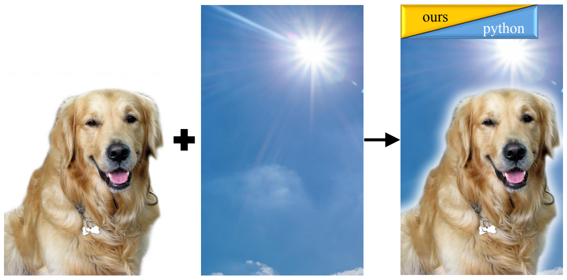
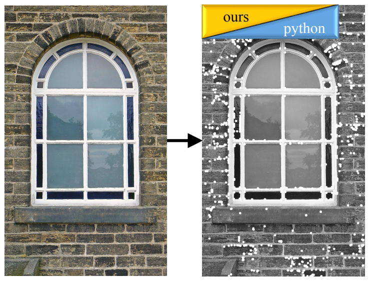
To appear in ACM TOG 35(6).

VizGen: Accelerating Visual Computing Prototypes in Dynamic Languages

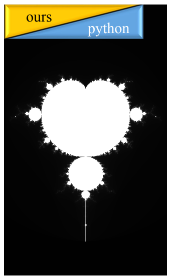
Yuting Yang[1] Sam Prestwood[1] Connelly Barnes[1]
[1]University of Virginia



(a) Composite (RGB), 1732x Speedup



(b) Harris corner, 1223x Speedup



(c) Mandelbrot, 266x Speedup

Figure 1: *Three applications that were compiled with our compiler, which is specialized for visual computing programs. Image compositing (a) gains a 1732× speedup over the equivalent code in the Python interpreter. Our result is shown in the diagonal upper-left corner, and the Python result is shown in the diagonal lower-right corner: notice there is no difference between the two images. A Harris corner detector (b) and a Mandelbrot fractal renderer (c) likewise gain significant speedups after being compiled with our compiler, relative to Python.*
Photo credits: (a) Karen Arnold and Skitterphoto, (b) modified from original by Tim Green.

Abstract

This paper introduces a novel domain-specific compiler, which translates visual computing programs written in dynamic languages to highly efficient code. We define "dynamic" languages as those such as Python and MATLAB, which feature dynamic typing and flexible array operations. Such language features can be useful for rapid prototyping, however, the dynamic computation model introduces significant overheads in program execution time. We introduce a compiler framework for accelerating visual computing programs, such as graphics and vision programs, written in general-purpose dynamic languages. Our compiler allows substantial performance gains (frequently orders of magnitude) over general compilers for dynamic languages by specializing the compiler for visual computation. Specifically, our compiler takes advantage of three key properties of visual computing programs, which permit optimizations: (1) many array data structures have small, constant, or bounded size, (2) many operations on visual data are supported in hardware or are embarrassingly parallel, and (3) humans are not sensitive to small numerical errors in visual outputs due to changing floating-point precisions. Our compiler integrates program transformations that have been described previously, and improves existing transformations to handle visual programs that perform complicated array computations. In particular, we show that dependent type analysis can be used to infer sizes and guide optimizations for many small-sized array operations that arise in visual programs. Programmers who are not experts on visual computation can use our compiler to produce more efficient Python programs than if they write manually parallelized C, with fewer lines of application logic.

CR Categories: I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques; I.4.9 [Computing Methodologies]: Image Processing and Computer Vision—Applications;

Keywords: Compilers, imaging, computational photography

1 Introduction

Visual datasets are rapidly growing in size, due to the widespread use of cell-phone cameras, as well as video and photo sharing sites such as YouTube and Facebook. For example, approximately 60 hours of video are added to YouTube every minute [Syed-Abdul et al. 2013] and 350 million photos are uploaded to Facebook every day [Quo]. We believe that these trends will continue to strengthen, due to forces such as increasing photograph resolutions and increased worldwide Internet adoption. To scientifically model, engineer, and derive knowledge from such datasets, it is necessary to develop efficient codes to process visual data. For the purpose of this paper, we define *visual computing* broadly as computation over these datasets, using graphics and vision.

However, there is currently a tension when developing programs that work with visual datasets. This tension is between the run-time efficiency of the final programs and easy exploratory prototyping.

At one extreme, if maximum efficiency is desired, then visual computing programs can frequently be optimized to run even faster than a naive C implementation. This can be done by hiring an engineer who is a domain expert, who can craft a highly efficient program. However, this efficiency typically comes at the cost of greater code complexity, less portability, and less maintainability [Ragan-Kelley et al. 2013]. Another alternative is to use a domain-specific language. For example, the Halide domain-specific language [Ragan-Kelley et al. 2013] can be used for computational photography applications, but it is not suitable for expressing all graphics or vision applications, nor is it Turing complete. As a result, the most fruitful optimizations currently either require a program that fits in a restricted domain-specific language (DSL) such as Halide's, or require hiring domain experts to develop complicated platform-specific optimizations.

At the other extreme, a scientist or developer might choose to work in a language that facilitates rapid prototyping, such as Python or MATLAB. These "dynamic" languages have a flexible runtime model and dynamic typing, which can be beneficial for quick iterative design. However, normally the resulting programs run slowly because such languages typically incur large overheads in run-time, and these are particularly acute in graphics and vision problems. This is due to low-level implementation details that must be handled by the language's interpreter or compiler, including issues such as

```python
def blur(input, output):
    temp = np.zeros(input.shape)

    for r in range(input.shape[0]-2):
        for c in range(input.shape[1]):
            temp[r, c] = (input[r, c] + input[r + 1, c] + input[r + 2, c]) / 3.0

    for r in range(input.shape[0]-2):
        for c in range(input.shape[1]-2):
            output[r, c] = (temp[r, c] + temp[r, c + 1] + temp[r, c + 2]) / 3.0
```

Python blur program

```c
void blur(IMG_3CHANNEL *input, IMG_3CHANNEL *output) {
    int r, c, k;
    IMG_3CHANNEL *temp = zeros_rgb(img->rows, img->cols);

    #pragma omp parallel for private(c)
    for(r = 0; r < img->rows - 2; r++) {
        for(c = 0; c < img->cols; c++) {
            temp->data[r][c][0] = (input->data[r][c][0] + input->data[r + 1][c][0] + input->data[r + 2][c][0]) / 3.0f;
            temp->data[r][c][1] = (input->data[r][c][1] + input->data[r + 1][c][1] + input->data[r + 2][c][1]) / 3.0f;
            temp->data[r][c][2] = (input->data[r][c][2] + input->data[r + 1][c][2] + input->data[r + 2][c][2]) / 3.0f;
        }
    }

    #pragma omp parallel for private(c)
    for(r = 0; r < img->rows - 2; r++) {
        for(c = 0; c < img->cols - 2; c++) {
            output->data[r][c][0] = (temp->data[r][c][0] + temp->data[r][c + 1][0] + temp->data[r][c + 2][0]) / 3.0f;
            output->data[r][c][1] = (temp->data[r][c][1] + temp->data[r][c + 1][1] + temp->data[r][c + 2][1]) / 3.0f;
            output->data[r][c][2] = (temp->data[r][c][2] + temp->data[r][c + 1][2] + temp->data[r][c + 2][2]) / 3.0f;
        }
    }
}
```

C blur program

Figure 2: *An example of a naively written two-stage blur program in Python (left), and C (right). The Python program is significantly more concise, and also more general: it can handle either grayscale or 3-channel input images, whereas the C program only handles 3-channel input images. Without using our compiler, the Python program is orders of magnitude slower than the C program. When our compiler is used in "approximating" mode, where it can rewrite 64-bit floats to 32-bit, the Python blur is 6× faster than the C program on a four-core machine. In non-approximating mode, the Python program is still 3× faster for both 32-bit and 64-bit precision. See Section 9 for results.*

boxing of types, heap-allocated variables, garbage collection, array allocation, and so forth. Specialized compilers for dynamic languages such as just-in-time (JIT) compilers have been developed, which can sometimes ameliorate these problems [Bolz et al. 2009] [Lam et al. 2015]. However, even when these compilers are used, visual computing codes that perform fine-grained looping or complicated array calculations in "dynamic" languages such as Python or MATLAB can still be orders of magnitude slower than C code (see e.g. our results in Section 9). This means that novice or non-expert developers cannot easily develop efficient visual computing programs, and even experts may struggle with either maintaining non-portable and highly optimized codes, or else choosing restrictive DSLs.

This paper proposes the following goal to the research community and takes steps towards achieving it: *it should be possible to automatically translate visual computing prototypes in dynamic languages into highly-performant code.* We believe that this goal is a worthy one for two reasons. First, dynamic languages are widely used in academic and industrial research. If one can automatically translate mock-ups into efficient code then this will lower the costs of technology transfer, increase application interactivity, and accelerate development. Second, being able to make such an automatic translation could help democratize visual computing by making it more accessible for novice and non-expert programmers. by permitting them to express efficient programs in easy-to-learn and simple languages.

Our compiler framework is based on three key properties of visual computing programs, which permit optimizations. These are: (1) many arrays have small, constant, or bounded size, (2) many operations on visual data are supported in hardware or are embarrassingly parallel, and (3) humans are not sensitive to small numerical errors in visual outputs due to changes in floating point precision.

Based on the previous key properties, we contribute a system that overcomes common performance challenges encountered when compiling visual programs in dynamic languages. We integrate program transformations that have been described previously, and improve these existing transformations to handle visual programs that perform complicated array computations. Visual programs tend to interleave many computations with both small and large arrays across function calls, and at different loop nesting depths. These operations tend to create many performance issues when compiled in a simplistic manner. For example, a naively compiled program could reallocate arrays within each loop iteration, use arbitrary size array constructors, fail to use the constant bounds to accelerate loops, or fail to use efficient vector code. Our system addresses these concerns by optimizing small and large array computations at multiple granularities of nesting and call sites. Our optimizations work together to rewrite operations over and within arrays to be more compute efficient.

One important property that we leverage in our compiler is the expressive power of matrix and vector notation in programs. We have found that not only is matrix and vector notation frequently more concise and general than the corresponding scalar code, but it also frequently permits greater optimization, since it is specified at a high level of abstraction. As a simple illustrative example, we show a two-stage separable blur program in Figure 2(left), which is written in Python, over pixels that can either be scalars or vectors. This code conveniently remains unchanged regardless of whether the pixels are grayscale, RGB, or alpha-premultiplied RGBA. Our compiler is able to specialize for constant numbers of color channels within this representation, and apply further optimizations such as changing array memory layout to better facilitate vectorization. We do this by using dependent types to infer both type and array size information.

We developed a prototype compiler implementation for the Python language and tested it on 12 visual computing applications. As a motivating example, consider again the simple blur example in Figure 2, which includes code in both Python and C. The Python program is significantly shorter, and when used with our compiler, is 3 to 6× faster than the C program. Our full results in Section 9 show a median speedup of orders of magnitude over the Python interpreter and Python JITs such as Numba and PyPy. This is sufficient to speed up many programs that originally were slower than real-time to interactive or real-time. Our applications were written by two students who are not domain experts. On average, the applications built with our compiler are both shorter (2.3×) and faster (2.4×) than their equivalents implemented in manually parallelized C. We use an autotuner, which is guided by an indicative workload, to discover the best set of optimizations. Thus, there is no annotation burden or other modification of the input program required. Currently, our compiler only accepts the Python language as input, however, the transformations we have developed are not specialized for Python, and therefore could also apply to other languages.

2 Related work

In this section, we will first discuss related work on image processing languages, followed by more general software engineering topics such as stencil optimizations, profile-guided optimization, Python compilers, and dependent types.

2.1 Image processing languages

The Halide [Ragan-Kelley et al. 2013] language is a domain-specific language (DSL) that permits image pipelines to be optimized by separating the *algorithm* and the *schedule*. The algorithm describes *what is to be computed*, and the schedule describes *how it should be computed*, in terms of fusing or inlining computation stages, storing or recomputing intermediate values, parallelism, and so forth. Our compiler also permits scheduling choices by a search over different optimizations, which we simply call program transformations. Halide is restricted to image pipelines which can be modeled as purely functional computations over a sequence of dense intermediate arrays, and therefore it cannot express many imperative programs, and it is not Turing complete. In contrast, our compiler accepts as input a general-purpose dynamic programming language, and then places domain expertise in the transformation rules used, by specializing these rules for visual computing. Other than Halide, polyhedral optimization has been used to generate efficient GPU code from image pipelines [Cornwall et al. 2009]. Polyhedral optimization has also been used in PolyMage [Mullapudi et al. 2015], which compiles an image processing DSL similar to Halide. Both Halide [Mullapudi et al. 2016] and PolyMage have recently developed model-driven approaches to determine the best schedules or optimizations. Earlier image processing languages focused on simpler optimizations, such as fusing only simple image stages without stencils [Elliott] [Shantzis 1994]. The Terra [DeVito et al. 2013] language uses multi-stage programming in conjunction with the dynamically-typed language Lua, and has also been used to explore image processing applications.

2.2 Stencil optimizations

Stencil codes are iterative computations over an array, where each computed value depends on a fixed surrounding region, called the *stencil*. Uses include solving partial differential equations, image processing, and other scientific applications. Stencils have been extensively studied; we briefly review some work in this space. Efficient cache-oblivious stencil computations were developed by Frigo and Strumpen [2005]. The Pochoir compiler transforms serial stencil computations into an efficient parallel cache-oblivious computation [Tang et al. 2011]. Stencil computations can be *tiled* [Krishnamoorthy et al. 2007], which typically improves cache efficiency, but depending on whether earlier stages are inlined, may introduce redundant computation along tile boundaries. Compiler researchers have generated efficient CPU and GPU code for stencils via tiling [Holewinski et al. 2012] [Krishnamoorthy et al. 2007]. In our compiler, we instead focus on program transformations that are specialized for visual computing in dynamic languages.

2.3 Profile-guided optimization

Optimizing compilers use semantics-preserving transformations that may or may not improve overall performance. For example, inlining functions on a frequently-visited path may improve run-time performance, while inlining functions on a rare path may have little effect on performance. One solution to this problem is to record a trace or *profile* of run-time information on an indicative workload. The efficient gathering (e.g., [Ball et al. 1998] [Graham et al. 1982]) and use (e.g., [Ammons and Larus 1998]) of such profile information is a long-studied subfield. Many modern compilers ship with some degree of support for profile-guided optimization (e.g., GCC's -fprofile-generate, LLVM's -fprofile-instr-generate, etc.). Useful profiles can even be approximated statically [Buse and Weimer 2009]. These approaches traditionally gather profile information and then choose optimizations or parameters: a two-step process. By contrast, our compiler utilizes an iterative feedback loop in which multiple candidate optimizations are separately evaluated in terms of their relative performance. We call this an "autotuner." In that light, our technique is closer to the "Fastest Fourier Transform in the West" adaptive tuning architecture [Frigo and Johnson 1998] or the ATLAS project [Whaley et al. 2001]. Recent works have focused on tuning programs using multiple search strategies [Ansel et al. 2014], and automatically tuning programs for better parallelism [Morajko et al. 2007; Karcher and Pankratius 2011]. More generally, this is an area of search-based software engineering [Harman and Jones 2001].

2.4 Python compilers

Our current compiler implementation accepts the Python language as input. There have been a number of compilers developed for the Python language. These include early compilers that do not feature array support, such as StarKiller [Salib 2004] and Shed-Skin [Dufour and Coughlan 2013], which are not well-suited for visual computing. More recent compilers do feature array support. These include HOPE [Akeret et al. 2015], unPython [Garg and Amaral 2010], the Numba just-in-time (JIT) compiler [Lam et al. 2015], PyPy [Bolz et al. 2009], and Pythran [Guelton et al. 2015]. The Nuitka[1] compiler does not currently do type inference and so is not particularly suited for performance-intensive visual computing. HOPE focuses on astrophysical simulations. We compare against the other compilers, and find they either generate code that is frequently orders of magnitude slower than ours for visual computing programs, or else they cannot accept our input programs due to them falling outside the compiler's targeted domain. The speedups found by our compiler are because we take advantage of domain knowledge in visual computing. In particular, we use dependent type analysis to determine small, constant size arrays, and then accelerate small array operations using a variety of strategies. The Cython [Behnel et al. 2011] language permits C type and parallelism annotations to be added to Python. Our current compiler outputs code in Cython. This helps simplify the design of the compiler, because the target language is similar to the source language.

2.5 Dependent types

A dependent type is a type whose definition depends on a value. For example, consider a function `zeros(n)` that accepts an integer argument n and returns a length n array of floats $[0.0, 0.0, \ldots, 0.0]$. This function could certainly be said to have `array` type, but because the length of the array is not specified in the type, this will not permit many useful optimizations. More usefully, we could say that `zeros(n)` has a dependent type, where the return type is "the set of arrays of length n," and n is the value passed in to the function. We use inference of dependent types to deduce constant array bounds throughout visual programs, as described in Section 4.1. Dependent types have been used to eliminate array bounds checks by inferring array sizes at compile time [Xi and Pfenning 1998], increase the safety of low-level C programs by using bounded pointers [Condit et al. 2007], as well as increase the expressiveness of the type system [Augustsson 1998; Xi and Pfenning 1999]. We specifically focus on using dependent types to infer array sizes throughout visual programs.

3 Overview

This section gives an overview of our system. We first discuss some key properties of the input programs that our compiler assumes.

[1]`http://nuitka.net/`

Next, we give an overview of the different components of our system, and relate these back to the key properties.

3.1 Key properties and assumptions

Visual computing programs have three key properties that we exploit to develop domain-specialized compiler optimizations:

1. Many data structures, such as lists and arrays, have **small, bounded, or constant size**. If array size information is inferred throughout the input prototype program, then the compiler can use optimizations such as constant folding, stack allocation, pre-allocating arrays only once, and loop unrolling.

2. Many operations on visual data are **embarrassingly parallel** or are **supported in hardware**. Loops in visual programs frequently have no dependencies, and so they can be trivially parallelized. Operations on pixels, vertices, and small matrices are supported in hardware. For example, one such hardware-supported optimization is *Simultaneous Instruction Multiple Data* (SIMD) vectorization.

3. In some cases, humans are not sensitive to very small errors in the output, such as a 0.001% error in colors of an image. In our case, if the programmer enables an "approximation" mode, then the floating-point precision in computations can be optimized by the compiler, to gain higher efficiency at the cost of lower precision.

Our current compiler uses profile-guided optimization, and therefore a key assumption that we make is that the input program can be run on an indicative workload. For our applications, this workload simply consists of a one-line call of the user's program on a representative set of parameters and/or input images. For efficiency, our compiler assumes that the input program does not use dynamic execution of string programs (using functions such as `eval()`), or raise unhandled exceptions. Within these constraints, the compiler preserves correctness, when the "approximation" mode is not used.

3.2 Overview of system components

An overview of the components of our system is shown in Figure 3. Our system has three main sub-parts: static analysis components, program transformations, and the autotuner. The compiler first applies *static analysis* components (discussed in Section 4), and then applies optimizations as a sequence of *program transformations* (discussed in Section 5). *Static analysis* is an analysis of a computer program that does not require running the program. A *program transformation* is an operation that transform an input program, yielding an output program. After discussing these components, in Section 6, we next present the autotuner, which automatically selects program transformations that make the input program efficient. Finally, we evaluate a suite of Python test applications in Sections 8 and 9. Throughout the document, we have hyperlinked components to appropriate sections. We next briefly summarize the static analysis components and the program transformations.

The static analysis components are:

S1. *Type and size inference* uses dependent type analysis to infer types, including constant sizes for arrays. This is related to key property (1);

S2. *Parallelism analysis* determines whether a loop can be safely parallelized. This is related to key property (2); and

S3. *Preallocation analysis* determines whether a array's storage can be preallocated. This is related to key property (1).

The program transformations are:

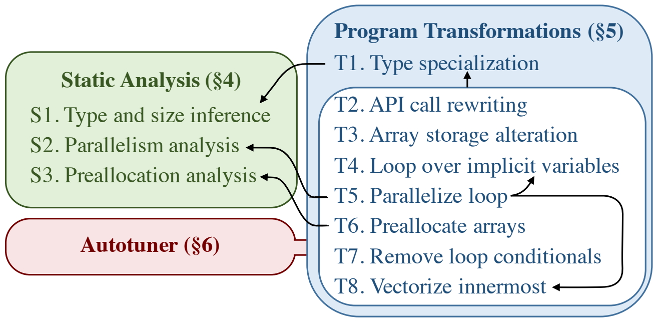

Static Analysis (§4)
S1. Type and size inference
S2. Parallelism analysis
S3. Preallocation analysis
Autotuner (§6)
Program Transformations (§5)
T1. Type specialization
T2. API call rewriting
T3. Array storage alteration
T4. Loop over implicit variables
T5. Parallelize loop
T6. Preallocate arrays
T7. Remove loop conditionals
T8. Vectorize innermost

Figure 3: *An overview of our system showing its components and their dependencies. There are three categories of components: static analysis components (S1, ..., S3, shown at top left), program transformations (T1, ..., T8, shown in right), and the autotuner (shown in bottom left). Arrows indicate dependencies between components: an arrow from A to B indicates that A depends on B. The program transformations T2-T8 depend on type specialization (T1), so the smallest arrow in the top-right indicates a multiple (block) dependency. See Section 3.2 for summary descriptions of these components.*

T1. *Type specialization* assigns static types for variables, which permits greater efficiency since dynamic type information does not need to be tracked at run-time;

T2. *API call rewriting* rewrites calls to common built-in and API functions to equivalents that are more efficient on the target hardware. This transformation is related to property (2);

T3. *Array storage alteration* modifies array types or shape to be more efficient. This is related to key properties (2) and (3);

T4. *Loop over implicit variables* fuses array operations into a single set of nested loops. This is related to key property (1);

T5. *Parallelize loop* converts a for loop into a multi-threaded parallel loop. This is related to key property (2);

T6. *Preallocate arrays* allocates array storage only once to decrease the burden on the memory manager and increase cache performance. This is related to key property (1);

T7. *Remove loop conditionals* accelerates innermost loops by removing bounds checks with the help of a theorem prover. This is related to key property (2); and

T8. *Vectorize innermost* applies SIMD vectorization to certain array operations. This transformation is related to property (2).

The dependency graph in Figure 3 indicates that some transformations have dependencies, which require that other transformations or static analysis components be applied first. For example, before vectorization (T8) can be applied, type specialization (T1) must have first been applied.

Our compiler develops a variety of different transformations, some of which are generic, and are present in other compilers. The emphasis of our paper is therefore on the more novel transformations, such as our focus on small-size arrays, dependent type analysis to determine array sizes, the "approximating" modification of 64-bit floats to 32-bit floats, and the beneficial interaction between these different components in dynamic programs. Type specialization and API call rewriting are common in JITs [Bolz et al. 2009; Lam et al. 2015]. Preallocation of arrays, removing conditionals, and vectorization have been used in PolyMage [Mullapudi et al. 2015] and Halide [Ragan-Kelley et al. 2013]. The extraction of vectorizable workloads from arbitrary R programs has been explored in Riposte [Talbot et al. 2012]. Array expression fusion has been used with similar effect to our "loop over implicit variables" transformation in the Julia language [Bezanson et al. 2012] (specifically, the devectorize macro), Theano [Bergstra et al. 2010], and other languages.

4 Static analysis

In this section, we discuss static analysis components. First, in Section 4.1, we show how dependent types can be used to statically infer types and array sizes throughout visual programs. Next, in Section 4.2, we discuss how static analysis can be used to determine whether to parallelize loops and preallocate arrays.

4.1 Type and array size inference

In dynamically typed languages, type inference can be used to identify known types throughout programs [Salib 2004]. These known types can be used to compile portions of a program to efficient code. One challenge we found in naively applying type inference is that vectors or matrices can easily be labelled as having unknown size or dimensionality. To address this challenge, we used a more sophisticated type inference strategy that identifies known small sizes of arrays throughout visual programs by using *dependent type analysis*.

As a motivating example, consider a small program that modifies an input color (r, g, b) in sRGB color space by multiplying it by a color matrix M. In pseudocode:

Algorithm 1 Color matrix program

```
1: function apply_color_matrix($r$, $g$, $b$)
2:     $n$ = 3
3:     $c$ = zeros($n$)
4:     $M$ = zeros(($n$, $n$))
5:     $c$[0] = $r$; $c$[1] = $g$; $c$[2] = $b$
6:     $M$[0, 0] = 1.05; $M$[1, 1] = 0.9; $M$[2, 2] = 1.0
7:     return matrix_mul($M$, $c$)
```

In the above program, c is a length 3 color vector, M is a 3x3 color matrix, the `zeros()` function builds either a 1D or 2D array of zeros depending on its argument's value, and `matrix_mul()` computes a matrix-vector product. In visual programs, such calculations involving small 2D, 3D, or 4D vectors or matrices are very common. This is because these vectors can be used to represent colors, vertices, directional quantities such as edge and ray vectors in Euclidean space, and linear transformations between these. These operations particularly tend to occur in the innermost loops of programs, where they are applied many times on pixel colors, vertices, or other elements of arrays. Dynamically-typed languages tend to represent arrays such as c and M using a generic multidimensional array type, for flexibility and generality. Because of this, dynamically-typed languages tend to be inefficient when executing such programs.

To make small array operations efficient throughout visual programs, we analyze *dependent types* throughout the program. Recall from our related work discussion that a dependent type is a type whose definition depends on a value. As a motivating example, in Algorithm 1, the `zeros()` function receives types that are either the integer n, or the tuple (n, n). Therefore, if we simply analyze only the types in the program, this will be insufficient to deduce that c is a length 3 vector or M is a 3x3 matrix. However, if we also analyze the values passed into the `zeros()` function, then we can discover that c has a *dependent type* of "length 3 array." Crucially, this dependent type depends on both the types and values supplied to `zeros()`. Likewise, we can discover that M has a dependent type of "3x3 matrix."

In general, dependent types can be fairly complicated to analyze, or even undecidable [Augustsson 1998]. Therefore, we implement only a simple dependent type analysis system that is well-suited for discovering small array bounds. This proceeds as follows. First, similar to most JITs, we determine all type signatures that are passed into functions, and produce a type-specialized variant of each function based on the types passed into it. Different type-specialized functions are produced for different small size array bounds: for example, specialized variants of a single blur function could be created for 2D grayscale image inputs, or a three-channel 3D array containing RGB colors. In our implementation, we rely on the indicative workload for the profile-guided optimization to determine type specializations, but this could also be done in a JIT compiler. For safety, if no type-specialized variant of a function is available, then the compiler falls back to a generic variant. Next, given these types, we perform a static analysis of dependent types.

The dependent type analysis proceeds by determining for every function call both the types and values of the arguments. The analysis is conservative so that programs are always correct: thus, arrays will labelled as having unknown size if their size is unknown, and any variable whose type cannot be inferred at all will be labelled as having "unknown" type (this could result in either a slower program or the programmer could manually annotate the type). To make sure the analysis is tractable, in our dependent type analysis, the values supplied are only those which are compile-time constants.

If a built-in or third-party module function is called such as the `zeros()` function, then it is supplied with all types and compile-time constant values. In the case of the `zeros()` function, type inference cannot proceed through the function, because the core functionality is implemented in C for efficiency. Thus, for such any builtin functions that cannot be analyzed through, if array bounds are to be inferred, then the compiler's library must supply an auxiliary "dependent type function." This dependent type function runs at compile-time. It is given the function's argument types and any argument compile-time constant values, and determines the dependent type returned by the function.

In our implementation, we have manually written dependent type functions for common builtin array routines. Functions where dependent types are particularly important include those that construct arrays from lists of numbers, or arrays that are uninitialized, filled with zeros, ones, or the identity matrix, as well as sums and products along given dimensions. This is because the return type of these functions depends on the argument values.

4.2 Static analysis of parallelism and preallocation

In addition to the dependent type analysis performed previously, we also statically analyze the input program to determine if loops can be parallelized and if arrays can be preallocated. As in the previous section, we perform such analysis conservatively to ensure the correctness of the compiled program.

Loop parallelism analysis. Many loops in visual computing programs are embarrassingly parallel. We use loop dependence analysis to independently determine whether each loop is trivially parallelizable. We use long-standing techniques for this that are similar to Polaris [Blume et al. 1996]: we determine whether there are no cross-iteration dependencies in loops, and identify whether there are any "loop private" variables (including arrays) that can safely be allocated in thread-local storage.

Preallocation analysis. Certain arrays such as temporary and output buffers can be pre-allocated once, rather than re-allocated with every execution of a procedure. We use similar ideas to existing work in this area, such as the preallocation analysis in SISAL [Cann and Evripidou 1995], which preallocated arrays that have a fixed maximal size. In our case, there are some additional subtleties because operations such as the "`zeros()`" function are often used to allocate arrays that represent say colors of an image. One step we have to take is perform a whole-program alias analysis to verify that no expressions that depend on an array would alias if the array is preallocated. Because data such as images could vary in size, we also check the size requirements of the "preallocated" array at the top of each function call, and dynamically increase the size if needed. Finally, initializing an array with zeros is slower than simply leaving the memory uninitialized, but the later is potentially unsafe. Therefore, we check if the range of array indices written to is either the full array, or larger than the range of array indices that are subsequently read from (the latter is proved by a theorem prover [De Moura and Bjørner 2008]). If this is the case, then an array need not be initialized.

5 Program transformations

In this section, we give a detailed explanation of each of the program transformations that were briefly mentioned in the overview (Section 3). We first discuss type specialization, since it is fundamental to the other transformations, and then discuss the other transformations in alphabetical order.

Type specialization. The static analysis stage identifies for each function, the set of input type signatures called for that function. For each function type signature, type specialization emits a specialized variant of that function with type declarations for all input, output, and local variables. The other transformations depend on type specialization, because type information is needed to make further optimizations. Type specialization also detects arrays with a constant size that is no larger than a maximum size T and rewrites these to be stack-allocated, so as to prevent unnecessary use of dynamic memory allocation (we use $T = 30$).

API call rewriting. In dynamic languages, simple API calls such as taking a dot product between vectors in $\mathbb{R}^3$ can generate highly inefficient code. This is because this may generate an API call for a dot product between vectors of arbitrary length. We observed that in visual programs, there is heavy use of linear algebra routines over vectors or matrices of two, three, and four dimensions. To improve the efficiency of such code, we detect when the dependent type of an array has a constant small size (2, 3, or 4), and use a typed macro facility to replace API calls such as tensor product with optimized C implementations of these. Currently we have implemented portable yet efficient C linear algebra routines such as norm, dot product, matrix-vector product, and length, as well as elementwise math operations such as clip, square, square root, absolute value, random, power, and so forth.

Array storage alteration. One goal of our compiler is to enable novice and non-expert programmers to easily write efficient code. However, some optimizations such as SIMD vectorization are highly useful for visual computing programs, but require detailed knowledge of low-level array storage formats. We attempt to shield programmers from needing to know these details, by developing a transformation that modifies array storage layouts throughout an entire graphical program. This transformation can optionally rewrite all double-precision arrays to single-precision, if the "approximation" mode is turned on. It can also rewrite all arrays ending with a dimension of length 3 to internally be stored such that the stride for the last dimension is 4 (so there is an unused extra value). The *stride of an array* is simply the number of array elements in the memory layout between successive elements along a given dimension. The rewriting from a stride 3 to 4 storage format facilities vectorization, and proceeds by modifying all color image read functions within this context to return stride 4 arrays, and by modifying all array operations between stride 3 and 4 arrays to recursively attempt to rewrite the shorter array to have stride 4.

Loop over implicit variables. It is common for array operations in dynamic languages to be expressed in a shorthand form that omits implicit variables. For example, if A, B, and C are vectors, matrices, or images of the same size, their average might be expressed as $D = (A + B + C)/3$. Typically, in such expressions, it is more efficient to calculate the result element-wise by fusing as much as possible of the computation. For instance, with a 2D array $D_{i,j}$, one could calculate directly $D_{i,j} = (A_{i,j} + B_{i,j} + C_{i,j})/3$. This is more efficient than calculating an intermediate result $R_1 = A + B$, computing a second intermediate array $R_2 = R_1 + C$, and then obtaining the final result $D = R_2/3$. This transformation simply fuses such computations and inserts loops over all implicit variables as needed. Furthermore, it is common in visual computing programs for arrays to be of known constant and small size. In this case, the transformation inserts known bounds for loops, thus further increasing efficiency. Unlike the previous three transformations, which are applied globally to the entire program, the loop over implicit variables is applied to a given program line.

Parallelize loop. The loop parallelization transformation may apply thread parallelism to any loop that has been identified as being suitable for parallelism in the static analysis stage.

Preallocate arrays. In prototype programs, it is common to allocate arrays on the fly as needed. For example, a high-pass filter might be described by the following pseudocode:

Algorithm 2 High-pass filter

```
1: Allocate arrays $temp$ and $output$ as same size as $input$.
2: Blur $input$ array into $temp$.
3: Calculate $output = input - temp$.
```

If called repeatedly, this function will repeatedly reallocate the intermediate arrays $temp$ and $output$. This is inefficient for the cache and the dynamic memory manager, because the allocated locations could continually move around. Preallocation transforms the code to allocate global buffers (or in multithreaded code, thread-local buffers) once when it is first run, which the arrays $temp$ and $output$ are then pointed to. These buffers are only reallocated on subsequent runs if the requested array size exceeds the current storage capacity. This transformation is always applied to all arrays which have been identified in the static analysis stage as suitable for preallocation.

Remove loop conditionals. In visual computing programs, it is common for there to be array lookups, which determine color or texture information. These lookups frequently include a conditional testing whether a pixel or voxel is out of bounds, in which case a default color might be used (such as black). However, such conditionals introduce major inefficiencies if performed in performance-critical inner loops [Grosser et al. 2014], due to their poor interaction with instruction pipelining and SIMD vectorization. In many cases, however, such problems can be eliminated by code rewriting. This can be done by either allocating additional *guard bands* around the array to be read [Ragan-Kelley et al. 2013], so conditionals are not required, or by breaking inner loops into several sections, so that only the *boundary region* sections need include conditionals [Grosser et al. 2014].

We use an approach based on boundary regions, because it does not require changes to memory layout. Specifically, the remove conditionals transform can be applied to any for loop. Given the loop, we find a minimal boundary region size r, such that we can prove all conditionals follow a single path if we are more than r elements from the edge of the array. Then we split the original loops into three sections so that the center section is always at least a distance r from the edge of the array. We do this by using a theorem prover [De Moura and Bjørner 2008] to find independently for each conditional i a minimal r_i such that the conditional always follows one path. We then take the maximum over all such r_i to obtain r. In the theorem prover, we use for candidate values of r small integer constants $1, \ldots, 5$, and all expressions drawn from the array indices. For example, consider the following input pseudocode which performs a blur operation:

Algorithm 3 Loop remove conditionals input program (blur)

```
1: for $y = 0, \ldots, h-1$:
2:     for $x = 0, \ldots, w-1$:
3:         $color = input(y, x)$
4:         if $x > 0$:
5:             Average $color$ with $input(y, x-1)$
6:         $output(y, x) = color$
```

After applying the loop remove conditionals transformation, we can prove that the boundary region size is $r = 1$, and the pseudocode becomes:

Algorithm 4 Loop remove conditionals result

```
 1: for $y = 0, \ldots, h-1$:
 2:     for $x = 0$:
 3:         $color = input(y, x)$
 4:         if $x > 0$:
 5:             Average $color$ with $input(y, x-1)$
 6:         $output(y, x) = color$
 7:     for $x = 1, \ldots, w-2$:
 8:         $color = input(y, x)$
 9:         Average $color$ with $input(y, x-1)$
10:         $output(y, x) = color$
11:     for $x = w-2$:
12:         $color = input(y, x)$
13:         if $x > 0$:
14:             Average $color$ with $input(y, x-1)$
15:         $output(y, x) = color$
```

Vectorize innermost. Many operations in visual computing programs involve manipulating vectors in 2, 3, or 4 dimensions. The vectorize innermost transformation allows an array operation to be converted to hardware-accelerated SIMD if the size of its last (innermost) dimension is a valid SIMD width. Typically, hardware supports SIMD widths of 2 or 4, and the length 3 case must be handled specially by the previously mentioned array storage rewriting. This transformation can be applied to any code line with an array operation.

6 Autotuner

In this section, we describe our offline autotuner, which automatically selects the fastest optimized variant of an unannotated input program. This design was inspired by the high performance achieved by autotuning systems such as FFTW [Frigo and Johnson 1998] and OpenTuner [Ansel et al. 2014].

We developed the autotuner because it can be challenging — especially for non-expert programmers — to decide which optimizations will be most beneficial. For instance, a non-expert programmer may find it challenging to determine whether a loop can safely be parallelized, and if the loop is parallelized, whether the program will actually be faster.

Our autotuner iteratively tries out a number of automatically produced *variants* of the input program, where each variant has program lines annotated with the different program transformations described in Section 5. For example, a for loop may be annotated with a *parallelize loop* transformation, or an array assignment statement might be annotated with either the *loop over implicit variables* or *vectorize innermost* transformation. Note that each transformation can only be applied to certain lines, as described in Section 5. For a given program variant, the autotuner first resolves dependencies by introducing new transformations if needed so the dependencies shown in Figure 3 are satisfied. Next, the optimized program is produced by applying the transformations, and the unit tests are run to validate that no code generation bugs occurred.

Our autotuner is initialized using a small number of program variants that constitute *good initial guesses*, and then uses hill climbing to incrementally change the best variant so as to improve the run-time. The initial variants we consider are up to 16 in number. These are constructed by considering all combinations of the following four choices: (1) Either parallelize or do not parallelize all outermost for loops that have been identified as parallelizable in the static analysis stage (Section 4.2); (2) Use type specialization for all functions; (3) Either use or do not use the *array storage alteration* transformation throughout the program; and (4) Resolve dependencies in either alphabetical or reverse-alphabetical order. The last option (4) forces a preference for either vectorizing or looping over implicit variables when resolving dependencies of the *parallelize loop* transformation shown in Figure 3.

Once the fastest variant from the initial guesses has been selected, hill climbing is used to modify the fastest variant to improve its run-time. The hill climbing randomly chooses to either: (1) Add a randomly selected new transformation (with 25% probability); (2) Mutate an existing transformation by changing its line number or any internal parameters associated with it (with 20% probability); (3) Delete an existing transformation (with 10% probability); or (4) Add a transformation by randomly sampling one from the set of good initial guesses (40% probability). The hill climbing is continued until convergence: in our case, we stopped tuning after 40 program variants had been observed.

We note that the user should construct an indicative workload for profiling that runs in a practical amount of time, so that the tuning stage is efficient. For example, a user could test a program using a modest resolution image instead of a 100 megapixel image.

7 Implementation for the Python language

In this section, we provide implementation details for our compiler, which is currently implemented to accept the language Python.

Our compiler works by reading Python into an abstract syntax tree (AST), which serves as an intermediate representation for program transformations. Each program transformation modifies this AST, by adding type or parallelism annotations, or rewriting code, until the final syntax tree is output in the Cython [Behnel et al. 2011] target language. Because the Cython language extends Python, and therefore is a superset of Python, it is also acceptable to simply apply no transformations. This will produce a valid program, however, it will not be any faster.

One limitation of our current implementation is that it is based on the mainline Python implementation, which includes a global interpreter lock (GIL). This lock essentially permits the Python interpreter to only advance execution in one thread at a time, and therefore prevents purely Python code from being faster when parallelized. Code that is parallelized should therefore be rewritten as much as possible into pure C code. This is why in Figure 3, there is a dependency between the *parallelize* and the *loop over implicit variables* and *vectorize innermost* transformations: any parallel code block aggressively calls other transformations that help rewrite code into C.

8 Test suite of applications

To evaluate our compiler, we assessed the performance of 12 applications from computer vision and graphics. Eleven of these are shown in Figure 4. The applications are as follows:

- *Bilateral grid* is an edge-preserving blur [Chen et al. 2007];
- *Camera pipeline* implements a camera raw photograph decoder [Adams et al. 2010];
- *Composite* composites a foreground image on a background;
- *Harris corner* implements a sparse interest point detector [Harris and Stephens 1988];
- *Interpolate* implements a simple Gaussian-pyramid based color interpolation [Ragan-Kelley et al. 2013];
- *Local Laplacian* is an edge-aware filter [Paris et al. 2011];
- *Mandelbrot* is a Mandelbrot fractal viewer;
- *One stage blur* implements a single stage blur based on a convolution;
- *Optical flow* computes a simplistic optical flow based only on a data term, by using PatchMatch [Barnes et al. 2009];
- *Pac-Man* produces an arcade animation;
- *Raytracer* is a simple raytracing program; and
- *Two-stage blur* implements a separable blur.

Applications were implemented by a last year undergraduate and a first year graduate student. These are students who are good general programmers, but are not particularly experienced with writing efficient graphics code.

Four applications were chosen to facilitate direct comparisons with the Halide and PolyMage DSLs: bilateral grid, interpolate, local Laplacian, and two-stage blur.

Three applications were chosen because they cannot easily be implemented in the Halide DSL. Although the dense corner detector stage of Harris corner can be implemented in Halide, the entire pipeline cannot be easily implemented. This is because the program first extracts a list of sparse interest points, but only dense arrays are supported in Halide. The program then uses scanline rendering to render the corners as circles, but scanline rendering is challenging in Halide. Pac-Man cannot be implemented easily in Halide because it uses rasterization and scan conversion, which require modifying sparse sets of pixels in loops that track imperative state. The optical flow based on PatchMatch is difficult to express in Halide because PatchMatch tracks complex imperative state inside loops, and the optical flow arrow rasterization routines also require sparse imperative looping constructs.

9 Evaluation

This section first discusses the results for the applications in our test suite. Subsequently, we then analyze the separate effect of each program transformation in Section 9.1, and assess the trade-offs made in floating-point precision in Section 9.2.

We determine running times as follows. All times were obtained on a MacBook 2.5 GHz Intel Core i7 with Haswell microarchitecture, 4 physical cores (8 hyperthreaded), 16 GB RAM, and compiled with GCC 5.1. To determine reported time, we ran an experiment where we take the minimum over a set of 10 runs of the program. Just-in-time (JIT) compilers were permitted to "warm up" and finish compiling by being run 10 times additional before any benchmarking is done. Because the reported times for the applications with our compiler were often below 1 millisecond, to improve reported accuracy we repeated the previous timing experiment 50 times and took the mean.

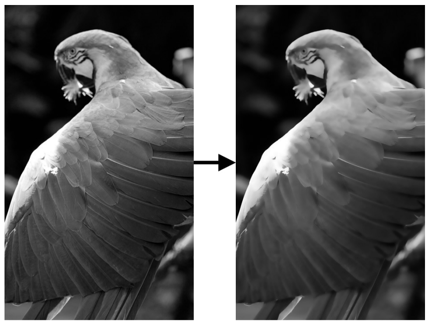
The main results are shown in Table 1. This table presents speed comparisons between our compiler and several reference compilers. For our compiler, by default, we run it in "approximating" mode, which can rewrite 64-bit floats to 32-bit floats. This introduces small numerical errors on the order of the float precision in the output. We compare this with our compiler in non-approximating mode, which disables this conversion, and find that in the median case, approximating is about 40% faster.

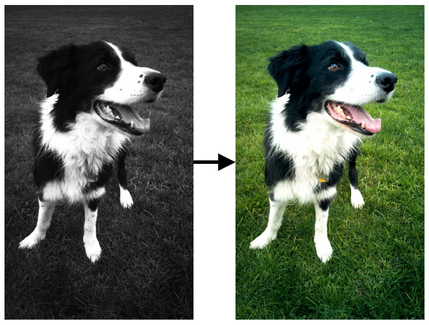
The reference compilers we compare against are the mainline Python implementation, and four Python compilers: the Numba [Lam et al. 2015] and PyPy [Bolz et al. 2009] JIT compilers, Pythran [Guelton et al. 2015], and our emulation of unPython [Garg and Amaral 2010]. For the unPython comparison, we found that the provided unPython package did not work reliably with our applications, so we simply emulated the features of unPython by using our compiler and only enabling the *parallelism* and *type specialization* transformations. We observe that our compiler is frequently orders of magnitude more efficient than other compilers. The most efficient Python compiler was Pythran, which for the median application was 7.1$\times$ slower than our compiler. However, unlike our compiler, Pythran requires the user to manually annotate the source code, and it did not successfully compile nearly half of the applications in the comparison. This was due to the approach taken by the Pythran compiler where it translates all data structures into C++ equivalents. This cannot always succeed in complex programs. Because Numba and PyPy are in widespread use, we next discuss these two in more depth.

The PyPy JIT compiler nearly always succeeded in running programs successfully, but for the median application ran 937$\times$ slower than the result of our compiler. The reason that PyPy occasionally has worse performance than Python is because its array support is not particularly strong (especially for small arrays).

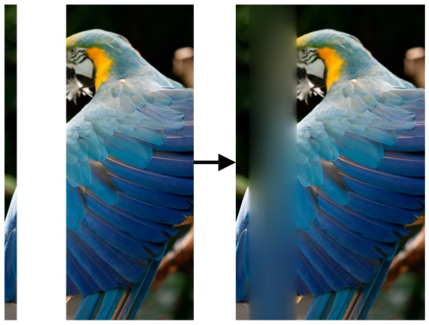
The widely-used JIT compiler Numba was also successful for most applications, but in the median case was 38$\times$ slower than the result from our compiler. This is mostly because Numba does not focus on optimization of small array operations. Our applications manipulate small arrays of constant size, for example, the RGB color vectors in the blur applications, but Numba does not perform the analysis and optimization on small arrays that we do in Sections 4 and 5. This makes Numba emit code with many performance issues, such as reallocating arrays in each loop iteration on the heap, using arbitrary size array constructors and operations, and failing to use the constant bounds to accelerate loops. Thus, Numba applications run orders of magnitudes slower than C in many applications.

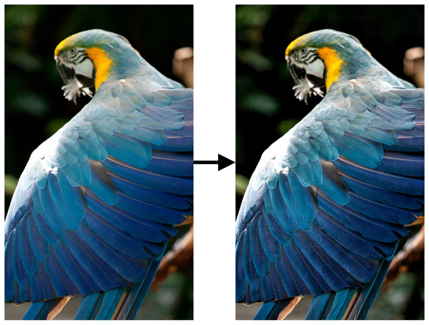
We also discovered that Numba works better when all the operations are in scalar form. For example, in one stage blur, if we manually edit the input program such that all operations work over scalar quantities instead of RGB color vectors, the Numba result becomes close to C in speed when using one thread. However, parallelizing Numba loops is currently highly nontrivial, so we used a single-threaded Numba variant when comparing with C on our 4 core test machine, and it remains about 3$\times$ slower than C. Similarly, there are a few applications that only use scalar operations in inner loops, such as grayscale composite. For these applications, Numba has better performance. However, we argue that manual conversion of programs to scalar form is not necessarily desirable. This is because scalar notation tends to make notation more complex and error-prone, especially for novice programmers. In contrast, vector

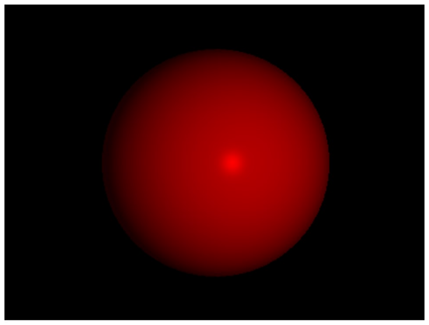

(a) Bilateral grid, 925x Speedup

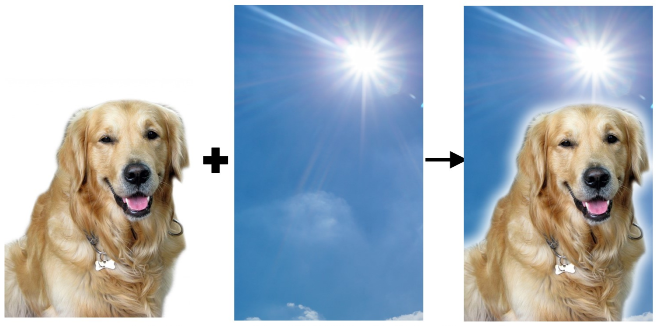

(b) Camera pipeline, 1193x Speedup

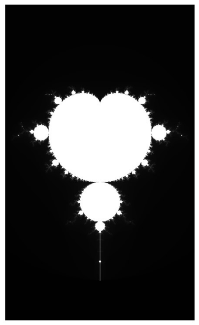
(c) Harris corner, 1223x Speedup

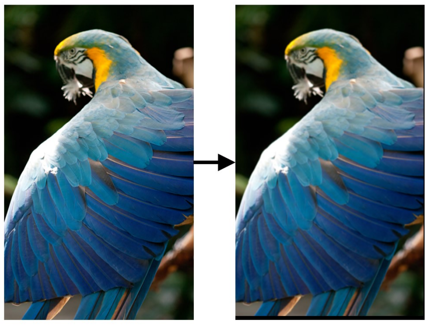

(d) Interpolate, 437x Speedup

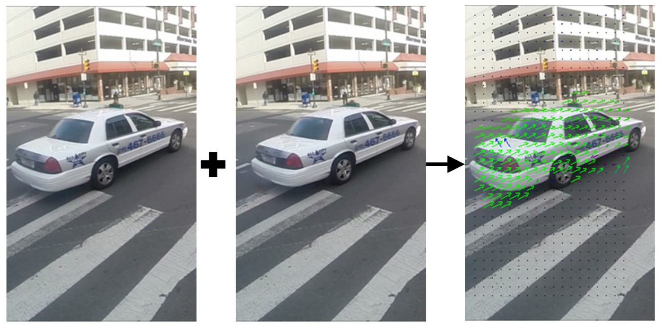

(e) Local Laplacian, 558x Speedup


(f) Raytracer, 1159x Speedup


(g) Composite (RGB), 1732x Speedup


(h) Mandelbrot, 266x Speedup


(i) Two stage blur (RGB), 1816x Speedup


(j) Optical flow, 931x Speedup

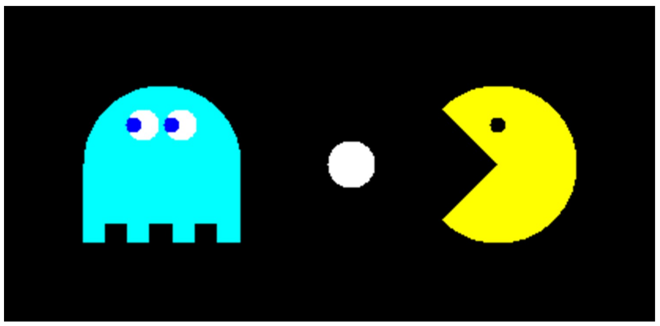

(k) Pac-Man, 273x Speedup

Figure 4: *Resulting visuals from each application, as well as the speedup of each application relative to the Python mainline implementation.*
Photo credits: (c) modified from original by Tim Green, (g) Karen Arnold and Skitterphoto.

Application	**Ours Approx. Time [ms]**	**Ours Speedup vs**							**Ours Lines**	**Ours Shorter vs**	
		Python	**Ours Non-approx.**	**Numba**	**PyPy**	**Pythran**	**unPython***	**C code**		**vs C**	**vs Cython**
Bilateral grid	94.4	925×	1.3×	1014×	3425×	Error	831×	1.0×	101	2.6×	3.0×
Camera pipeline	0.9	1193×	1.5×	1116×	2864×	Error	729×	1.1×	173	1.6×	3.0×
Composite (gray)	0.2	786×	1.2×	1.4×	40×	2.5×	1.3×	1.8×	6	2.3×	3.0×
Composite (RGB)	0.4	1732×	1.5×	72×	1604×	12×	1624×	1.9×	6	2.7×	3.0×
Harris corner	9.6	1223×	1.7×	4.1×	5322×	Error	2.0×	2.4×	92	1.2×	3.2×
Interpolate	6.7	437×	1.3×	Error	346×	Error	354×	12×	39	4.8×	4.8×
Local Laplacian	3.1	558×	1.4×	Error	793×	Error	368×	2.8×	76	4.8×	3.7×
Mandelbrot	20.0	266×	1.0×	3.0×	3.7×	10×	1.0×	3.1×	29	1.6×	2.8×
One stage blur (gray)	0.4	2281×	1.5×	3.3×	255×	4.1×	1.5×	2.1×	31	1.8×	9.5×
One stage blur (RGB)	0.7	4650×	2.1×	414×	18319×	77×	4306×	2.3×	31	2.5×	2.3×
Optical flow	8.7	931×	1.2×	Error	Error	Error	12×	0.9×	232	1.6×	3.1×
Pac-Man	0.1	273×	1.4×	1.1×	4.9×	1.9×	3.1×	11×	111	1.2×	2.0×
Raytracer	1.6	1159×	1.1×	1189×	1081×	Error	1130×	0.8×	49	3.6×	3.1×
Two stage blur (gray)	0.5	713×	1.7×	2.6×	170×	4.4×	2.1×	2.5×	10	2.1×	4.3×
Two stage blur (RGB)	0.8	1816×	2.3×	81×	1582×	24×	1684×	6.0×	10	2.5×	4.3×
Median		1816×	1.4×	38×	937×	7.1×	354×	2.3×		2.3×	3.1×

Table 1: *Comparison of the speedups and lines of code for our compiler versus alternatives. We evaluate all applications with our compiler in approximating mode (64-bit floats can be recast to 32-bit floats), and compare with our compiler in non-approximating mode (64-bit floats stay 64-bit), the mainline Python implementation and three compilers designed to accelerate Python code: Numba, PyPy, Pythran, and our emulation of unPython (*We use our compiler infrastructure to mimic the unPython compiler). We also include a comparison against handwritten, manually ported C code that uses 64-bit precision and has been compiled with a vectorizing C compiler (GCC 5.1.0). We frequently find significant speedups against these alternatives, despite having less lines of code than handwritten C and the generated Cython code output by our compiler. The number of lines does not include imports, includes, comments, blank lines, or test lines.*

or matrix notation is frequently more concise, general, and because of its greater abstraction, facilitates additional optimizations such as vectorization.

Three of our applications produce efficient code when run on both grayscale and RGB images *without any change to the input Python program.* This happens because type specialization identifies different constant array sizes for the third dimension. Thus the colorspace, either RGB or grayscale, is noted in the performance evaluations. We believe this automatic specialization shows the benefit of our focus on array- and image-related optimizations.

We also compare against naive handwritten C in Table 1. This C code was written by the same students who wrote the Python applications, and had loop parallelism directives manually inserted, and compiled with maximum optimizations. This C code was *intentionally written to be in a simplistic form* that does not rely on much knowledge of hardware or optimization, to mimic the style of the Python code, as well as the style that an inexperienced programmer might write code in. For example, images are not explicitly padded to use 4 channels to facilitate SIMD vectorization. In some sense this comparison is unfair towards our compiler, because it depends on a human, who must manually port the program between languages and then parallelize it, whereas our compiler is fully automatic. However, the results for our compiler are still competitive.

Application	Ours time	Ours vs Halide	Ours vs PolyMage
Bilateral grid	94 ms	8× slower	4× slower
Interpolate	107 ms	6× slower	2× slower
Local Laplacian	759 ms	8× slower	7× slower
Two stage blur	35 ms	2× slower	N/A

Table 2: *Speed comparison with Halide [Ragan-Kelley et al. 2013] and PolyMage [Mullapudi et al. 2015]. All comparisons are made using multi-threaded builds.*

In the median case the C programs are 2.3× slower than the result of our compiler, but in the worst case, C is up to 12× slower than our result. The C programs are also in the median case 2.3× longer in lines of code than the Python programs.

Finally, in Table 2, we compared our application run-times against the Halide and PolyMage domain-specific languages (DSLs). Again, in a sense this is an unfair comparison because it requires humans to port programs from a general-purpose Turing complete language (Python) to special-purpose DSLs that are not Turing complete and can only express certain image pipelines over dense arrays. For these applications, which can fit in the compute model of these DSLs, we find that our compiler generates programs that are 2 to 8× slower than Halide and 2 to 7× slower than PolyMage. Note that other applications such as Harris corner, Pac-Man, and optical flow are difficult to express entirely within the Halide DSL. The difference in running time is primarily due to the loop fusion rules in Halide and PolyMage, which are difficult to implement as code transformations in general-purpose imperative programming languages. Note that the compilers community has investigated loop fusion rules [Gao et al. 1993; Kennedy and McKinley 1994], but these are inadequate to gain the full benefits of the fusion and parallelization strategies used by Halide and PolyMage. We leave the integration of these complex strategies into general-purpose, imperative languages for future research. We emphasize that *the benefit of our compiler is in being able to handle arbitrary programs in an imperative, Turing-complete language of Python, whereas DSLs are both less widely adopted and have a more limited range of programs they can express.*

In Figure 5, we show the total amount of profiling and compilation time used when building the code for each application. Neither the C compiler stage nor our code generator have been particularly optimized, so we expect these times could be improved.

To aid reproducibility and spur future work, we are releasing our compiler, applications, and output C code as an open source project.

Application	API call rewriting*	Array storage alteration	Loop over implicit variables*	Parallelize loop	Preallocate arrays	Remove loop conditionals	Type specialization*	Vectorize vs native C*†	Vectorize vs interpreted*†
Bilateral grid	147×	0.9×	4.0×	2.2×	1.0×		0.7×		
Camera pipeline	274×	1.1×	1.0×	2.1×	1.0×		1.1×		
Composite (gray)		0.9×		2.2×		1.0×	321×		
Composite (RGB)		0.8×		1.7×		1.0×	1.0×	2.3×	1334×
Harris corner	1.0×			3.3×	1.2×		247×		
Interpolate	0.9×	0.9×		1.3×	1.0×		0.9×	1.3×	422×
Local Laplacian	158×	1.2×	1.3×	1.5×	1.0×		4.5×		
Mandelbrot	1.0×	1.0×		2.9×	1.0×		58×		
One stage blur (gray)				2.8×		1.4×	328×		
One stage blur (RGB)		0.8×		2.8×			0.7×	4.3×	2736×
Optical flow	11×		1.0×	1.3×	1.0×		47×		
Pac-Man	2.3×	1.7×			1.0×		58×		
Raytracer	138×		2.3×	4.1×	1.0×		0.8×		
Two stage blur (gray)		2.0×		1.6×	1.1×		128×		
Two stage blur (RGB)		0.9×		1.3×	1.0×		0.9×	5.3×	1335×
Mean	81×	1.1×	1.9×	2.2×	1.0×	1.1×	80×	3.3×	1457×
Max	274×	2.0×	4.0×	4.1×	1.2×	1.4×	328×	5.3×	2736×

Table 3: *Transformations used by each application. Numbers in cells indicate the speedups for a particular transformation and application, relative to not using the given transformation. Blank cells indicate a transformation was not chosen by the autotuner. These speedups are determined by starting from the final program and then removing a single transformation at a time. Every transformation is independently useful, and transformations frequently give compound benefits. *These transformations can rewrite interpreted Python code into native C. †Vectorize is compared relative to both non-vectorized yet fully compiled native C, as well as interpreted Python code. For details, see Section 9.1.*

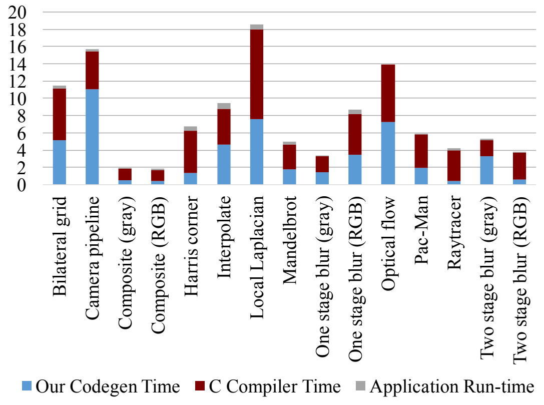


Figure 5: *The total time in minutes spent in the profiling and compilation stages (for both our code generator and the C compiler) for each application.*

9.1 Assessing the effect of each transformation

This section explores the speed-ups that are obtained from each particular transformation and each application, as well as aggregate statistics across applications. The resulting speed-ups for each transformation are shown in Table 3. We see that most transformations give significant speed-ups. More importantly, we see that transformations are synergistic so that *several transformations can contribute to the high performance of a given program*, and *every transformation contributes* to the high performance of some application.

The speedups are determined by starting with each final tuned program and removing transformations one at a time in an order such that no dependencies in Figure 3 are broken. If any two transformations are "tied" such that either could be removed without breaking dependencies then the transformation with a name that alphabetically comes first is removed first. Each time a transformation is removed, the program typically slows down, and thus, a speed-up for re-inserting the given transformation can be calculated as the inverse of this slow-down. We assess the effect of vectorization as a special case, because vectorization requires code that has been compiled as native C, i.e. without calls to the interpreter. Therefore, in Table 3, we compare each vectorized program against a fully compiled but not vectorized variant, that instead loops over implicit variables (indicated by "vectorize vs native C"), as well as against the original interpreted Python (indicated by "vectorize vs interpreted"). For this experiment, the compiler was run in "approximating" mode, where 64-bit floats can be rewritten to 32-bit floats.

Table 3 also indicates which transformations are used for each application by the presence of absence of numbers in the cells. Note the variety of different transformations that are applied to different visual computing programs. The richness of this transformation space permits our compiler to achieve high performance.

Frequently, significant performance gains are due to rewriting code that calls the Python interpreter to native C code. Four of our transformations help transform interpreted code to native C: *type specialization*, *API call rewriting*, *loop over implicit variables*, and *vectorize innermost*. These are indicated with asterisks in Table 3. These transformations are greatly facilitated by the tracking of array shapes throughout the program by our dependent type analysis (Section 4.1). Please see the supplemental document for details on when exactly the rewriting to native C occurs.

9.2 Assessing trade-offs in floating-point precision

This section explores the speed gains encountered during the 64-bit to 32-bit floating-point precision change, which is a feature enabled in our compiler's "approximation mode." Converting to 32-bit can reduce memory and speed up run-time, but the precision loss introduced may potentially result in small errors on the order of the machine epsilon. Manually specifying 32-bit in Python requires carefully including the primitive data type everywhere that arrays are used, and the performance impacts of this may not be apparent to novice programmers. On the other hand, some applications may gain no speed improvement from using lower precision 32-bit floats, so one may as well use higher precision 64-bit floats. We thus added the approximation mode to make exploration of this optimization trade-off as easy as possible.

In Figure 6, we show the speedup of the 32-bit variant of each application relative to the 64-bit variant. We also show the mean speedup over all applications. In the scenario where all programs are limited by memory bandwidth, 32-bit variants require half as much bandwidth and should be twice as fast as 64-bit variants. However, in Figure 6 we observe that this is not always the case. This can be because in compute-bound situations, the speedup due to choosing 32-bit depends on vectorization, because in vector units, two 32-bit instructions can be performed in the same time as one 64-bit instruction. We note that applications that are compute-bound and vectorize poorly, such as Mandelbrot and raytracer, have speedups closer to one.

To examine these relatively small speedups, we looked further into the compute-limited application Mandelbrot. We investigated the

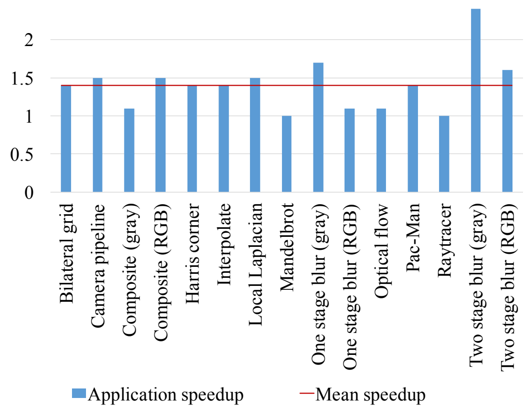


Figure 6: *The speedup of the 32-bit floating-point variant of each application relative to the 64-bit variant.*

instructions from both 32-bit and 64-bit bit programs both in C and with our compiler. We found that in the innermost loop there are no vectorization instructions. Thus, we believe that without manual intervention, our C compiler (GCC 5.1) is not able to vectorize programs with complicated logic in their innermost loops. We conclude that compute-limited programs will only see a larger speedup in 32-bit mode relative to 64-bit if the compiler's vectorizer is sufficiently capable. We note that better vectorization is possible in C-like compilers such as `ispc` [Pharr and Mark 2012] that use the *single program, multiple data* programming model.

10 Discussion

Our compiler approach has a few important limitations. First of all, the dependent type analysis expects that any built-in or third-party functions implemented in foreign languages (such as C) are annotated with "dependent type functions," as described in Section 4.1. If this is not done then types in the program might have to be manually annotated by the programmer, which is inconvenient. In future work, it would be interesting to explore automatically generating these dependent type functions.

Our compiler currently does not target graphics processing units (GPUs). A compiler that targets GPUs might be able to incorporate interesting optimizations such as placing parallel workloads on both the CPU and the GPU, using half-precision floating point arithmetic, or vectorizing using the large GPU vector widths.

The observations we used for visual computing programs might also be applied to similar domains that involve accelerating vector and matrix operations in 2, 3, and 4 dimensions, such as physical and acoustical simulation, and scientific visualization.

In conclusion, our compiler framework allows speedups of orders of magnitude to be obtained over state-of-the-art compilers for Python. It does this by specializing for the domain of visual computing, while keeping the input language to be a general-purpose, Turing-complete, dynamically-typed language. Unlike DSLs, we can express a more rich variety of visual computing programs. The resulting programs are both shorter and faster than their C equivalents, and our programmers ran into fewer complicated memory management issues than when they were writing the C comparisons. We believe that the ideas contained within our compiler framework will be able to impact compilers for other dynamic languages and open up high-performance visual computing to broader audiences.

Acknowledgements

We thank our anonymous reviewers for helpful feedback. Thanks to the photographers for licensing photos under Creative Commons or public domain, as indicated in figure captions. This project was partially funded by the NSF grants HCC 1011444 and SHF 1619123.

References

ADAMS, A., TALVALA, E.-V., PARK, S. H., JACOBS, D. E., AJDIN, B., GELFAND, N., DOLSON, J., VAQUERO, D., BAEK, J., TICO, M., ET AL. 2010. The frankencamera: an experimental platform for computational photography. In *ACM Transactions on Graphics (TOG)*, vol. 29, ACM, 29.

AKERET, J., GAMPER, L., AMARA, A., AND REFREGIER, A. 2015. Hope: A python just-in-time compiler for astrophysical computations. *Astronomy and Computing 10*, 1–8.

AMMONS, G., AND LARUS, J. R. 1998. Improving data-flow analysis with path profiles. In *ACM PLDI*.

ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, ACM.

AUGUSTSSON, L. 1998. Cayennea language with dependent types. In *ACM SIGPLAN Notices*, vol. 34, ACM, 239–250.

BALL, T., MATAGA, P., AND SAGIV, S. 1998. Edge profiling versus path profiling: The showdown. In *POPL '98*, 134–148.

BARNES, C., SHECHTMAN, E., FINKELSTEIN, A., AND GOLDMAN, D. 2009. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics 28*, 3, 24:1–24:10.

BEHNEL, S., BRADSHAW, R., CITRO, C., DALCIN, L., SELJEBOTN, D. S., AND SMITH, K. 2011. Cython: The best of both worlds. *Computing in Science & Engineering 13*, 2, 31–39.

BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. 2010. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf.*

BEZANSON, J., KARPINSKI, S., SHAH, V. B., AND EDELMAN, A. 2012. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*.

BLUME, W., DOALLO, R., RAUCHWERGER, L., TU, P., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., ET AL. 1996. Parallel programming with polaris. *Computer*, 12, 78–82.

BOLZ, C. F., CUNI, A., FIJALKOWSKI, M., AND RIGO, A. 2009. Tracing the meta-level: Pypy's tracing jit compiler. In *Workshop on Implementation, Compilation, Optimization*.

BUSE, R. P. L., AND WEIMER, W. 2009. The road not taken: Estimating path execution frequency statically. In *31st International Conference on Software Engineering, ICSE 2009*.

CANN, D. C., AND EVRIPIDOU, P. 1995. Advanced array optimizations for high performance functional languages. *Parallel and Distributed Systems, IEEE Transactions on 6*, 3, 229–239.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. In *ACM Transactions on Graphics (TOG)*, vol. 26, ACM, 103.

CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. C. 2007. Dependent types for low-level programming. In *Programming Languages and Systems*. Springer, 520–535.

CORNWALL, J. L., HOWES, L., KELLY, P. H., PARSONAGE, P., AND NICOLETTI, B. 2009. High-performance simt code generation in an active visual effects library. In *Proceedings of the 6th ACM conference on Computing frontiers*, ACM, 175–184.

DE MOURA, L., AND BJØRNER, N. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

DEVITO, Z., HEGARTY, J., AIKEN, A., HANRAHAN, P., AND VITEK, J. 2013. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, vol. 48.

DUFOUR, M., AND COUGHLAN, J., 2013. Shedskin: an experimental (restricted-python)-to-c++ compiler. `https://code.google.com/p/shedskin/wiki/docs`.

ELLIOTT, C. Functional image synthesis. In *Proceedings Bridges 2001, Mathematical Connections in Art, Music, and Science*.

FRIGO, M., AND JOHNSON, S. G. 1998. Fftw: An adaptive software architecture for the fft. In *1998 IEEE Acoustics, Speech and Signal Processing*, vol. 3.

FRIGO, M., AND STRUMPEN, V. 2005. Cache oblivious stencil computations. In *Proc. 19th conf. Supercomputing*, ACM.

GAO, G., OLSEN, R., SARKAR, V., AND THEKKATH, R. 1993. Collective loop fusion for array contraction. In *Languages and Compilers for Parallel Computing*. Springer, 281–295.

GARG, R., AND AMARAL, J. N. 2010. Compiling python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on GPGPU*, ACM.

GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. 1982. gprof: a call graph execution profiler (with retrospective). In *ACM SIGPLAN PLDI*.

GROSSER, T., COHEN, A., HOLEWINSKI, J., SADAYAPPAN, P., AND VERDOOLAEGE, S. 2014. Hybrid hexagonal/classical tiling for gpus. In *Proc. IEEE/ACM Symposium on Code Generation and Optimization*.

GUELTON, S., BRUNET, P., AMINI, M., MERLINI, A., CORBILLON, X., AND RAYNAUD, A. 2015. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery 8*, 1, 014001.

HARMAN, M., AND JONES, B. F. 2001. Search-based software engineering. *Information and Software Technology 43*, 14.

HARRIS, C., AND STEPHENS, M. 1988. A combined corner and edge detector. In *Alvey vision conference*, vol. 15, Citeseer, 50.

HOLEWINSKI, J., POUCHET, L.-N., AND SADAYAPPAN, P. 2012. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, ACM, 311–320.

KARCHER, T., AND PANKRATIUS, V. 2011. Run-time automatic performance tuning for multicore applications. In *Euro-Par 2011 Parallel Processing*. Springer, 3–14.

KENNEDY, K., AND MCKINLEY, K. S. 1994. *Maximizing loop parallelism and improving data locality via loop fusion and distribution*. Springer.

KRISHNAMOORTHY, S., BASKARAN, M., BONDHUGULA, U., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2007. Effective automatic parallelization of stencil computations. In *ACM Sigplan Notices*, vol. 42, ACM, 235–244.

LAM, S. K., PITROU, A., AND SEIBERT, S. 2015. Numba: a llvm-based python jit compiler. In *Proceedings of Workshop on LLVM Compiler Infrastructure in HPC*, ACM.

MORAJKO, A., MARGALEF, T., AND LUQUE, E. 2007. Design and implementation of a dynamic tuning environment. *Journal of Parallel and Distributed Computing 67*, 4, 474–490.

MULLAPUDI, R. T., VASISTA, V., AND BONDHUGULA, U. 2015. Polymage: Automatic optimization for image processing pipelines. In *Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, ACM.

MULLAPUDI, R. T., ADAMS, A., SHARLET, D., RAGAN-KELLEY, J., AND FATAHALIAN, K. 2016. Automatically scheduling halide image processing pipelines. In *Proc. ACM SIGGRAPH*.

PARIS, S., HASINOFF, S. W., AND KAUTZ, J. 2011. Local laplacian filters: edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph. 30*, 4, 68.

PHARR, M., AND MARK, W. R. 2012. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, IEEE, 1–13.

Quora: How many photos are uploaded to facebook every day? `https://www.quora.com/How-many-photos-are-uploaded-to-Facebook-each-day`.

RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. 2013. Halide: A language and compiler for optimizing parallelism, locality and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SALIB, M. 2004. *Starkiller: A static type inferencer and compiler for Python*. PhD thesis, Citeseer.

SHANTZIS, M. A. 1994. A model for efficient and flexible image computing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, ACM, 147–154.

SYED-ABDUL, S., FERNANDEZ-LUQUE, L., JIAN, W.-S., LI, Y.-C., CRAIN, S., HSU, M.-H., WANG, Y.-C., KHANDREGZEN, D., CHULUUNBAATAR, E., AND NGUYEN, P. A. 2013. Misleading health-related information promoted through video-based social media: anorexia on youtube. *Journal of medical Internet research 15*, 2, e30.

TALBOT, J., DEVITO, Z., AND HANRAHAN, P. 2012. Riposte: a trace-driven compiler and parallel vm for vector code in r. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 43–52.

TANG, Y., CHOWDHURY, R. A., KUSZMAUL, B. C., LUK, C.-K., AND LEISERSON, C. E. 2011. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 117–128.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimizations of software and the atlas project. *Parallel Computing 27*, 1, 3–35.

XI, H., AND PFENNING, F. 1998. Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices 33*, 5, 249–257.

XI, H., AND PFENNING, F. 1999. Dependent types in practical programming. In *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM.