Automatically Translating Image Processing Libraries to Halide

MAAZ BIN SAFEER AHMAD, University of Washington, Seattle
JONATHAN RAGAN-KELLEY, University of California, Berkeley
ALVIN CHEUNG, University of California, Berkeley
SHOAIB KAMIL, Adobe

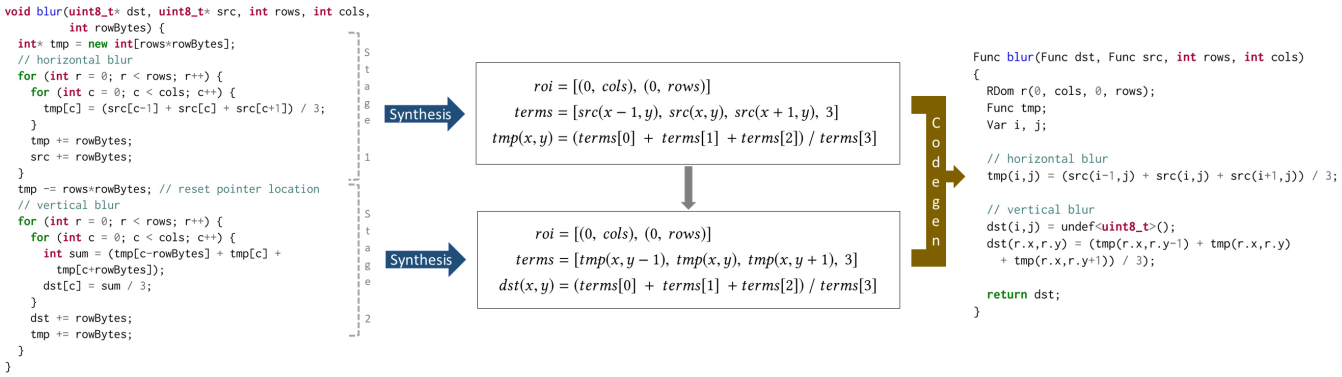

Fig. 1. Dexter parses the input C++ function (shown on the left) into a DAG of smaller stages, then uses our 3-step synthesis algorithm to infer the semantics of each stage, expressed in a high-level IR (middle). Finally, code generation rules compile the IR specifications into executable Halide code (right).

This paper presents Dexter, a new tool that automatically translates image processing functions from a low-level general-purpose language to a high-level domain-specific language (DSL), allowing them to leverage cross-platform optimizations enabled by DSLs. Rather than building a classical syntax-driven compiler to do this translation, Dexter leverages recent advances in program synthesis and program verification, along with a new domain-specific synthesis algorithm, to translate C++ image processing code to the Halide DSL, while guaranteeing semantic equivalence. This new synthesis algorithm scales and generalizes to much larger and more complex functions than prior work, including the ability to handle tiling, conditionals, and multi-stage pipelines in the original low-level code. To demonstrate the effectiveness of our approach, we evaluate Dexter using real-world image processing functions from Adobe Photoshop, a widely used multi-platform image processing program. Our results show that Dexter can translate 264 out of 353 functions in our test set, with the original implementations ranging from 20 to 150 lines of code. By leveraging Halide's advanced auto-scheduling capabilities, we get median speedups of 7.03× and 4.52× for Dexter-translated functions as compared to the original implementations on Intel and ARM architectures, respectively.

CCS Concepts: • **Computing methodologies → Image processing**; • **Software and its engineering → Search-based software engineering**; *Automatic programming*;

Additional Key Words and Phrases: Verified Lifting, Machine Programming, Stencil Computation

ACM Reference Format:
Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.* 38, 6, Article 204 (November 2019), 13 pages. https://doi.org/10.1145/3355089.3356549

Authors' addresses: Maaz Bin Safeer Ahmad, University of Washington, Seattle, maazsaf@cs.washington.edu; Jonathan Ragan-Kelley, University of California, Berkeley, jrk@berkeley.edu; Alvin Cheung, University of California, Berkeley, akcheung@cs.berkeley.edu; Shoaib Kamil, Adobe, kamil@adobe.com.

0730-0301/2019/11-ART204 $15.00
https://doi.org/10.1145/3355089.3356549

1 INTRODUCTION

Domain specific languages (DSLs) for image processing [Adobe 2010; Guenter and Nehab 2010; Ragan-Kelley et al. 2013] enable high performance, portability, and maintainability, but extending these benefits to existing low-level code is difficult. Rewriting entire legacy applications in DSLs requires huge amounts of human effort and risks adding bugs. Building compilers to automatically translate low-level image processing code to high-performance DSLs using traditional code rewriting techniques such as syntax-directed translation [Aho et al. 2006] is fragile, since these methods are prone to failure when code does not exactly match expected syntactic patterns. These techniques also provide no guarantees that the translated code is correct.

In this paper, we propose a different way to solve the translation problem. Rather than constructing syntax matching rules, we consider the translation problem as a *search*: given the input code, we build and solve a search problem that helps us find a provably semantically equivalent program in the target language, finding solutions with the aid of recent advances in program synthesis [Bodík and Jobstmann 2013; Gulwani 2010] and automatic formal verification techniques. To validate this concept, in this paper we take image processing operations and pipelines written in C++ and translate them to Halide [Ragan-Kelley et al. 2012], a high-performance DSL for image computation.

While prior work has proposed similar approaches for different application domains [Ahmad and Cheung 2018; Cheung et al. 2013; Kamil et al. 2016], translating image processing code presents unique challenges. First, although Halide is not a Turing-complete general purpose language, the number of programs expressible is nevertheless huge. In addition, image processing works with arrays, but the lack of first-class multidimensional arrays in languages like C++ leads to low-level code including pointer arithmetic and using single-dimensional arrays to represent multi-dimensional data. Prior work in other domains has also not considered operations common in low-level image processing code like bit-shifting, widening and narrowing casts, and exact integer arithmetic over multiple data types, each of which present challenges for program synthesis and verification. Combined with various low-level optimizations common in C++ image processing code (e.g., loop tiling or vectorization), these characteristics make reasoning about, and hence translating, image processing code difficult.

We rely on two insights to make the problem tractable. First, rather than searching directly over the textual representation of Halide programs, we search over an intermediate representation (IR) that closely resembles Halide, but without details such as type annotations. Second, to make the search scalable, we decompose the search problem into three parts: given an image processing operation written in C++, we first reason about the number of arrays modified by the computation, along with their dimensionality. We analyze how the arrays are traversed to determine the region of interest (ROI) of the operation. Next, we identify the inputs to the operation: the input array reads and scalars used to compute the output. Finally, we use the information inferred from the first two steps to reason about the actual computation performed over the input image to generate the output. Our algorithm infers full specifications for image processing functions and pipelines, which can then be straightforwardly used to generate executable Halide code. We argue that our new search-based algorithm is both simpler and more general than designing ad hoc syntax matching rules in a traditional compiler.

To evaluate the effectiveness of our approach, we have implemented our translation algorithm in Dexter, a translator for rewriting C++ image processing functions in Halide. Dexter performs translation by first synthesizing a summary program written in our IR that decomposes the image processing operation into the three components described above. Then, it uses the synthesized summary to generate executable Halide code. Finally, Dexter leverages the Halide auto-scheduler [Adams et al. 2019] to generate efficient schedules for the translated functions. We show that our Dexter prototype can translate 264 image processing functions, developed over decades, from Adobe Photoshop source code performing various blend and filter operations. In addition, we show that Dexter can also translate complex, difficult-to-understand optimized implementations containing vector intrinsics and loop tiling optimizations, along with multi-stage image processing pipelines.

Overall, this paper presents the following contributions:

- We describe a three-stage search algorithm for specification inference of image processing functions. Our algorithm is much more scalable than existing techniques, both in terms of the ability to infer specifications from complex code and the time needed to infer them. Furthermore, the algorithm expands the types of operations supported (e.g., bit-wise operations, which are not supported in existing synthesizers).
- We describe how Dexter translates larger image-processing functions, such as those implementing multi-stage pipelines, by parsing the functions into a directed-acyclic graph (DAG), where each node in the DAG corresponds to a loop-nest implementing an individual operation in the pipeline.
- We implement a prototype called Dexter[1] based on our algorithm, and show that it can automatically translate 264 functions from a set of 353 functions from the source code of Adobe Photoshop. These functions, implemented using over 36k lines of C++ code, include complexities such as vectorization, loop-tiling, type-casting, bitwise operations, reductions and conditionals, all of which were beyond the scope of prior work [Kamil et al. 2016]. By leveraging Halide and its auto-scheduler, our translated functions are not only more portable, but perform up to 73× faster than the original implementations.

In the rest of the paper, we first discuss prior work and background in §2. Then in §3, we give an overview of Dexter using an example, and discuss our three-stage algorithm in §4. We describe the implementation of Dexter in §5, followed by experimental results in §6.

2 RELATED WORK & BACKGROUND

2.1 Automatically Translating Image Processing Code

We discuss related techniques that optimize either image processing or stencil code via automatic rewriting or compilation in three categories: dynamic analyses that use runtime techniques for optimization, hybrid analyses that use both compilation and runtime mechanisms to optimize input code, and classical compilation (i.e., static analysis).

2.1.1 Dynamic Analysis. Dynamic analysis based techniques perform runtime profiling of existing code to derive equivalent translations. Helium [Mendis et al. 2015], an example of such a tool, identifies and converts image processing kernels from stripped binaries to Halide. Helium uses dynamically generated program traces to learn the shapes and values of the input and output buffers, generalizing the computation into a symbolic expression tree that is then used to generate Halide code. Such runtime techniques are fast and can be used even if the code is available only in binary form. However, the reconstructed kernels are merely an approximation of the original code based on the observed set of traces and such techniques do not offer any soundness guarantees. Furthermore, if the traces only exercise a specific set of parameters (for example, a single blur radius for a filter that supports user-definable blur radii), the translated function will only support the specific observed parameters, limiting the tool's usefulness.

2.1.2 Static-Dynamic Hybrid Analysis. To compensate for the lack of soundness guarantees in dynamic techniques, hybrid analysis uses static compilation techniques in addition to runtime profiling.

[1] http://dexter.uwplse.org

For instance, STNG [Kamil et al. 2016] is a compiler for translating FORTRAN stencils to Halide. It statically analyzes the input code to ensure that the generated Halide implementations are semantically equivalent to the original over all possible inputs.

Like Dexter, STNG uses program synthesis to find a valid translation given the input. In the absence of a scalable synthesis algorithm, such as the one described in this paper, STNG restricts its search to a space of candidate Halide programs defined by a template. STNG constructs these templates through dynamic analysis of program traces, similar to Helium. Therefore, STNG's approach, although sound, suffers from many of the same limitations as Helium. The generated templates are often over-fitted to the set of traces observed and can exclude valid translations from the search space. While any translation found by STNG is guaranteed to be correct, STNG is limited to translating only simple functions as each runtime trace can capture only a single path through the complex control flow in a program, and reconstructing the original control flow through a small set of traces remains challenging. In addition, STNG cannot handle many important operations found in image processing, including casting and bitwise arithmetic.

Similar hybrid approaches are used by systems outside of image processing that synthesize programs based on input-output examples, such as Scythe [Wang et al. 2017], which synthesizes database queries based on user-provided input-output pairs, and FlashFill [Gulwani 2011], a feature in Microsoft Excel that uses examples to guess user-intended transformations.

2.1.3 Syntax-Driven Compilation. Classical source-to-source transformations have been utilized to generate optimized code from higher-level descriptions, based on syntax-driven transformations, which enable fast performance when input code matches expected syntactic forms. As discussed in §1, such compilers are based on syntax matching rules to translate input programs, and developing such rules requires major engineering effort. Such approaches have been used to find optimal schedules for image processing operations [Boechat et al. 2016; Mullapudi et al. 2016; Ragan-Kelley et al. 2012], along with compiling image processing operations to hardware [Hegarty et al. 2014, 2016].

[Yang et al. 2016] employs syntax-driven techniques to translate image processing code in Python by transforming the Python abstract syntax tree into lower-level Cython [Dalcin et al. 2010] code, which is a mixture of Python and C, while performing several program transformations. Similarly, SEJITS specializers [Catanzaro et al. 2009; Kamil et al. 2012] use syntax-driven code generation for translating subsets of Python code into various languages. Such systems invariably handle only a subset of ways input programs can encode their operations and do not provide the kinds of correctness guarantees possible with program verification.

2.2 Program Synthesis and Verification

In Dexter, we use program synthesis to infer a *summary* for each image processing operation in the input library. Program synthesizers take in two inputs: a search space of candidate program summaries written in our IR, and a way to verify if a candidate is semantically equivalent to the input code. The former is described using a grammar over our IR, to be discussed in §4. For the latter, we leverage Hoare-style verification conditions [Hoare 1969] that are readily expressible in forms understood by solvers such as Z3 [De Moura and Bjørner 2008].

Dexter relies on the Sketch [Solar-Lezama 2019] program synthesizer to generate and search through the candidate program summaries, in conjunction with a solver for validation. There have been a number of approaches to solve the synthesis problem [Bodík and Jobstmann 2013; Gulwani 2010] using different algorithms, such as constraint-based search [Solar-Lezama et al. 2007], enumerative search [Phothilimthana et al. 2016], or stochastic search [Schkufza et al. 2014]. Internally, Sketch solves the search problem by sampling the search space, and using a SAT solver to check for the validity of the sampled programs. If a sample is incorrect, the solver will return a counterexample (i.e., an image that, when fed into the sample and input program, returns different results). Sketch will then reduce the space by sampling only from those programs that satisfy the set of counterexamples found thus far. This proceeds iteratively until either Sketch finds a correct program or the search times out.

Unfortunately, standard algorithms for synthesis fail to solve our translation problem, as the number of candidate programs is simply too large to be considered by an existing synthesizer. To make the search efficient, specialized algorithms have been developed for different application domains, and we discuss how we address the issue for image processing operations in §4.

3 OVERVIEW

We now describe how Dexter translates image processing functions in C++ to Halide, using an example to illustrate the workflow.

3.1 Image Processing Functions

Dexter targets image processing functions written in standard C++. Such functions are often expressed using a sequence of loop nests that iterate over the input buffers to compute intermediate or output buffers. Each loop nest iterates through a region of interest (ROI) and, for each point i within the ROI, computes the corresponding value in the output buffer using a neighborhood of values around i and invoking different kinds of operators, such as arithmetic, bitwise, and conditional expressions (i.e., Halide's `select` operator); array reads using i; and reductions. The input image can be stored using arrays, vectors, or even user-defined types (UDTs). We outline the set of C++ features supported by our implementation in §5.1. Dexter only targets code that implements image processing logic, and does not translate setup or logging code present in image processing applications (e.g., memory allocation, I/O, etc), as such code does not yield performance improvement even if expressed in Halide.

3.2 Translating Image Processing Functions to Halide

The input to Dexter is a library of image processing functions implemented in C++. As output, Dexter generates a new, semantically equivalent version of the input library implemented using Halide. Dexter translates the input by parsing each function as a directed acyclic graph (DAG), with each node in the DAG corresponding to a loop nest in the input code, and synthesizing a semantically equivalent Halide function for each node in the DAG. Each translated function then becomes a stage in the overall Halide pipeline.

```
1  void blur(uint8_t* dst, uint8_t* src, int rows, int cols,
2            int rowBytes) {
3    int* tmp = new int[rows*rowBytes];
4    // horizontal blur
5    for (int r = 0; r < rows; r++) {
6      for (int c = 0; c < cols; c++) {
7        tmp[c] = (src[c-1] + src[c] + src[c+1]) / 3;
8      }
9      tmp += rowBytes;
10     src += rowBytes;
11   }
12   tmp -= rows*rowBytes; // reset pointer location
13   // vertical blur
14   for (int r = 0; r < rows; r++) {
15     for (int c = 0; c < cols; c++) {
16       int sum = (tmp[c-rowBytes] + tmp[c] +
17         tmp[c+rowBytes]);
18       dst[c] = sum / 3;
19     }
20     dst += rowBytes;
21     tmp += rowBytes;
22   }
23 }
```

(a) Input: C++ Implementation of a 3×3 box blur filter.

$$roi = [(0;\ cols);\ (0;\ rows)]$$
$$terms = [src(x-1;y);\ src(x;y);\ src(x+1;y);\ 3]$$
$$tmp(x;y) = (terms[0] + terms[1] + terms[2]) / terms[3]$$

$$roi = [(0;\ cols);\ (0;\ rows)]$$
$$terms = [tmp(x;y-1);\ tmp(x;y);\ tmp(x;y+1);\ 3]$$
$$dst(x;y) = (terms[0] + terms[1] + terms[2]) / terms[3]$$

(b) Summary expressed using Dexter's IR that describes the blur function.

```
1  Func blur(Func dst, Func src, int rows, int cols) {
2    RDom r(0, cols, 0, rows);
3    Func tmp; Var i, j;
4    tmp(i,j) = (src(i-1,j) + src(i,j) + src(i+1,j)) / 3;
5    dst(i,j) = undef<uint8_t>();
6    dst(r.x,r.y) = (tmp(r.x,r.y-1) + tmp(r.x,r.y)
7      + tmp(r.x,r.y+1)) / 3);
8    return dst;
9  }
```

(c) Output: Halide implementation of the blur function, as generated from the IR summary shown in b).

Fig. 2. Using Dexter to translate a 3×3 box blur filter from C++ to Halide, with casting removed for clarity.

To demonstrate this process, we show how Dexter translates a 3×3 box blur to Halide. As shown in Figure 2a, the original implementation uses the composition of a 1×3 and a 3×1 blur filter, each implemented as a pair of nested loops, to compute the overall 3×3 blur. The first loop nest iterates the source image `src` and saves the intermediate output in `tmp`. Each iteration of the outer `for` loop (Lines 5–11) uses the inner `for` loop (Lines 6–8) to compute the `r`'th row of the `tmp` buffer, then moves the input and output buffer pointers to the first column of the next row (Lines 9–10) using pointer arithmetic. Similarly, the second loop nest iterates over the intermediate buffer and performs a 3×1 vertical blur to compute the final output stored in `dst`. Unfortunately, generating a Halide implementation using syntax-driven rules directly from this C++ implementation is challenging given the myriad of ways the same computation can be expressed in low-level languages like C++.

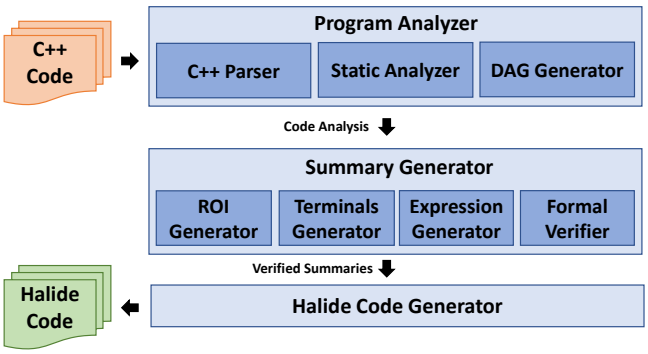

Fig. 3. Dexter's system architecture. Image processing functions in C++ (orange) are translated into Halide (green).

Dexter's goal is to rewrite the code into Halide by inferring a summary expressed using an intermediate representation (IR) based on Halide, as shown in Figure 2b. The summary describes the blur function as a DAG of two operations, where each operation in the DAG is described using three components: its ROI, which describes the range and dimensionality over which the operation is realized, its *terms*, which is the set of all array reads, constants, and scalar variables used in the computation, and finally how each point in the output image is computed using the set of available terms.

Generating Halide code from the IR summary is now straightforward, as the summary eliminates all scheduling information as well as any low-level optimizations (like vectorization) that may exist in the input code, leaving only a declarative description of each output point. We show the generated Halide code for the blur function in Figure 2c. Once expressed in Halide, the pipeline can then be optimized by the auto-scheduler, for instance, by merging the two stages when iterating over each location in the source image.

Instead of relying on pattern-matching translation rules, which are brittle and difficult to construct manually, the key insight in Dexter is to use a search-based technique known as *program synthesis* [Bodík and Jobstmann 2013; Gulwani 2010] to find the equivalent IR summary, as described in §2.2. Unfortunately, searching through the space of all possible summaries is prohibitively expensive, and has not been addressed by state-of-the-art program synthesizers. Hence, Dexter uses a new algorithm (discussed in §4) to make the search efficient.

3.3 System Architecture

Figure 3 shows Dexter's overall architecture comprising three modules that make up its compilation pipeline.

First, the *program analyzer* parses input C++ code into an Abstract Syntax Tree (AST) and statically analyzes each library function to identify important features about the code, such as the set of variables that are read (input) or modified (output). Then, it parses each function's AST into a DAG of smaller operations before sending it along with the analysis results to the next module.

$$
\begin{array}{rcl}
ROI & := & [B_0; B_1; :::; B_D] \\
B_i & := & (IntExpr; IntExpr) \\
IntExpr & := & intvars \mid const \mid IntExpr\ Op\ IntExpr \\
Op & := & + \mid - \mid \times
\end{array}
$$

(a) Grammar for synthesizing an operation's region of interest.

$$
\begin{array}{rcl}
Term & := & intvar \mid floatvar \mid boolvar \mid const \\
 & \mid & arrvar(Index; :::) \\
Index & := & intvar \mid intvar \pm const \mid const \\
 & \mid & arrvar(Index; :::)
\end{array}
$$

(b) Grammar for synthesizing terminal mappings.

$$
\begin{array}{rcl}
Expr & := & terms \mid iden \mid Expr\ BOp\ Expr \mid UOp\ Expr \\
 & \mid & (Expr\ ?\ Expr\ :\ Expr) \mid f(Expr; :::) \\
 & \mid & cast\langle Type\rangle(Expr) \\
Type & := & float \mid uint8\_t \mid int8\_t \mid uint16\_t \mid ::: \\
BOps & := & + \mid - \mid * \mid / \mid << \mid\ \& \mid\ != \mid ::: \\
UOps & := & \sim \mid - \mid !
\end{array}
$$

(c) Grammar for synthesizing the computation performed in an operation.

Fig. 4. Search grammars used to synthesize summaries for image processing operations. Each summary represents a possible Halide tranlsation for the input operation.

Next, the *summary generator* synthesizes summaries expressed using Dexter's IR for each operation in the DAG. First, the ROI generator synthesizes the dimensionality and ROI for each operation. Then, the *terminals generator* synthesizes a mapping for all terminals (variables and array reads) found in the input code into a normalized iteration space. Finally, the *expression generator* synthesizes an expression that encodes the computation performed in each operation using the previously synthesized terminals. To ensure semantic equivalence to the input, any summary candidate identified by the summary generator is passed to the verifier for validation. We discuss the search and verification procedure in detail in §4.

Once a summary is inferred and verified, the *code generator* translates the summary from Dexter's IR into executable Halide code. The translation is straightforward given the resemblance between the IR and Halide. The code generator traverses the summary and generates equivalent Halide code for each expression, outputting a compilable Halide generator that can be combined with an optimized schedule to produce high performance code. We provide more details of the code generation process in §5.4.

4 FINDING SUMMARIES FOR IMAGE PROCESSING OPERATIONS

Dexter uses program synthesis to find translations of image processing operations. To search for translations, we define the search space of Halide programs using a *grammar*, an excerpt of which is shown in Figure 4. Given this grammar, the synthesizer will conceptually enumerate all programs that can be constructed using it by randomly choosing a production rule up to a fixed number of times, and check if any of the constructed Halide programs is semantically equivalent to the input.

Unfortunately, this process is prohibitively expensive: even if we limit the synthesizer to expand only up to 5 production rules, the grammar shown in Figure 4 expands to tens of thousands different Halide programs; such a search space is at least an order of magnitude larger than what any state-of-the-art synthesizer can handle, making this approach infeasible without further optimizations.

Our key insight to make the search problem tractable in Dexter is to exploit domain-specific knowledge about image processing operations. In particular, we observe that we can decompose many such operations into three components:

- A *Region of Interest* (ROI) that describes the dimensionality of the operation and the bounds for each dimension within which the output is realized.
- The set of *terminals* used to compute the value of each point within the ROI. Such terminals can consist of numeric constants, program variables, or array reads.
- The *computation* performed using the aforementioned set of terminals to compute the values inside the ROI.

Dexter exploits this insight to decompose the overall summary synthesis problem into three separate synthesis sub-problems, each targeting one component above.

4.1 Synthesizing the Region of Interest

An image processing operation's region of interest (ROI) describes its dimensionality and bounds. In this section, we describe how Dexter synthesizes the ROI for each image processing operation.

4.1.1 ROI Grammar. Like other synthesis problems, Dexter synthesizes the ROI for each image processing operation by encoding the search space of candidate ROIs using the grammar shown in Figure 4a. In the grammar, each ROI consists of D *bound expressions* ($B_i ::: B_D$), one for each dimension. Each bound expression consists of an upper and lower bound $IntExpr$ that is made up of integer constants ($const$), the set of integer variables and pointers read or updated ($intvars$) extracted through static analysis of each input function, and combinations of such expressions using arithmetic operators.

For example, Figure 2b shows the ROI for each operation in the box-blur function, shown in Figure 2a. It describes the 1x3 row-blur as two-dimensional, with bounds $0 \le d_1 < cols$ and $0 \le d_2 < rows$ for the first and second dimension, respectively. The synthesizer can construct this ROI from the grammar by first setting D to be 2, and then applying the appropriate production rules shown in Figure 4a to construct the bound expressions for each dimension.

4.1.2 ROI Verification. To synthesize the ROI, we need a way to check whether a candidate ROI is correct. Recall that we have not yet synthesized the set of terminals used or the actual computation performed by the input code. Hence, to verify a candidate ROI, we create a "reduced" version of the input code fragment, where all statements in the fragment's body are removed, except for those (if any) that update loop counters, array pointers, or array contents. We replace all array updates with the special value $\perp$ to indicate

```
1  RDom r(0, cols, 0, rows);
2  Func Var i, j;
3  dst(i, j) = undef<int>(); // ROI's contents undefined
4  dst(r.x, r.y) = ⊥; // except for locations within r
```

(a) A candidate ROI expressed in Halide.

```
1  for (int r = 0; r < rows; r++) {
2    for (int c = 0; c < cols; c++) {
3      dst[c] = ⊥;
4    }
5    dst += rowBytes;
6    tmp += rowBytes;
7  }
```

(b) A reduced version of the 3x1 column-blur used to synthesize the ROI.

Fig. 5. Dexter synthesizes the ROI of an image processing operation by constructing a reduced version of the input code fragment.

that the array entry has been updated, but using an expression that we do not yet know (to be synthesized in the last step).

Consider again the 3x3 blur function from Figure 2a. To synthesize the ROI for the second operation in the pipeline (3x1 column-blur), Dexter prepares a reduced version of the loop nest, shown in Figure 5b. Given this code, Dexter generates ROI candidates using the grammar shown in Figure 4a. To check the validity of a candidate, Dexter creates a skeletal Halide program; one that corresponds to the ROI candidate $[(0; cols); (0; rows)]$ is shown in Figure 5a, where the special value $\perp$ is written to the ROI defined by the reduction domain (r) on Line 4. The validity of a candidate ROI is determined by checking the equivalence of the candidate (Figure 5a) and the reduced input code (Figure 5b) through program verifiers.

4.2 Synthesizing the Terminal Mappings

Once an operation's ROI is synthesized, Dexter next infers the computation performed by the code fragment for each location within the ROI. As discussed earlier, Dexter partitions this problem into two further steps. First, Dexter learns the terminals used in the computation, such as variables, constants, and array reads. Recall from §4.1 that we replaced the values of all array updates with the special value $\perp$. Conceptually, the goal of this step is to learn the arguments that are needed to compute $\perp$, as shown in Figure 6a.

4.2.1 Extracting Terminals. To extract the set of terminals, Dexter statically analyzes the input code for each operation in the function. The analysis starts at each statement that updates the output image (such as Line 18 in Figure 2a), and extracts all terminals involved in these assignments, i.e., `sum` and 3. Then, it traverses the code backwards to recursively extract all terminals used to compute the extracted values. Since `sum` is in the extracted set, the terminals `src[c-rowBytes]`, `src[c]` and `src[c+rowBytes]` replace `sum` in the extracted set after Line 16 in Figure 2a is analyzed. The final set of terminals extracted for the 3x1 column-blur operation is {`src[c-rowBytes]`, `src[c]`, `src[c+rowBytes]`, 3}. These serve as the inputs to $\perp$ as shown in Figure 6c.

4.2.2 Mapping Terminals. The terminals extracted through static analysis are defined in the context of loops found in the original code, which can contain low-level optimizations (such as tiling and array flattening) that use different indexing than the Halide code to be synthesized. For example, the input code shown in Figure 2a

$$roi = [(0; cols); (0; rows)]$$
$$dst(x;y) = \perp(??)$$

(a) The goal is to synthesize the arguments required to compute $\perp$.

```
1  RDom r(0, cols, 0, rows);
2  Var i, j;
3  dst(i,j) = undef<int>();
4  dst(r.x,r.y) = ⊥(src(x,y-1), src(x,y), src(x,y+1), 3);
```

(b) A candidate mapping expressed in Halide.

```
1  for (int r = 0; r < rows; r++) {
2    for (int c = 0; c < cols; c++) {
3      dst[c] = ⊥(src[c-rowBytes], src[c], src[c+rowBytes], 3);
4    }
5    dst += rowBytes;
6    src += rowBytes;
7  }
```

(c) A reduced version of the 3x1 column-blur used to synthesize terminal mappings.

Fig. 6. Once the ROI has been determined, Dexter synthesizes a mapping for terminals found in the input code.

stores the input image `src` as a 1-D array, while the translated Halide code stores the input as a 2-D array as determined by the ROI. Hence, the terminal `src[c-rowBytes]` in the input code can be mapped to $src(x; y-1)$ in the Halide summary, where x and y are the loop induction variables bound to the two dimensions of the ROI. Determining the mappings for constants is trivial: all constants map to themselves. For all other terminals, Dexter synthesizes their mappings through program synthesis.

4.2.3 Grammar for Terminal Mapping. Figure 4b describes the grammar used to synthesize terminal mappings. A terminal (*Term*) can map to scalar values or array reads. While generating array indexing expressions (*Index*), the grammar allows offsetting integer variables, such as the induction variables, by a constant. This allows the synthesizer to explore reads from neighboring indices. Finally, the grammar can also express indirect array accesses to handle code that use pre-computed lookup tables, for instance, the terminal `histogram[src[i]]` that looks up from a pre-computed histogram based on the current location's pixel value.

4.2.4 Verifying Mappings. Dexter again constructs a reduced version of the input to check the correctness of any synthesized mapping. Like ROI synthesis, Dexter removes all statements in the input code fragment, except for loops and assignments to output arrays. Rather than changing array assignments to $\perp$, Dexter instead changes array assignments to a special $\perp$ function with parameters being the extracted terminals. Dexter then generates a similar skeletal Halide program, an example of which is shown in Figure 6b, with the special assignment shown on Line 4. Verification, like before, is done by checking the equivalence of the two programs.

4.3 Synthesizing the Computation

The final step in synthesizing summaries is to infer how the terminals combine to compute the values used to update the locations within the ROI. To do so, Dexter replaces the special value $\perp$ (as shown in Figure 6a) with an actual Halide expression.

4.3.1 Expressions Grammar. Figure 4c shows the grammar to used to synthesize Halide expressions. Besides simple expressions such as arithmetic expressions, Dexter also supports conditionals in the form of ternary operations, as well as type-casting to different integer bit-widths, between integer and floating point representations, along with signed and unsigned representations. The only terminals available in the grammar (*Terms*) are the terminals synthesized in the second stage and the special *iden* terminal which represents no-op. The no-op operator is useful for describing operations such as threshold blends, where the input data values control whether a point in the ROI is modified or not.

4.3.2 Verifying the Summary. Replacing $\perp$ with a candidate Halide expression yields a candidate summary of the input code. Similar to the previous stages, Dexter verifies a candidate by constructing the corresponding Halide program and then testing the equivalence of the generated candidate program with the *original* code fragment. If verification succeeds, then we have found a valid translation of the input code. If not, the synthesizer attempts to generate another candidate, until it exhausts the search space (i.e., the search space encoded by the grammar is not expressive enough), or it times out.

As explained in §2.2, Dexter uses the Sketch synthesizer to sample the search space for each of the three sub-problems created by our algorithm. In Appendix A, we provide supplementary details on the finer optimizations used by Dexter to optimize the search process. Once synthesized, the summary is sent to the code generator to produce executable Halide code. We discuss the details in §5.4.

5 IMPLEMENTATION

We implemented Dexter's program analyzer using the Clang [Lopes and Auler 2014] compiler's `libTooling` library to parse C++ code into an abstract syntax tree (AST). The analyzer traverses the AST to perform static analysis and DAG generation, and sends the results to the summary generator. Dexter's summary generator, implemented in Java, uses an off-the-shelf synthesizer called Sketch [Solar-Lezama 2019], along with Z3 [De Moura and Bjørner 2008] for verification. The code generator for parsing the synthesizer's output and generating the output Halide code is also implemented in Java.

In the remainder of this section, we first outline the subset of C++ that Dexter supports. Then, we discuss how users can interact with, extend, and fine-tune Dexter. Last, we provide details about Halide code generation.

5.1 Supported C++ Constructs

To translate any input code fragment, Dexter must parse the input and generate search grammars for the different components (as described in §4). Dexter currently supports a core set of C++ constructs, such as basic assignment and declaration statements, conditionals, loops, functions, and user-defined types.

5.1.1 Types Supported. Dexter supports all built-in primitive C++ data types and operators. It also processes reads and writes into primitive arrays or `std::vector` types. Dexter only supports pointers that represent dynamically sized primitive arrays, and internally models them as a data array and an integer offset that represents the pointer's location within the array. This enables supporting pointer de-referencing and arithmetic when generating search grammars.

To support user-defined types, Dexter traverses the program AST to find declarations of all user-defined `structs` used in the code being translated. It then adds these types to the underlying synthesizer's and verifier's type systems. This is useful especially when planar image data is stored in a `struct` with arrays for each channel. Dexter can also generate search grammars for code that involves user-defined types, including the use of constructors and methods.

5.1.2 Loops. Dexter can process different types of loops (`for`, `while`, `do`), including those with loop-carried dependencies after applying classical transformations [Aho et al. 2006] to convert them into `while(true){...}` loops.

5.1.3 Functions. Dexter handles function calls by inlining the function bodies, except if the function being called is pure and computes a scalar quantity from other scalar quantities. Dexter translates such functions to equivalent pure functions in our IR and adds them to the search space to keep the generated code clean and understandable. For example, in Figure 7, the output `dst` is computed using the function `Mul8x8Div255`, which multiplies the two input 8-bit values, divides by 255, and returns the result. Since `Mul8x8Div255` is a pure function, Dexter will add it to the expressions grammar in Figure 4c. Dexter currently does not support recursive functions and functions with side-effects other than array writes, since neither are expressible in Halide.[2]

5.1.4 External Library Functions. Users can provide semantic models of external library functions used in the input code by implementing them using Dexter's IR. Dexter already comes with built-in support for a number of common functions from the C++ standard library (e.g., `min`, `max`, `abs` etc).

5.2 DAG Construction

Image processing functions often implement a pipeline of operations computing multiple intermediate and output values. Summaries expressed in our IR that describe the output of an entire multi-stage pipeline are not only difficult to synthesize (due to their potentially large size), but they often do not exist as not all pipelines can be inlined into a single operation. This is the case, for example, of a pipeline where an earlier stage computes a histogram that a later stage uses. To address this issue, Dexter parses the input functions into a DAG, where each node in the graph represents a single loop-nest found in the code and is treated as an operation in a larger pipeline. This allows us to introduce ordering between different fragments of computation within the function, all of which can be expressed using our IR and therefore be translated to Halide to produce a Halide pipeline.

Dexter generates the DAG through a forward traversal of the function's statements, assigning each statement to a stage in the DAG. At the start, Dexter initializes the DAG with a single stage that has no statements. It then adds all basic program statements, such as variable assignments and declarations, to the current stage

[2]Functions with side effects can be called from Halide by using extern functions, but such translations are beyond the scope of this paper.

```
1  void adjustOpacity(uint8_t* dst, int opacity, int rows,
2  int cols, int rowBytes) {
3    assert (cols <= rowBytes); // required user annotation
4    for (int r = 0; r < rows; r++) {
5      for (int c = 0; c < cols; c++) {
6        dst[c] = Mul8x8Div255(dst[c], opacity);
7      }
8      dst += rowBytes;
9    }
10 }
```

Fig. 7. User annotation helps Dexter determine that each index of the array is updated only once.

```
1  for (int row = 0; row < rows; row++) {
2    for (int col = 0; col < cols; col++) {
3      int x = msk[row*cols + col];
4      x = 255 - x + noiseData[HashFunction(row,col)];
5      if (x < 256)
6        msk[row*cols + col] = 255;
7      else
8        msk[row*cols + col] = 0;
9    }
10 }
```

Fig. 8. Synthesizing the mapping for terminal `noiseData` requires synthesizing the hash function.

of the DAG until it reaches a loop nest. Once a loop is encountered, Dexter adds the loop nest to the current stage and creates a new stage as a child of the previous stage. Dexter then resumes the process until either another loop is encountered or all of the function statements have been assigned. A special case is made for conditional statements (e.g., `if`) that contain a loop in either branch of the control flow path (or both). Each branch of the conditional is recursively parsed into a DAG, with the heads of each sub-DAG connected to the current graph as child nodes (representing a fork in the DAG). The last stages in each of the sub-DAGs are merged back together just as the original program control flow merges.

As an illustration, the blur function in Figure 2a is parsed by Dexter into two consecutive stages, where stage 1 contains all statements from Line 3 to 11, and stage 2 contains all statements from Line 12 to Line 22. Dividing the input code this way replaces one difficult synthesis problem (finding a summary that involves 10 terms) to two much simpler synthesis problems (finding summaries involving only 4 terms).

5.3 User Interaction

In this section, we discuss the miscellaneous inputs a user may provide to Dexter and how users may tune or extend the system in the future.

5.3.1 Code Annotations. Occasionally the functions in a library make assumptions about the input parameters that are not explicitly expressed in the source code and yet are essential to its correctness. For instance, the code shown in Figure 7 takes as input variables `cols` and `rowBytes` representing the number of columns to compute in each row and the width of the output buffer row in bytes, respectively. The code implicitly assumes that the number of columns is less than or equal to the row-width; otherwise, the assignment on Line 6 would be executed multiple times for some locations in `dst`. Because of this possibility, Dexter will fail to translate the code fragment as there exist inputs where the fragment is not equivalent to a two-dimensional Halide assignment. Users can help Dexter translate such kernels by adding annotations, such as the `assert` statement on Line 3, to clarify the intent of the code.

5.3.2 Tuning Search Grammar. The default grammar used by Dexter represents a broad class of image processing operations. Users may alternatively want to specialize the grammar to the library they intend to translate. For instance, if the library only includes point-wise operations, the grammar could be adjusted to not explore neighboring points when synthesizing point mappings. Similarly, users may want to compose Halide expressions from a set of custom higher-level library-specific operations. Dexter is designed to make such modifications easy: it allows users to express grammars by writing them in a format similar to those shown in §4. Our default grammar is expressed using fewer than 250 lines of code.

5.3.3 Extending Dexter. Dexter is designed to be highly extensible. For instance, to support custom types or external library functions in the input source code, users only need to provide models for said types and functions using Dexter's IR. To demonstrate the ease of extending Dexter, we discuss two patches to the system that enable the translation of benchmarks that Dexter failed to translate during our evaluation: a dissolve blend and an addition blend.

The dissolve blend uses a specialized hash-function over the loop counters to pseudo-randomly read noise data from a pre-computed table, as shown in Figure 8. To find the mapping for terminal `noiseData[HashFunction(row,col)]`, Dexter would have to synthesize this hash function using our points grammar, which is very challenging. A straightforward solution is to implement `HashFunction` in Dexter using the IR and update the *Index* rule in the default points grammar (Figure 4b) as follows:

$$
\begin{aligned}
\mathit{Index} \quad &:= \quad \mathit{intvar} \mid \mathit{intvar} \pm \mathit{const} \mid \mathit{const} \\
&\mid \quad \mathit{arrvar}(\mathit{HashFunction}(\mathit{Index}; \mathit{Index}); :::) \\
&\mid \quad \mathit{arrvar}(\mathit{Index}; :::)
\end{aligned}
$$

The addition blend fails since it calls the function `UDIV255`, to perform an unsigned divide-by-255, that is implemented using hand-written assembly, a feature currently not supported by Dexter. Providing Dexter with a semantic model of the `UDIV255` function is sufficient to translate this benchmark:

```
uint_t UDIV255(uint_t x) { return x / 255; }
```

5.4 Code Generation

Since the synthesizer outputs code using a stylized subset of Halide, code generation is straightforward and is done via a small set of rules. The ROI described by the summary is used to declare the set of induction variables, one for each dimension, as well as constructing the reduction domain (`Halide::RDom`) to iterate over, which defines the set of points over which the stencil executes. The expressions synthesized in the final step describe how each location in the output buffer is computed, and has a one-to-one correspondence with Halide's `Func` assignment statements.

Figure 9 lists a part of Dexter's code generation function `Gen()`, which takes in a Dexter IR construct and generates executable

$$
\begin{aligned}
\texttt{Gen}(roi = [(lb_0,\ ub_0),\ \ldots]) &= \texttt{RDom}(\texttt{Gen}(lb_0),\ \texttt{Gen}(ub_0), \ldots) \\
\texttt{Gen}(e_1 = e_2) &= \texttt{Gen}(e_1) = \texttt{Gen}(e_2) \\
\texttt{Gen}(\ (e_1\ ?\ e_2\ :\ e_3)\ ) &= \texttt{select}(\texttt{Gen}(e_1),\ \texttt{Gen}(e_2),\ \texttt{Gen}(e_3)) \\
\texttt{Gen}(\ cast{<}\tau{>}(e)\ ) &= \texttt{Halide::cast} <\tau>(\texttt{Gen}(e)) \\
\texttt{Gen}(e_1 + e_2) &= \texttt{Gen}(e_1) + \texttt{Gen}(e_2) \\
\texttt{Gen}(var) &= var
\end{aligned}
$$

Fig. 9. A subset of Dexter's code generation function Gen().

Halide code. Gen() is recursively called: for instance, calling Gen() on $e_1 + e_2$ will recursively call Gen() to translate each operand. Expressions such as variables and constants represent the base cases, as they trivially map to themselves. Translating the required declarations works similarly; Line 1 in Figure 9 converts the synthesized ROI description into an RDom declaration.

6 EVALUATION

In this section, we present a comprehensive evaluation of Dexter's ability to: (1) translate complex and diverse image processing code, and (2) translate code efficiently. Furthermore, we investigate the performance of the compiled Halide library against the original C++ implementation in various contexts.

All benchmarks in our evaluation were compiled on a high-performance server with 4 Intel Xeon E7-4890v2 2.8 GHz 15-core processors, 1 TB of memory, running Ubuntu OS 16.04. For synthesis, Dexter utilized Sketch 1.7.5 with a parallelism factor of 100. Z3 version 4.8.3 was used for verification. Runtime performance evaluation for compiled code was performed using a 15-inch Apple Macbook Pro (2018) with a 6-core 2.6 GHz Intel Core i7 processor and 16 GB of memory, running macOS 10.14.2; and a 2018 12-inch iPad Pro with a 2.5 GHz Apple A12X processor[3] (ARM64 architecture) running iOS 12.2. The Intel machine supports AVX2 vectorization, and the ARM machine supports NEON vector instructions. We use Git commit cf73bfe6 of Halide for all tests, using the default auto-scheduler weights.

6.1 Code for Evaluation

We evaluate Dexter on 3 suites of image processing functions from Photoshop by Adobe, containing a total of 353 performance critical functions. These functions are called when performing a variety of essential image processing operations, including compositing layers and basic transformations such as rotations and blurs. Due to their importance, some essential functions have been hand-optimized with vectorized x86 implementations; however, due to the difficulty of hand-optimization, only a small subset has been optimized, and these implementations do not take advantage of the latest vectorization capabilities of x86 processors.

Blend Suite consists of 186 functions across approximately 13k lines of C++ code, which perform point-wise image blending operations such as Normal, Multiply and Dissolve blends. For the most basic operation, i.e. the Normal blend mode [Porter and Duff 1984], two image layers $A; B$ are combined based on a per-pixel weight W, such that the output pixel c_{output} is a linear combination of input pixels $c_A; c_B$:

$$c_{output} = W \times c_a + (1 - W) \times c_B$$

The set of functions supports a large number of blend modes, but also includes a number of other operations. In addition, the suite contains specialized implementations for specific bit-widths and color formats, as well as specializations where weights are constant.

SSE Blend Suite is a set of hand-optimized blending operations, containing 36 functions implemented in 4.5k lines of code, with a mix of SSE2 intrinsics and hand-written assembly. These are highly non-portable implementations, making it difficult to run Photoshop efficiently on non-x86 hardware.

Filter Suite contains 131 functions implementing various image filtering algorithms that convolve an image with a filter, written in 19k lines of code. These include filters with specific radii, filters for which the radius is an input, as well as specializations for specific image formats.

6.2 Feasibility Analysis

Dexter was able to automatically translate 264 out of 353 functions to Halide, achieving a coverage of 88% for the Blend Suite, 100% for the SSE Blend Suite and 50% for the Filter Suite. The total time required by Dexter to compile all three suites was 182 hours, an average of 47 minutes per function. The compilation time essentially equals the synthesis time, since synthesis dominates the process. Time spent in all other stages, such as parsing, DAG generation and code generation is insignificant.

Of the 89 code fragments that Dexter failed to translate, 51 failed due to lack of front-end support of language constructs in our current implementation, such as embedded assembly instructions, recursive functions or switch statements. Another 38 benchmarks took too long to synthesize and timed-out after 6 hours of search. See §5.3.3 for examples of such failures, as well as a discussion on how Dexter can be patched to translate them. The relatively lower coverage of the Filter Suite is due to the increased complexity of the input code. However, this complexity does not stem from the convolutional nature of the functions but instead from how they are implemented. For example, the Filter suite contains recursive implementations and pointer type-casts, i.e., language features that either our current prototype cannot reason about or are unsupported by Halide.

Photoshop executes the code fragments in our test suite on individual tiles of the image that fit into processor caches. As such, the loops inside the operations do not benefit from tiling optimizations common in image processing. To demonstrate that Dexter can also translate tiled implementations, we manually modified one image operation implementation to execute in tiles of 16×32, as shown in Figure 10. Dexter successfully translated the loops to recover the correct region of interest (ROI) and the untiled implementation. However, synthesizing the ROI for the tiled implementation took approximately 6× longer since two additional invariants were required due to the additional `for` loops.

Compared to Helium [Mendis et al. 2015], which uses dynamic execution traces to perform translation, Dexter translates many

[3]Multithreaded performance is limited to 2.3 GHz.

```
void Darken (uint8_t *dst, uint8_t *src, uint8_t *msk,
int rows, int cols, int rowBytes)
{
 for (int rowOut=0; rowOut < rows; rowOut += 16){
   for (int colOut=0; colOut < cols; colOut += 32){

 for (row = rowOut; row < min(rows, rowOut+16); row++){
   for (col = colOut; col < min(cols, colOut+32); col++){
       uint16_t delta = (src[row * rowBytes + col])
              - (dst[row * rowBytes + col]);
       if (delta < 0)
           (dst[row * rowBytes + col]) -=
                Mul8x8Div255((msk[row * rowBytes + col]),
                             -delta);
    }
   }}}
}
```

Fig. 10. An example of a tiled implementation that Dexter can successfully de-schedule to recover the program summary.

more operations. We ran Helium[4] on Photoshop and were able to fully-translate 7 operations; part of the difficulty in applying Helium is that the operations must be triggered after starting tracing, which is then manually stopped after the operation is complete. Attempting to translate compositing operations fails under this scenario, because the composite calls many different operations, causing Helium's heuristics to fail; thus, all of the successful translations are from the Filter Suite. Of the 7 translated operations, one (`boxBlur`) cannot be translated by Dexter due to its recursive nature; in addition, Helium only translates the radius 1 specialization of this function. The other 6 functions are also translated by Dexter successfully.

6.3 Translation Performance

In this section, we discuss two experiments to evaluate the effectiveness of Dexter's three-stage search algorithm. STNG, which lifts Fortran code to Halide, uses monolithic search combined with traces generated by symbolic execution to limit the search space. Though not a direct comparison between the systems (since Dexter does not use execution traces, and since the systems target different front-end languages), the experiments in this section compare the monolithic search strategy of STNG against the modular search of Dexter.

For our first experiment, we compare the performance of Dexter's modular search against naive monolithic search over a set of 5 library functions using the same synthesizer. We hand-picked the simplest functions to give naive search the best chance of completion. Each of the five benchmarks were successfully translated in less than 10 seconds by our three-stage algorithm, whereas monolithic search timed out after 15 minutes for all five functions. This demonstrates a speedup of roughly 100× in synthesis time.

Our second experiment aims to investigate how uniformly the synthesis problem is partitioned. To do so, we reviewed the amount of time spent in each of the three stages of synthesis. Across the 264 benchmarks that were successfully translated, Dexter spent 23% of the translation time during the first stage, 34% of the time during the second stage and the remaining 43% of the time was spent during the third stage, showing that dividing the original synthesis problem into three parts improves overall search efficiency.

6.4 Runtime Performance

To demonstrate the possible benefits of applying Dexter to existing image processing code, we compare the performance of the original reference code to the generated Halide code. We apply the newest Halide autoscheduler [Adams et al. 2019] to each translated pipeline; this newest autoscheduler uses a combination of learned models and auto-tuning to obtain the best performance. For our x86 testbed, we allow the autoscheduler to explore 32 potential schedules and utilize the best-performing one. Because the current tooling for the autoscheduler does not support exploring GPU schedules or executing on iPads [5], we allow the autoscheduler to use a model augmented by obtained performance on the x86 candidates. We do not argue that these are the best schedules, but they give a sense of what kind of performance can be obtained fully automatically.

In Photoshop, low-level image operations are called on image tiles of a fixed size, instead of on the entire image or layer. The default tile size is 1024×1024, and, depending on the operation, tiles may be interleaved or planar. In the actual application, a separate subsystem subdivides tiles among different threads for parallel execution; for these experiments, we run the original code in isolation, without parallelism, since the operations themselves do not contain parallel directives[6]. We measure performance when the tile is present in cache, as this scenario is the most common in Photoshop.

Figure 11 shows the performance improvements from applying Dexter to Photoshop image processing source code[7]. The median performance improvement on our x86 test machine is 7.03×, with 70% of all benchmarks achieving a speedup of at least 2×. While compilers are able to vectorize some of these functions automatically, the use of Halide enables a much more efficient vectorized and parallelized schedule with little effort, demonstrating the usefulness of Dexter in bringing the benefits of Halide to legacy image processing code.

6.4.1 Porting to Different Architectures. Photoshop currently only runs on Intel architectures, so for this experiment we demonstrate the usefulness of porting legacy code to Halide automatically to enable cross-platform performance. For our ARM testbed, the median speedup is 4.52×. In some cases, especially for interleaved benchmarks, Figure 11 shows that the autoscheduler chooses a suboptimal schedule; this could easily be fixed by changing just a line or two in the generated schedule, unlike the case when hand-optimizing C++ code. To test this, we explored why some of the benchmarks are more than 10× slower, and discovered that the autoscheduler attempted to parallelize over the channel dimension for the interleaved benchmarks; in essence, this schedule forces fine-grained synchronization between cores by sharing cache lines. By writing

[4] Git commit id 139a4a95

[5] Apple requires code signing for execution, and the current tooling for the autoscheduler does not implement this requirement.

[6] Thus, within Photoshop, the performance is often higher than shown here. However, the purpose of this experiment is to show runtime performance of translated kernels, not to compare against Photoshop performance.

[7] Because some translated functions are difficult to test in isolation (e.g. they are leaf operations in a multi-step pipeline), we show performance results for only functions with unit tests that execute them in isolation (88% of the test set).

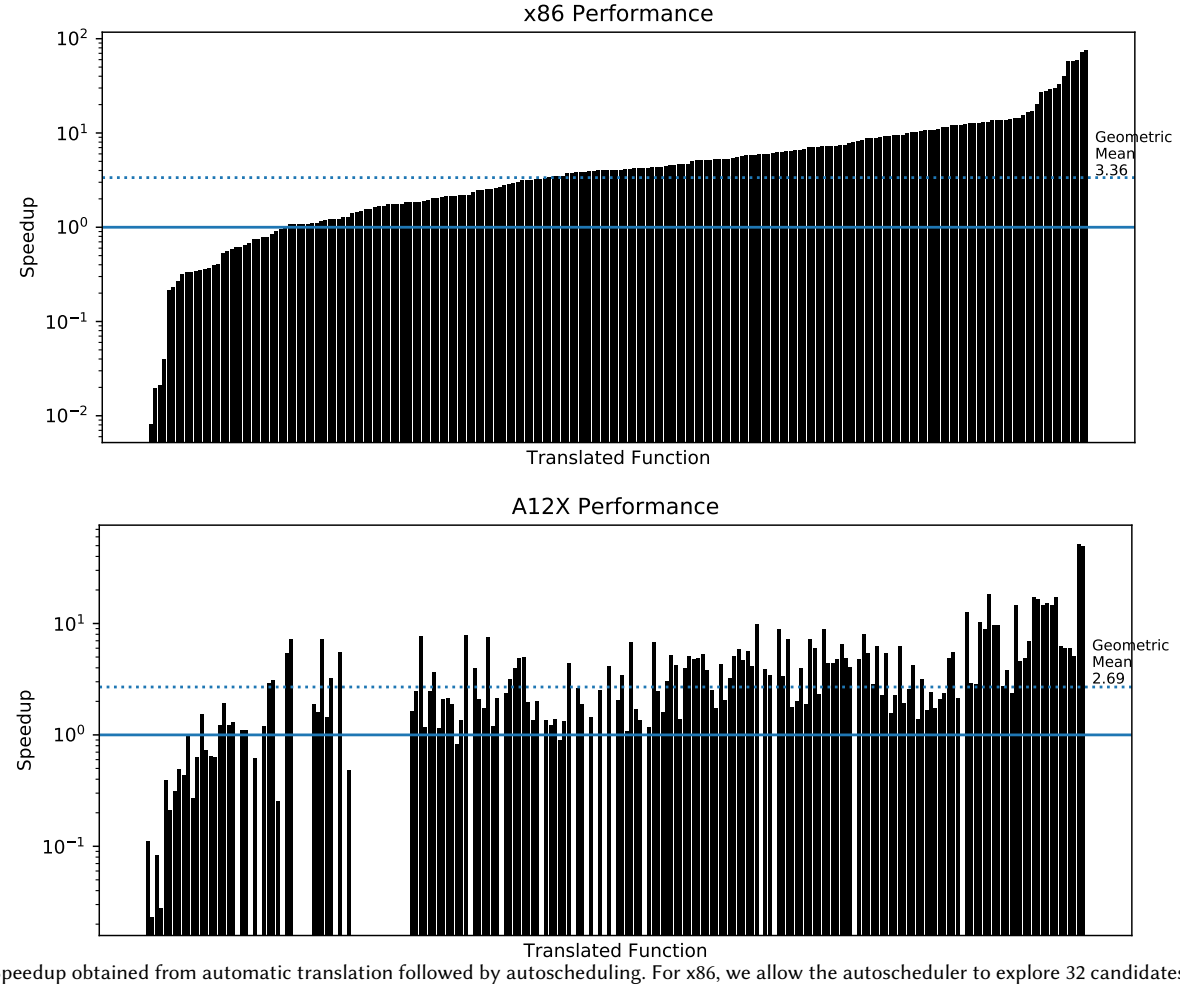

Fig. 11. Speedup obtained from automatic translation followed by autoscheduling. For x86, we allow the autoscheduler to explore 32 candidates, while for ARM we allow the autoscheduler to augment its model using the x86 performance, but do not explore multiple schedules. Benchmarks are ordered by their x86 speedup; the ARM chart shows fewer benchmarks because we cannot execute hand-written SSE for comparison purposes.

a simple schedule by hand, we obtain 1.4–4.4× speedups for these benchmarks.

6.5 Composing Translated Functions into Pipelines

A document in Photoshop, like in many image processing programs, consists of *layers* of image data, which are composited together using *blend modes* and filters. The overall document can be thought of as a directed acyclic graph (DAG), with multiple source nodes representing the layer data, intermediate nodes representing blending or filtering operations, and a single sink node representing the final, composited document. In Photoshop, each intermediate node calls multiple low-level routines from the different benchmark suites. In this section, we construct a simple document-specific Just-In-Time (JIT) compiled pipeline that performs the full composite for a small document, and compare its performance with calling the translated (and optimized) routines individually. In both cases, we utilize the autoscheduler for optimization.

The document consists of two layers that are blended together using a normal blend (which calls three different translated functions), and then blurred using a radius of one. Calling individual routines yields a performance of 1.02 Gpixels/sec, while a combined function obtains 3.13 Gpixels/sec, a speedup of over 3×. This speedup is almost completely due to avoiding unnecessary memory traffic, since no intermediates need to be written to memory. We anticipate that JIT-compiling document-specific pipelines for GPUs will yield even greater speedups. This demonstrates that translation to Halide opens up new possibilities for optimization that would be difficult using the legacy C++ code, and Dexter enables such optimization by automatically translating legacy C++ code into Halide.

7 CONCLUSION

In this paper, we presented Dexter, a tool that automatically translates image processing functions written in C++ to the Halide DSL. Unlike traditional compilers, Dexter uses a novel domain-specific synthesis algorithm to infer the summaries from the input image processing operations. Our prototype can translate many real-world image processing operations, and the translated code performs significantly better when compared to the original implementation.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants IIS-1546083, IIS-1651489, and OAC-1739419; DARPA award FA8750-16-2-0032; DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Adobe, Amazon, Google, Huawei, and NVIDIA.

REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, MichaÃńl Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019).

Adobe. 2010. Pixel Bender Language Reference. https://www.adobe.com/devnet/archive/pixelbender.html

Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1205–1220. https://doi.org/10.1145/3183713.3196891

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Rastislav Bodík and Barbara Jobstmann. 2013. Algorithmic program synthesis: introduction. *International Journal on Software Tools for Technology Transfer* 15 (2013), 397–411.

Pedro Boechat, Mark Dokter, Michael Kenzel, Hans-Peter Seidel, Dieter Schmalstieg, and Markus Steinberger. 2016. Representing and scheduling procedural generation using operator graphs. *ACM Trans. Graph.* 35, 6 (2016), 183:1–183:12.

Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. 2009. SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization. In *PMEA*.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2491956.2462180

L. Dalcin, R. Bradshaw, K. Smith, C. Citro, S. Behnel, and D. S. Seljebotn. 2010. Cython: The Best of Both Worlds. *Computing in Science & Engineering* 13 (09 2010), 31–39. https://doi.org/10.1109/MCSE.2010.118

Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Brian Guenter and Diego Nehab. 2010. *The Neon Image Processing Language*. Technical Report. Microsoft Research. https://www.microsoft.com/en-us/research/publication/the-neon-image-processing-language/

Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 13–24.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330.

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144:1–144:11.

James Hegarty, Ross G. Daly, Zachary DeVito, Mark Horowitz, Pat Hanrahan, and Jonathan Ragan-Kelley. 2016. Rigel: flexible multi-rate image processing hardware. *ACM Trans. Graph.* 35, 4 (2016), 85:1–85:11.

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. 2012. Portable Parallel Performance from Sequential, Productive, Embedded Domain-specific Languages. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 303–304. https://doi.org/10.1145/2145816.2145865

Bruno Cardoso Lopes and Rafael Auler. 2014. *Getting Started with LLVM Core Libraries*. Packt Publishing, Birmingham, UK.

Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2737924.2737974

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (2016), 83:1–83:11.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 297–310. https://doi.org/10.1145/2872362.2872387

Thomas Porter and Tom Duff. 1984. Compositing Digital Images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. ACM, New York, NY, USA, 253–259.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. ACM, Seattle, WA, USA, 519–530.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 53–64.

Armando Solar-Lezama. 2019. Sketch Synthesizer. https://people.csail.mit.edu/asolar/. Accessed on: 2019-01-11.

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 167–178.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1631–1634. https://doi.org/10.1145/3035918.3058738

Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *ABZ (Lecture Notes in Computer Science)*, Vol. 10817. Springer, 105–120.

Yuting Yang, Sam Prestwood, and Connelly Barnes. 2016. VizGen: accelerating visual computing prototypes in dynamic languages. *ACM Trans. Graph. (TOG)* 35, 6 (2016), 206:1–206:13. http://dl.acm.org/citation.cfm?id=2982403

A SYNTHESIS OPTIMIZATIONS

A.1 Symmetry Elimination and Memoization

The space of possible expressions encoded by the grammar in Figure 4c contains a large amount of symmetry: it can generate syntactically different expressions that are semantically equivalent due to the presence of commutative and associative operations (for instance, $a * (b + c) \equiv a * b + a * c$). Furthermore, larger functions frequently contain recurring sub-expressions, especially across different branches of control flow. A naive search over this grammar would consider far too many redundant expressions with the same semantics. Therefore, Dexter, inspired by prior work in relational algebra [Wang et al. 2018], uses a bottom-up expression generator that prunes away redundant expressions, while memoizing already generated sub-expressions for reuse.

The expression generator maintains a list of expressions, initially instantiated with the set of available terminals. To construct new expressions, the generator first chooses an operator from the set of operators available in the grammar. Next, it chooses operands for

	Input Terminals				Step 1	Step 2	Step 3	Step 4	Step 5
Trace 1	msk(i)	src1(i)	src2(i)	1	1 == msk(i)	src1(i) * src2(i)	src1(i) * src2(i) * msk(i)	(1==msk(i) ? src1(i) * src2(i) : src1(i) * src2(i) * msk(i))	-
Trace 2	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	src1(i) * src2(i) * msk(i)	(msk(i)==1 ? src1(i) * src2(i) : src1(i) * src2(i) * msk(i))	-
Trace 3	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	msk(i) * src1(i) * src2(i)	(msk(i)==1 ? src1(i) * src2(i) : msk(i) * src1(i) * src2(i))	-
Trace 4	msk(i)	src1(i)	src2(i)	1	msk(i) == 1	src1(i) * src2(i)	src1(i) * src2(i)	msk(i) * src1(i) * src2(i)	(msk(i)==1 ? src1(i) * src2(i) : msk(i) * src1(i) * src2(i))

Fig. 12. Four possible traces of Dexter's expression generation algorithm. The steps in green indicate successful termination of the algorithm, whereas the steps in red indicate a pre-emptive rejection of the trace due to violation of symmetry breaking rules.

the operator from the list of expressions to construct a new expression. Finally, the generator checks whether this newly generated expression is the correct expression for the summary by invoking the solver. If so, the algorithm terminates and the expression is returned. If it does not verify, the expression is memoized by appending it to the end of the list and the process is repeated until the correct expression is found.

We illustrate our algorithm with an example. Suppose our goal is to generate the following expression: $(msk(i)\ ==\ 1\ ?\ src_1(i) * src_2(i)\ :\ src_1(i) * src_2(i) * msk(i))$. In this expression, src_1 and src_2 represent two layers that we want to blend, and msk is the blend mask. Figure 12 shows four possible traces (decision paths) of our algorithm for producing semantically equivalent expressions. Not all traces are viable as they contain steps violating our symmetry breaking rules (marked in red), which we explain later. To demonstrate our algorithm, we walk through trace 3, which successfully generates this expression in four steps. In step 1, the algorithm combines terminals $msk(i)$ and 1 using the equality operator to construct a boolean expression, but finds that the generated expression is not the desired expression. In step 2, it combines $src_1(i)$ and $src_2(i)$ using multiplication. In the third step, the expression generated in step 2 $(src_1(i) * src_2(i))$ is combined with the terminal $msk(i)$ using multiplication to get the alternate expression. The fourth and final step uses these three generated expressions as operands to the ternary operator to construct the desired output expression.

There are several benefits of Dexter's memoization approach. First, it maintains a total ordering over the generated expressions, based on the list index at which they are stored. This is useful for eliminating symmetries in the search space: for commutative and associative binary operators, the expression generator only allows binary expressions $e_1\ op\ e_2$ where e_1 is stored at a lower index in the expressions array than e_2, and likewise for the test, consequent, and alternate expressions used in a conditional. To see the benefit, consider traces 1, 2, and 3 in Figure 12. All three traces generate semantically equivalent expressions, yet since they are syntactically different, the synthesizer would enumerate all three in its search. With our order-based pruning constraints, however, both trace 1 and trace 2 would be rejected as they violate the constraints at step 1 and step 3 respectively. Furthermore, since the generated sub-expressions are stored in the list, they can be reused subsequently. This reduces the number of steps the synthesizer must take to generate expressions that contain recurring sub-expressions. This is illustrated in trace 4 in Figure 12. This trace also generates the correct expression, but since it does not reuse the sub-expression $src_1(i) * src_2(i)$, the algorithm requires an extra step to build the expression compared to trace 3.

A.2 Analysis Based Suggestions

The optimizations discussed so far are not particular to a specific input kernel. Dexter also analyzes the input code to populate the starting list of expressions with input-specific recommendations to the synthesizer. For example, in the blur kernel, static analysis can extract that the value being written into the `dst` array is `(tmp[c-rowBytes] + tmp[c] + tmp[c+rowBytes]) / 3`. By substituting our synthesized terminal mappings, we can get the equivalent IR expression: $(tmp(x;y-1) + tmp(x;y) + tmp(x;y+1))$ / 3. Dexter therefore adds this expression as one of the initial expressions to consider during synthesis. If the suggestion is correct, or is a sub-expression of the correct expression, the synthesizer can construct the result in fewer steps. If the suggestion is incorrect, the synthesizer can simply ignore it and construct the correct expression using the set of terminals. The overhead of providing these recommendations is minimal and the benefits of a correct recommendation are significant (over 100x faster synthesis).