p',i)
                         ∧ P_s(p')≡v[e]=e'
                         ∧ e_fn(p',i)=idx});
}
```

Our compiler uses algebraic reasoning to simplify procedure RD. The symbolic simplifier reasons with equalities, inequalities, and logical connectives. The simplification procedure relies on the fact that when we have an assignment of the form x[g(j)] = e, the constraint g(j) = idx often suffices to fully define the iteration space point in terms of idx, by inverting g. It then remains only to test whether the values thus derived satisfy the remaining constraints for qualifying execution points. Also, finding the most recent point of assignment is done in a staged manner, by first finding the most recent point corresponding to *each* assigning statement and then picking the most recent among them.

In the above example, we can find the last program point prior to point p where out[idx] has been updated by the particular statement s_4 (if such a point exists), as follows:

```
p' = max({(s4,t) | (s4,t) <_P p
                 ∧ (T_get(t,'j')≥ 0
                    ∧ T_get(t,'j')<I_get(i,'N')
                    ∧ A_fn(T_get(t,'j'), (s3,t), i)> 0)
                 ∧ T_get(t,'j')=idx});
```

Note that the constraint T_get(t,'j') = idx fully defines the value of j to be equal to idx, so the statement above can be replaced with a couple of simple assignments.

```
t = T_map(new T,'j',idx);
```

```
int out_fn(int idx, P p, I i)  {
  T t = T_map(new T, 'j', idx);

  P p4 = new P(s4,t);
  if (! (p4 < p && idx >= 0 && idx < I_get(i,'N')
         && A_fn(idx, new P(s3,t), i) > 0))
    p4 = P_b;
  P p5 = new P(s5,t);
  if (! (p5 < p && idx >= 0 && idx < I_get(i,'N')
         && ! A_fn(idx, new P(s3,t), i) > 0))
    p5 = P_b;
  P p6 = new P(s6,t);
  if (! (p6 < p && idx >= 0 && idx < I_get(i,'N')))
    p6 = P_b;

  P p' = (p4 < p5 ?  (p5 < p6 ? p6 : p5) :
          (p4 < p6 ? p6 : p4));
  switch (P_s(p')) {
    case s4: return I_get(i, 'in', T_get(P_t(p'),'j'));
    case s5: return -I_get(i, 'in', T_get(P_t(p'),'j'));
    case s6:
      return sqrt(out_fn(T_get(P_t(p'),'j'), p', i));
  }
}
```

Figure 3. v-function for array out

```
p' = (s7,t);
if (! (p' < p && idx >= 0 && idx < I_get(i,'N')
       && A_fn(idx, (s3,t), i) > 0))
  p' = P_b;
```

To complete our example, we find statement-specific most recent points for each assigning statement in a similar manner, and pick the most recent among these points. The final v-function for out is shown in Figure 3.

6. Evaluation

In this section, we present an empirical evaluation of our system using several kernels from the MultiGrid method as case studies. From the case studies, we were able to validate three basic claims.

Scalability. We prove that the system scales to complex real-world implementations of important kernels. For example, we were able to synthesize in a matter of minutes an implementation for a kernel that involved 14 different loops from a sketch that had 44 different holes.

Usability. The case studies also allow us to describe a typical use scenario for a sketching system. In particular, we describe how we were able to explore different implementation strategies, discarding those that don't work and refining those that do, without the risk of introducing bugs.

Performance of generated code. We show that creating these complex implementations is worth the effort. In particular, one of the implementations we sketched was over 8 times faster than the original reference implementation on a 1.3 GHz Itanium-2, even though they were compiled and optimized using the Intel Fortran compiler version 9.1, arguably one of the best compilers commercially available on the Itanium architecture. In other words, the sketch expressed optimization ideas which the compiler was unable to discover on its own.

The results of the performance evaluation of the synthesizer are summarized in Table 1. The table lists all the benchmarks

	Loop	Holes	i-bits	Size	SAT	Total
timeSkew	3	12 expr	5	10469	137	269
cache0bv	rec	2 expr	5	5313	141	165
interp1	3	111	3	1930	422	424
interp2	7	74	3	1954	85	88
rb3d1	6	36	5	2471	97	108
rb3d10dd	14	43	3	27045	138	214
rb3d2	7	30	3	18994	82	145
rb2d1	4	16	6	1809	81	88
rb2d2	4	22	6	3868	72	78

Table 1. Solution time for the different benchmarks in the paper. Table columns specify: the number of loops present in the final implementation (rec stands for recursive); the number of holes the synthesizer filled in; the number of bits used to represent integers; the number of Boolean and arithmetic operations in the reduced problem after some simplification; the time (in seconds) spent in SAT solver queries; and the total time (in seconds) required to resolve the benchmark.

that appear in the paper together with their solution time, and a few statistics regarding their complexity. The statistics include the loops in the final implementation, the number of holes filled by the synthesizer, and the number of integer and boolean operations in the spec and the sketch after reduction and some algebraic simplification. This last quantity is simply to give a measure of how complex the reduced problems are. The table also shows the solution time for each benchmark on an Intel Pentium M 1.6 GHz laptop, and what fraction of the time is spent solving SAT problems. It should be pointed out that SKETCH doesn't invoke a SAT solver directly, but instead uses the circuit equivalence checking engine from ABC [14], which has proven to be much more efficient.

6.1 Sketching for MultiGrid

The MultiGrid algorithm is used for solving partial differential equations for a wide range of domains, including fluid dynamics and solid mechanics. It is composed of three main kernels: relax, interpolate, and restrict [3]. Each application of the relaxation routine produces a closer approximation to the solution, but with a very fine grid the low frequency components of the error in the approximation take too long to die out. To address this problem, MultiGrid computes corrections to the solution by creating a coarser problem (restrict), solving it recursively, and then mapping the correction back to a finer grid (interpolate). We have sketched implementations of relaxation and interpolation kernels in 3D; the restrict kernel is quite similar to interpolate.

We sketched several implementation tricks from the literature and from hand optimized implementations of these kernels. In a couple of cases, it took less than half an hour to write and synthesize implementations that we estimate would have taken half a day if written by hand. Additionally, some of our sketched implementations were several times faster than the clean reference implementations, even after the latter had been optimized by the Intel Fortran compiler.

Relaxation. The relaxation phase of MultiGrid starts with an approximation to the solution of the problem and uses it to compute a closer approximation to the solution. For our case study, we implemented a Red-Black Gauss-Seidel relaxation scheme for both a 2D grid and a 3D grid following more or less the same implementation strategies. The specification for the 3D benchmark is shown below. The algorithm assigns a color to each cell on the grid in a checkered pattern, and then applies a six point stencil on the red cells, followed by another (same) stencil on the black cells. This widely used relaxation scheme has well known implementation strategies to optimize it for different architectures [8].

```
// red
for (int i = 1; i < N-1; ++i)
  for (int j = 1; j < N-1; ++j)
    for (int k = 1; k < N-1; ++k)
      if (i%2 == 0 ^ j%2 == 0 ^ k%2 == 0)
        out[i,j,k] = F(i,j,k, in);
// black
for (int i = 1; i < N-1; ++i)
  for (int j = 1; j < N-1; ++j)
    for (int k = 1; k < N-1; ++k)
      if (! (i%2 == 0 ^ j%2 == 0 ^ k%2 == 0))
        out[i,j] = F(i,j,k, out);
```

Here, `F(i,j,k, prev)` expands to

```
f[i,j,k] + v0*in[i,j,k] + v1*prev[i-1,j,k]
     + v2*prev[i+1,j,k] + v3*prev[i,j-1,k]
     + v4*prev[i,j+1,k] + v5*prev[i,j,k+1]
     + v6*prev[i,j,k-1];
```

The above specification is written in the simplest possible way, using the xor expression `i%2 == 0 ^ j%2 == 0 ^ k%2 == 0` to decide the color of each cell. These types of conditions tend to confuse traditional dependence analysis, so even a state-of-the-art compiler like the Intel compiler is unable to optimize the kernel fully.

In order to produce a better implementation, we used two implementation strategies from [8]. In this paper, Douglas *et al.* provide only high-level descriptions of their optimization strategies (no pseudo-code), but those low-level details omitted in the paper are exactly what SKETCH can synthesize.

The first implementation we created is quite simple; it just eliminates the conditionals by computing the output in blocks of eight elements at a time: four red and four black. The sketch for the case where N is even is very simple; it has two loop-nests with unspecified bounds, each with four assignments of the form

```
out[2*i-??,2*j-??,2*k-??] =
    F(2*i-??,2*j-??,2*k-??, in);
```

The sketch describes the high-level idea that we compute first all the red cells four at a time, and then all the black cells, also four at a time. The sketches `rb2d1` and `rb3d1` from Table 1 correspond to the 2D and 3D instances of this sketch respectively. One can see that both sketches resolved quite fast despite having a large number of holes. Moreover, the resulting implementation is about 45% faster for the 3D case and 70% faster for the 2D case.

The implementation for the case where `N` is odd is considerably more complicated because one must cover a lot of corner cases, particularly in 3D, where the final implementation is composed of 14 different loops. However, with the support of sketching, it is easy to construct the odd case from the even case. To do this, we took the 3D implementation for the even case produced by the previous sketch, and simply sketched the corner cases on top of it. Using the implementation generated from the even case as a starting point allowed the sketch to scale; a sketch for the odd case that left everything unspecified proved to be intractable for the compiler. Fortunately, we did not have to start from scratch. We were able to exploit the fact that we already had a solution for the even case to make the odd case more tractable. The sketch for the red cells for the 2D odd case is shown below. Statements 1 and 2 came from the implementation of the even case, and on top of it, we added a corner case for the last cell in each row (3), and the last row (4), Since we are not sure if the last cell in the last row (5) needs to be treated separately, we ask the solver to decide.

```
for (int i = ??; i < N/2-??; ++i){
  for (int j = ??; j < N/2-??; ++j){
    out[2*i-1,2*j-1] = F(2*i-1,2*j-1, in); //1
    out[2*i,2*j] = F(2*i,2*j, in);         //2
  }
  out[2*i-??,N-??] = F(2*i-??,N-??, in);   //3
}
for (int j = ??; j < N/2-??; ++j){
  out[N-??,2*j-??] = F(N-??,2*j-??, in);   //4
}
if (??) out[N-??,N-??] = F(N-??,N-??, in); //5
```

Our second implementation for this benchmark uses another strategy mentioned in [8], namely computing the red and black cells together in a single pass through the array. In this case, careful attention is required in order to preserve the dependencies. We first implemented the trick in 2D by creating a loop that updates both red and black cells as shown below, and then two more loops to handle the corner cases.

```
for (int i = ??; i < N/2-??; ++i) {
  for (int j = ??; j < N/2-??; ++j) {
    // red
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, in);
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, in);
    //black
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, out);
    out[2*i-??,2*j-??] = F(2*i-??,2*j-??, out);
  }
}
```

From the sketch, the synthesizer was able to discover that it had to compute the black cells with an offset with respect to the red cells in order to preserve dependencies. Note that neither the loop bounds nor the array access offsets are trivial; getting them right would have been quite challenging for the programmer.

```
for (int i = 2; i < N/2; ++i){
  for (int j = 1; j < N/2; ++j) {
    // red
    out[2*i-1,2*j-1] = F(2*i-1,2*j-1, in);
    out[2*i,2*j] = F(2*i,2*j, in);
    //black
    out[2*i-3,2*j-0] = F(2*i-3,2*j, out);
    out[2*i-2,2*j-1] = F(2*i-2,2*j-1, out);
  }
}
```

As shown in Table 2, this optimization delivered a 60% performance improvement compared to the previous optimization, which was already 70% faster than the spec. This shows that the optimizations we are able to sketch go beyond what the Intel compiler is able to do on its own given a straightforward specification.

We tried to implement the same trick in 3D. In this case, our sketch produced an implementation that computed the black cells for the plane $2i$ after computing the red cells in the plane $2i + 2$. Unlike the 2D case, however, this produced a performance degradation, probably due to the fact that the planes are too big to fit in the cache, so accessing too many of them at a time simply confuses the prefetcher with no benefit to performance.

Interpolation. The interpolation routine maps the values in a coarse grid to a finer grid of size $2N \times 2N \times 2N$, as illustrated in Figure 4. Points in the fine grid that correspond to points in the coarse one are copied, while the other points in the fine grid are computed by averaging the values of their neighbors in the coarse grid. As illustrated in Figure 4, this leads to four different cases, depending on whether we average 1, 2, 4 or 8 points. The following

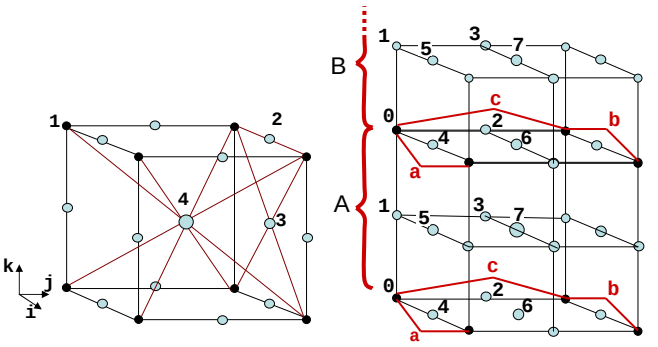

Figure 4. (a) Stencil for interpolation distinguishes four different cases. Either the new point matches a point in the coarse grid (1), is in an edge in the old grid (2), in a face (3), or in the center of a cube formed by consecutive points in the old grid (4). (b) The optimized version will precompute the sums a, b and c.

code shows a fragment of the specification, describing a few of the cases.

```
for (int i = 0; i < 2*N-2; ++i)
 for (int j = 0; j < 2*N-2; ++j)
  for (int k = 0; k < 2*N-2; ++k) {
   if (i%2 == 0 && j%2 == 0 && k%2 == 0)  // Case 1
     out[i,j,k] = in[i/2,j/2,k/2];

   if (i%2 == 1 && j%2 == 0 && k%2 == 0)  // Case 2
     out[i,j,k] =
       0.5 * (in[i/2,j/2,k/2] + in[i/2+1,j/2,k/2]);
   ...

   if (i%2 == 1 && j%2 == 1 && k%2 == 0)  // Case 3
     out[i,j,k] =
       0.25 * (in[i/2,j/2,k/2] + in[i/2+1,j/2+1,k/2]
       + in[i/2,j/2+1,k/2] + in[i/2+1,j/2,k/2]);
   ...
  }
```

As in the previous sketch, we started by blocking the computation to eliminate all the conditionals in the specification. For each point in the coarse grid, there is a $2 \times 2 \times 2$ block in the fine grid which constitutes the smallest repeating pattern. Because the output grid is of size $(2N)^3$, odd grid sizes are not a problem.

The sketch was very easy to write because we left every single array offset unspecified, as well as the bounds of all loops. We only specified that on each iteration of the innermost loop, there were 8 assignments to entries of `out`, 1 for case 1, 3 for case 2, 3 for case 3, and 1 for case 4. The individual cases are shown below.

```
out[2*i+??,2*j+??,2*k+??] = in[i+??,j+??,k+??];
out[2*i+??,2*j+??,2*k+??] =
  0.5 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
out[2*i+??,2*j+??,2*k+??] =
  0.25 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
          in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
out[2*i+??,2*j+??,2*k+??] =
  0.125 * (in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??] +
           in[i+??,j+??,k+??] + in[i+??,j+??,k+??]);
```

As shown in Table 2, this simple transformation allowed the implementation to run 8 times faster than the spec.

The second sketch we did for this benchmark describes an optimization which is used by the HPF implementation of this kernel in the NAS benchmark suite [2]. The key insight behind this optimization is that a lot of the sums are computed more than once, so we can reuse some of them when computing different blocks. Figure 4(b) shows two consecutive blocks (A and B). The pairs `a`, `b` and `c` represent a partial sum of two points from the original grid. We can see that the partial sum `a` can be used to compute 6 different points: `A5`, `A7`, `B4`, `B5`, `B6`, `B7`. Similarly, the partial sums `b` and `c` can be reused in computing most of the other points. The implementation uses this insight by pre-computing the partial sums `a`, `a+b` and `c`; these are stored in temporary arrays for each value of $(\mathtt{i}, \mathtt{j})$, to make the rest of the computation easier to vectorize.

We wrote the sketch for this implementation trick in less than one hour. As before, the sketch leaves unspecified every single array offset and every single loop iteration bound. Below, one can see the loop that pre-computes the sub-expressions:

```
for (int i = ??; i < N-??; ++i) {
  float ta = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  float tb = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  float tc = in[i+??,j+??,k+??] + in[i+??,j+??,k+??];
  aplusb[i] = ta + tb;
  a[i] = ta;
  c[i] = tc;
}
```

The code for each of the expressions was sketched following Figure 4(b). In particular, the picture shows that the three points corresponding to case 2 are computed one from `a`, one from `c`, and one by adding the two vertices labeled `0`. Similarly, for points corresponding to case 3, one is computed from two entries from `c`, one from two entries from `a`, and one is just `a+b`. And finally, case 4 is the sum of two consecutive `a+b`. So the basic idea is clear from the picture, and can be sketched directly with the statements shown below. On the other hand, the details of which offsets to use, and what the loop bounds should be are not clear from the picture, so those are left unspecified for the solver to resolve.

```
output[k*2+??,j*2+??,i*2+??] = in[k+??,j+??,i+??];
output[k*2+??,j*2+??,i*2+??] =
  0.5 * (in[k+??,j+??,i+??] + in[k+??,j+??,i+??]);
output[k*2+??,j*2+??,i*2+??] = 0.5 * a[k+??];
output[k*2+??,j*2+??,i*2+??] = 0.5 * c[k+??];
output[k*2+??,j*2+??,i*2+??] =
  0.25 * (c[k+??] + c[k+??]);
output[k*2+??,j*2+??,i*2+??] =
  0.25 * (a[k+??] + a[k+??]);
output[k*2+??,j*2+??,i*2+??] = 0.25 * aplusb[k+??];
output[k*2+??,j*2+??,i*2+??] =
  0.125 * (aplusb[k+??] + aplusb[k+??]);
```

The sketch resolves in less than three minutes.

On the Itanium-2, this optimization had a very minimal impact on the performance compared with simple blocking. However, what is important is the fact that we were able to sketch the implementation trick, and get a complete implementation for it, all without the risk of introducing bugs. In fact, in the process of sketching these optimizations, we tried many other variations on the basic tricks. Some ideas were rejected by the compiler while others caused performance degradations. Nevertheless, we were able to try them easily and without introducing bugs.

7. Related Work

The algorithms described in this paper are implemented as an extension to the SKETCH system, and rely heavily on the combinatorial synthesis engine described in [22]. The original SKETCH compiler handled unbounded programs by artificially bounding the size of their input. This is completely unsuitable for stencils, because the resulting formula would be proportional (by a very large factor) to the maximum size of the input grids, leading to intractably large problems: even the simplest sketch examined in this paper (`rb2d1` from Section 6) causes the original SKETCH compiler to run out of memory when asked to consider grids of size $N < 16$. For the same size problem (3 bit integers), the new algorithm resolves the same sketch in 7 seconds.

The reduction technique presented in this paper extracts a bounded representation of stencils, making the problem tractable for the combinatorial synthesizer. Thus it extends the domain of the SKETCH system by allowing it to reason about unbounded programs. Additional techniques—like those presented in [5]—can be applied on top of the bounded representation in order to prove equivalence very efficiently.

Our reduction technique is based on exploiting the high-level structure of a program, together with abstraction to make a combinatorial analysis more tractable. These principles are well-rooted in the research of model checking: for example, uninterpreted functions and their representation using input variables is widely used in verifying hardware and (more recently) software [4, 6], as is the idea of breaking the problem into cases that we apply to bounding the output [13].

7.1 Alternative approaches

We find our approach to be complementary to more traditional forms of compiler optimization. On the one hand, having a very powerful optimizer available allows sketching-based efforts to focus only on higher-level optimization. On the other hand, sketches can be used to express implementations that are beyond the capabilities of traditional compiler optimization. Optimizers relying on dependence analysis—such as [16]—must be able to reason statically about all array reads and writes, so every array index expression must have a clean algebraic form to ensure that no dependence will be violated after a transformation. Additionally, dependence analysis must deal very conservatively with index expressions and conditionals that depend on inputs.

In contrast to dependence analysis, our reduction procedure only needs to reason about array index expressions on the left-hand side of an assignment, since these are used to symbolically derive the expression for the latest assignment to an array (see Section 5). Therefore, complex conditionals, and even input dependent array accesses and looping structures, do not pose further complication; the reduced problem is delegated to the combinatorial engine, which analyzes it under all possible inputs.

Finally, search-based optimization for performance-critical kernels has gained increasing popularity in the high-performance community. Such approaches explore and test candidate implementations from a suitably restricted implementation space, either by executing them on the target machine or by detailed simulation. For example, FFTW [9] uses a planner to try many different execution plans for an FFT at run-time, and pick the best one. SPIRAL [17] generates high-performance DSP kernels by searching the space of possible implementations, taking advantage of the structure of the algorithm and the implementation space to speed up the search. Demmel *et al.* [7] also use search-based methods to generate dense and sparse linear algebra kernels. These systems use hand-crafted code generators to produce candidate implementations, which makes building them a difficult and error-prone task. This in turn restricts the space of implementations that these systems can explore.

Sketching may benefit from adopting search-based tuning techniques: conceivably, a sketch synthesizer can generate a set of cor-

Relax (Red-Black) 2D				Relax 3D				Interpolate			
N	spec	rb2d1	rb2d2	N	spec	rb3d1	rb3d2	N	spec	interp1	interp2
1000	17	10	6	100	15	10	9	75	141	18	18
2000	66	38	24	200	115	77	109	100	338	44	43
3000	153	84	54	300	385	236	634	150	1146	149	147
4000	264	148	97	400	923	585	1650	175	1935	238	232
				500	1787	1174	2428	200	2822	339	335

Table 2. Running times of benchmarks implementations. The size of the grid for the Red-Black code is N^2 and N^3 for 2D and 3D respectively. The size of the fine grid for Interpolate is $(2N)^3$. Time is in milliseconds.

rect completions of the sketch, and then search for the most efficient one.

8. Conclusion

The paper describes a sketching synthesizer for stencil computations. It is the first system for sketching programs that go beyond the finite programs of the original SKETCH language and synthesizer [21]. The synthesis is enabled with an abstraction that makes the stencil synthesis problem finite without sacrificing precision, which allows reduction onto the finite synthesizer. The resulting stencil synthesizer is surprisingly scalable.

Acknowledgment

We are grateful the anonymous referees for their helpful comments. This work is supported in part by the National Science Foundation with grants CCF-0613997, CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, and CNS-0524815, the University of California MICRO program, the MARCO Gigascale Systems Research Center, an Okawa Research Grant, an IBM Graduate Fellowship, and a Hellman Family Faculty Fund Award. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA.

References

[1] W. Ackermann. *Solvable cases of the decision problem*. Studies in Logic and the Foundations. of Mathematics. NorthŰHolland„ 1954.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. 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. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, 2000.

[4] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[5] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proc. TACAS 2007*, March 2007.

[6] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *Int. J. Parallel Program.*, 34(1):61–91, 2006.

[7] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005.

[8] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Elect. Trans. Numer. Anal.*, 10:21–40, 2000.

[9] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.

[10] M. Frigo and V. Strumpen. The memory behavior of cache oblivious stencil computations. *The Journal of Supercomputing*, 39(2):93–112, 2007.

[11] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM Press.

[12] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. A. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In B. Calder and B. G. Zorn, editors, *Memory System Performance*, pages 36–43. ACM, 2005.

[13] K. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods: 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 1999.*, pages 219–237, 1999.

[14] A. Mishchenko, S. Chatterjee, and R. Brayton. Dag-aware AIG rewriting: A fresh look at combinational logic synthesis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 532–535, New York, NY, USA, 2006. ACM Press.

[15] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt). *International Journal on Software Tools for Technology Transfer (STTT)*, 2, December 1998.

[16] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.

[17] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, accepted for publication.

[18] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner. Compiling stencils in high performance fortran. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, New York, NY, USA, 1997. ACM Press.

[19] S. Sellappa and S. Chatterjee. Cache-efficient multigrid algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):115–133, 2004.

[20] L. Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, 1999.

[21] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2006)*, pages 404–415, New York, NY, USA, 2006. ACM Press.

[22] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS '06*, San Jose, CA, USA, 2006. ACM Press.

[23] D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):1–221, 2002.