A Dynamic to Static DSL Compiler for Image Processing Applications

Pierre Guillou, Benoît Pin, Fabien Coelho, François Irigoin
MINES ParisTech, PSL Research University, France
firstname.lastname@mines-paristech.fr

***Abstract*—Computer vision is a thriving field of research, and Python is an instrument of choice for developing image processing software applications. Used in conjunction with specialized libraries written in C or C++, performance can be enhanced to match native code. The SMIL library [1] is a new C++ image processing library offering ease of programming with a Python wrapper. However, SMIL applications also have to be executed on embedded platforms such as FPGAs on which a Python interpreter is not available. The generic answer to such an issue is to re-code the original Python applications in C or C++, which will be then optimized for every hardware target, or to try to compile Python into native code using tools such as Cython [2]. The approach taken by the FREIA project [3, 4] is to ease portability of applications written in a DSL embedded in C (the FREIA API) by using specific optimizations such as image expressions evaluation, removal of temporary variables or image tiling. Is it possible for SMIL Python applications to benefit from the FREIA compilation toolchain in order to increase their portability onto specialized hardware targets? We present in this paper (1) a methodology to convert a dynamic DSL into a static one that preserves programmability, (2) a working implementation which takes care of types, memory allocation, polymorphism and API adaptation between SMIL and FREIA, (3) and experimental results on portability and performance.**

I. Introduction

Computer vision is now a fast-growing field of research and is going to play a large role in everybody's life in the near future. Augmented reality or autonomous vehicles such as drones or cars are showing more and more promise and should be available to consumers in the next decade. To support this innovative field, new image processing libraries are developed and compete in terms of performance and programmability. They are often based on a Python API: the Python programming language offers good programmability through its high-level abstractions, dynamic type system and easy-to-learn syntax. Native performance can also be achieved using wrappers around C or C++ code.

However, in our hardware jungle era [5], substantial efforts have to be made to have those libraries efficiently running onto all kinds of modern accelerators, such as GPUs, FPGAs or many-core processors. Production compilers are not yet able to target the whole range of today's hardware: it is up to developers to provide an optimized version of their library onto a specific hardware. Programming models such as OpenMP [6] for shared memory, OpenCL [7] for heterogeneous platforms or MPI [8] for distributed computing can help target a class of accelerators at once, but further optimizations are often hardware-specific.

New compiler techniques must arise to support complex image processing applications without sacrificing programmability. This paper focuses on two image processing interfaces considered as DSLs, SMIL and FREIA, supporting each a different set of hardware targets and providing different levels of programmability. We built a compiler to automatically generate lower-level but more portable FREIA DSL code from high-level SMIL DSL applications. We evaluate this compiler on a set of seven image processing applications.

II. Context

Mathematical Morphology is an image processing theory based on lattice theory initiated at MINES ParisTech. Several software libraries have been developed since the inception of this theory, each providing better performance or usability. SMIL and FREIA are two of them.

A. The SMIL library

SMIL (*Simple Morphological Image Library*) [1, 9] is a new C++ image processing library developed at MINES ParisTech. It focuses on efficiently implementing mathematical morphology operators such as erosions and dilatations onto modern multicore CPUs. It aims at providing:

- good performance, using loop auto-vectorization through the GCC compiler [10] and OpenMP parallelization [6];
- ease of programming, through several Swig [11] auto-generated interfaces to higher-level programming languages such as Python;
- portability on several CPUs and operating systems, using the CMake [12] compilation toolchain;
- maintainability and extensibility, through C++ templates and functors.

Figure 1 depicts an example of a SMIL script using the Python interface. This script reads an image from a file, performs an morphological dilatation on this input image and then saves the resulting image.

```python
import smilPython as smil

imin = smil.Image("input.png")
imout = smil.Image(imin)
smil.dilate(imin, imout)
imout.save("output.png")
```

Figure 1. Morphological dilatation in SMIL

B. The FREIA framework

FREIA (*Framework for Embedded Image Applications*) [3] is a C image processing framework. It provides a C API divided into elementary and composed image operators. This API abstracts several implementations targeting different categories of hardware accelerators :

- multicore and vector CPUs with SMIL (through an intermediate C wrapper around SMIL C++ code);
- CPUs with vector extensions through Fulguro [13];
- FPGAs with the SPoC [14] and Terapix [15] backends;
- manycore CPUs such as the Kalray MPPA [16] with Sigma-C [17, 18], a dataflow programming language;
- GPUs using OpenCL [7].

Used in combination with our in-house C source-to-source compiler framework PIPS [19], FREIA applications can be further optimized at the image operator level for the designated hardware target [4, 20].

Figure 2 represents an abridged version of a morphological dilatation using the FREIA API. In this example, image structures must be explicitly allocated before use and freed after.

```c
#include "freia.h"
int main(void) {
  /* initializations... */
  freia_data2d *imin = freia_common_create_data(/*...*/);
  freia_data2d *imout = freia_common_create_data(/*...*/);
  freia_common_rx_image(imin, /*.. */);
  freia_cipo_dilate(imout, imin, 8, 1);
  freia_common_tx_image(imout, /*.. */);
  freia_common_destruct_data(imin);
  freia_common_destruct_data(imout);
  /* shutdown... */
}
```

Figure 2. Morphological dilatation in FREIA (excerpt)

C. Bridging the gap

The SMIL library supports several CPU targets using the GCC compiler and the CMake toolchain. Nevertheless, porting this image processing library on specific hardware accelerators such as GPUs to take advantage of heterogeneous architectures can be a hard task. Meanwhile, the FREIA framework supports a wide range of hardware accelerators, but fails in comparison to offer easy programmability: users still have to manage memory and write C code.

Several solutions are possible to reconciliate SMIL programmability and FREIA set of hardware targets. A total port of SMIL on every FREIA target is a hard and long task, and would not reuse the work done for FREIA. One can try to re-implement the SMIL API using FREIA, but this solution does not allow our C compiler PIPS to perform its optimizations. Allowing C- and Fortran-supporting PIPS to handle C++ source code to regenerate directly target-optimized FREIA code is also a lengthy process, and although it can yield some long-term benefits, it is not in the scope of this project.

In this paper, we present `smiltofreia`, a source-to-source compiler designed to convert SMIL applications written using the Python interface into FREIA C. Thus a SMIL application can be automatically ported to the FREIA-supported hardware targets and take advantage of the PIPS source-to-source compiler optimizations. Figure 3 represents our compilation chain highlighting the benefits of `smiltofreia` in terms of portability.

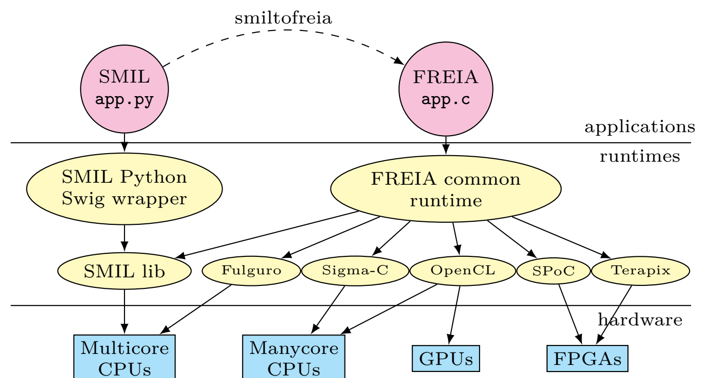

Figure 3. Compiler toolchain diagram

III. Manipulating and accelerating Python code: RedBaron and Cython

In order to analyze and convert SMIL Python applications into FREIA C, we tried two Python tools: RedBaron [21, 22], a Python refactoring framework, and Cython [2], a Python-to-C compiler.

The Python standard library itself provides low-level tools to parse and query Python code. Among them, *inspect* [23], which can be used to inspect and modify running Python code, and *ast* [24], for manipulating Python code Abstract Syntax Trees.

A. RedBaron, a Python refactoring tool

We built our `smiltofreia` compiler on top of the RedBaron refactoring tool. RedBaron is a high-level Python interface allowing developers to easily refactor their Python code without losing information. It is based on the Baron [25] FST (*Full Syntax Tree*) which, unlike a traditional AST, does not drop comments and formatting data. As a consequence, regenerating source code from a FST should be an invariant transformation:

```python
fst_to_code(code_to_fst(source_code)) == source_code
```

For instance, the RedBaron FST of the SMIL dilatation call `smil.dilate(imin, imout)` from Figure 1 is represented in Figure 4. Note that the formatting information (line

breaks and spaces) separating the words is kept in this data structure in order to regenerate the very same source code.


```
{"type": "atomtrailers",
 "value": [
     {"type": "name", "value": "smil"},
     {"type": "dot", "first_formatting": [],
      "second_formatting": []},
     {"type": "name", "value": "dilate"},
     {"first_formatting": [], "third_formatting": [],
      "type": "call", "fourth_formatting": [],
      "second_formatting": [],
      "value": [
          {"type": "call_argument",
           "first_formatting": [],
           "second_formatting": [], "target": {},
           "value": {"type": "name", "value": "imin"}},
          {"type": "comma", "first_formatting": [],
           "second_formatting": [{"type": "space",
                                  "value": " "}]},
          {"type": "call_argument", "first_formatting": [],
           "second_formatting": [], "target": {},
           "value": {"type": "name", "value": "imout"}}
      ]}]}
```


Figure 4. FST of `smil.dilate(imin, imout)`

RedBaron provides an efficient and intuitive object-oriented interface to query and manipulate this FST. Top-level nodes, corresponding to actual lines of codes, can be accessed and modified through array subscripts and assignments. Transforming the dilatation in the FST `fst` of Figure 1, line 4, into an image copy is as simple as rewriting the content of the corresponding node `fst[3] = "imout = imin"`.

Compared to RedBaron, the Python standard module *ast* provides a clumsier interface and drops some essential formatting information, useful when refactoring.

B. *Cython, a Python-to-C compiler*

We investigated the use of Cython, a Python-to-C compiler, for generating FREIA code from our SMIL Python applications. First, we wrapped a subset of the FREIA API in Python using the Cython extension system, which provides an easy way to interface Python applications and C libraries. Then we used RedBaron to convert SMIL applications into FREIA Python, and from this point generate standalone C code using the Cython compiler. Figure 5 represents our Cython toolchain to generate FREIA from SMIL.

Our SMIL dilatation script in Figure 1 is thus transformed into the FREIA/Cython Python script of Figure 6. The Cython compiler then generates a C source file from this Python code. Figure 7 shows the output of the Cython Python-to-C compiler around the FREIA dilatation call. However, the generated source code is too low-level, and thus too far from FREIA, for our source-to-source framework PIPS to perform additional relevant optimizations.

Cython introduces a lot of new variables and uses opaque data structures, which makes the code a lot more complex to analyze. As a consequence, PIPS regeneration of optimized source code for the specific hardware targets could not work.

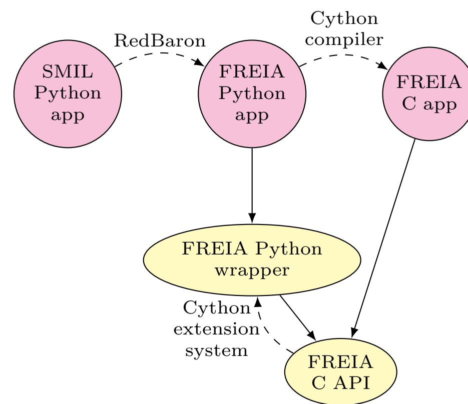

Figure 5. Using Cython to convert SMIL applications into FREIA

```python
def main(*args):
    freia.initialize(*args)
    fdin = freia.DataIO()
    fdin.openInput(0)
    fdout = freia.DataIO()
    fdout.openOutput(0, fdin.framewidth, fdin.frameheight,
                     fdin.framebpp)
    freia_img_imin = freia.Data2D(fdin.framebpp,
                                  fdin.framewidth,
                                  fdin.frameheight)
    freia_img_imin.rxImage(fdin)
    freia_img_imout = freia.Data2D(fdin.framebpp,
                                   fdin.framewidth,
                                   fdin.frameheight)
    freia_img_imin.cipoDilate(freia_img_imout, 8, 1)
    freia_img_imout.txImage(fdout)
    freia.shutdown()
```

Figure 6. Conversion of a SMIL dilation in FREIA/Cython

The Cython approach, which works well for interfacing Python and C code and hence accelerating Python applications, is thus not recommended for post-processing the generated C code. Even though this approach was not pursued, experiments with Cython nonetheless played a role in PIPS development by providing hard to analyze generated C code.

```c
static PyObject *__pyx_pf_9smil_dilate_6Data2D_14cipoDilate(
    struct __pyx_obj_9smil_test_Data2D *__pyx_v_self,
    struct __pyx_obj_9smil_test_Data2D *__pyx_v_imout,
    __pyx_t_7pyfreia_int32_t __pyx_v_connexity,
    __pyx_t_7pyfreia_uint32_t __pyx_v_size) {
  PyObject *__pyx_r = NULL;
  __Pyx_RefNannyDeclarations PyObject *__pyx_t_1 = NULL;
  __Pyx_RefNannySetupContext("cipoDilate", 0);
  __Pyx_XDECREF(__pyx_r);
  __pyx_t_1 = PyInt_FromLong(
      freia_cipo_dilate(__pyx_v_imout->_c_data2d,
                        __pyx_v_self->_c_data2d,
                        __pyx_v_connexity, __pyx_v_size));
  __Pyx_GOTREF(__pyx_t_1);
  __pyx_r = __pyx_t_1;
  __pyx_t_1 = 0;
  __Pyx_XGIVEREF(__pyx_r);
  __Pyx_RefNannyFinishContext();
  return __pyx_r;
}
```

Figure 7. Actual C call to FREIA dilatation after Cython compilation

IV. smiltofreia, a SMIL Python to FREIA C compiler

Instead of using Cython for generating low-level C from Python, we developed an in-house Python-to-C compiler for SMIL applications. Our compiler, named `smiltofreia`, generates directly FREIA C code from SMIL Python applications. `smiltofreia` iterates over the RedBaron FST of a SMIL application and transforms each node into a corresponding C statement. An example of this compiler output is available on Figure 8.

```
#include "freia.h"
#include "smil-freia.h"

int main(int argc, char *argv[]) {
  /* initializations... */
  freia_data2d *imin;
  imin = freia_common_create_data(/* */);
  freia_data2d *imout;
  imout = freia_common_create_data(/* */);
#define e0 SMILTOFREIA_SQUSE
#define e0_s 1
  freia_cipo_dilate_generic_8c(imout, imin, e0, e0_s);
  freia_common_tx_image(imout, &fdout);
  freia_common_destruct_data(imout);
  freia_common_destruct_data(imin);
  /* shutdown... */
}
```

Figure 8. Simplified FREIA C output of our compiler for Figure 1

Python is a dynamic language with a garbage collector dealing with memory allocation. A contrario, C is lower-level: variables must be declared; memory management is done by hand; and heap-allocated memory must (ideally) be freed at the end of its use. Besides, SMIL and FREIA API, although close, can differ. Our compiler addresses these differences to generate code that respects the C specification and the semantics of the source SMIL application. As a consequence, our compiler input is constrained: only pure SMIL Python code without other Python modules is supported, and the type of all variables must be statically inferable.

A. Typing

Our compiler is focused on a subset of the SMIL API that has an equivalent in FREIA, and must also deal with issues arising when trying to generate static code from a dynamic one. We wrote a defensive implementation that puts programming constraints on the Python input code. The goal is to ensure that a successful transformation will produce a well-typed and well-memory-managed C code. The `smiltofreia` compiler knows both SMIL and FREIA APIs and the correspondence between their functions' signatures. Variables are typed at first initialization and cannot be mutated. The compiler fails otherwise with a consistent error message, thanks to RedBaron FST, which provides a convenient way to locate a specific node in Python code. Function arguments are also typed and transformed before they are passed to FREIA functions.

B. Function Polymorphism

The SMIL library features polymorphism i.e., methods can have several signatures, which requires some care when transforming. We also chose to keep real-world SMIL Python as a developer would write it as an input. However, FREIA is more rigid and needs fully-typed arguments when calling functions. For example, we use several tricks to deal with optional parameters such as rewriting Python code on the fly to a canonical form closer to the corresponding FREIA call. For this purpose, the RedBaron ability to access and modify FST nodes is key.

The following Python code illustrates the polymorphism of the `smil.dilate` function regarding its last argument:

```
1 smil.dilate(imin, imout, 5)
2 smil.dilate(imin, imout, smil.SquSE(5))
```

- At Line 1, the last parameter is an integer; in this case it denotes a 5-pixel wide square structuring element.
- At Line 2, the last parameter is a full-fledged structuring element.

The first line is internally modified, using RedBaron abilities to rewrite nodes, to yield

```
smil.dilate(imin, imout, smil.SquSE(5))
```

which is consistent with the second line.

C. Image Expression Atomization

The SMIL library massively uses operator overloading, which eases image manipulation such as arithmetic operations etc. This allows to write expressive codes, but corresponds internally to nested calls. Our compiler manages this issue, sometimes by generating intermediates variables. Operands can also be API calls. Since FREIA calls do not return images pointers, SMIL arithmetic expressions are decomposed into their atomic forms. RedBaron helps us by taking care of operators precedence. For instance, the following SMIL expression:

```
out = in0 * in1 + ((in2 - in4) | (in5 & in1)
```

is transformed in the five following FREIA operator calls, according to the semantics of the operators:

```
freia_aipo_mul(tmp0, in0, in1);
freia_aipo_sub(tmp1, in2, in4);
freia_aipo_and(tmp2, in5, in1);
freia_aipo_or(tmp3, tmp1, tmp2);
freia_aipo_add(out, tmp0, tmp3);
```

Four intermediate image variables are added.

D. Dealing with API variations

SMIL and FREIA, being both mathematical morphology libraries, provide relatively close APIs: function names and parameters are similar, which eases the conversion. The remaining differences must nonetheless be taken care of.

1) Structuring elements: One example of an API variation between SMIL and FREIA is the structuring element data structure. A structuring element is a data structure describing a neighborhood for stencils. They are widely used in mathematical morphology operators.

In FREIA, structuring elements are boolean integer arrays and only take the first neighbors into account.

Operating on a larger neighborhood amounts then to iterating several times over the operation. In SMIL, the corresponding data structure is more complex: it involves in particular a `std::vec` of neighbors and a integer size. When converting a SMIL morphological operator into FREIA, `smiltofreia` takes care of the size of the structuring element to generate a loop over the FREIA operator call. For instance, the following SMIL dilatation with a structuring element of size 5:

```
smil.dilate(imin, imout, smil.SquSE(5))
```

is translated into the following FREIA code:

```
#define e0 SMILTOFREIA_SQUSE
#define e0_s 5
  freia_cipo_dilate_generic_8c(imout, imin, e0, e0_s);
```

Common-used structuring elements are stored in a separate `smil-freia.h` compatibility header as constants. Preprocessor macros are used to mimic the SMIL data structure for keeping track of the structuring element size, while allowing our source-to-source compiler to fully forward-substitute these variables.

2) Altering FREIA: During the development of `smiltofreia`, we came to realize that some transformations would be eased by adapting directly the FREIA API. For example, part of the FREIA API implies that we always use a default structuring element, whereas the SMIL equivalent accepts arbitrary ones. Functions having the following signature

```
void freia_cipo_dilate(freia_data2d *imout,
                       freia_data2d *imin,
                       uint32_t size);
```

only use square structuring elements (a boolean array of nine ones), but of arbitrary size: internally, a loop around `freia_aipo_dilate` is used.

Instead of adding additional constraints into `smiltofreia` inputs, we can alter and improve FREIA to support such cases. New functions have thus been added to FREIA to bring it closer to the SMIL API:

```
void freia_cipo_dilate_generic_8c(freia_data2d *imout,
                                  freia_data2d *imin,
                                  const int32_t *se,
                                  uint32_t size);
```

These new functions ease the generation of FREIA code from SMIL. Another example is the `smil.mask()` function, which had no direct equivalent in FREIA prior to this work. A workaround combining two existing FREIA functions but adding temporary images would be easier to implement, although at the expense of the global performance.

V. Evaluation

We evaluated our compilation chain using seven image processing FREIA applications taken from [4], which we rewrote entirely in idiomatic SMIL Python. We are thus able to compare the performance of the output of our `smiltofreia` compiler to the original application. FREIA applications can also be further optimized by the source-to-source compiler PIPS. This optimized version is compared below to the non-optimized one. We execute these applications using the SMIL backend of FREIA on an Intel IvyBridge Core i7-3820 CPU. A summary of our evaluation methodology is represented Figure 9.

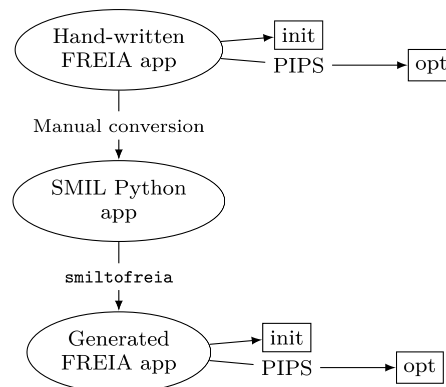

Figure 9. Evaluation methodology

The execution times of our seven applications are available in Table I. This table represents the figures of the original FREIA applications, their optimized version using PIPS and their port on SMIL Python. Here, Column “SMIL” refers to the Python applications; Column “Hand-written” is the original FREIA applications, which have been rewritten in SMIL and executed in their original form in Sub-column “init” or optimized by PIPS in Sub-column “opt”. Similarly, Column “Generated” represents the output of our `smiltofreia` compiler. Two sub-columns, “init” and “opt”, show original and optimized execution times.

Apps	**SMIL**	**FREIA**			
	Python – C++	Hand-written		Generated	
		init	opt	init	opt
anr999	0.69	0.63	0.47	0.64	0.47
antibio	26.0	24.7	24.6	25.0	25.0
burner	49.5	8.5	8.37	8.56	8.36
deblocking	29.8	30.3	30.0	30.0	29.9
licensePlate	2.68	4.31	1.93	4.29	1.95
retina	8.54	7.7	6.58	6.7	6.61
toggle	1.58	1.53	1.53	1.35	1.39

Table I
Execution times (ms) of SMIL and original FREIA versions of a set of applications

More interesting is the representation of the speedups between original SMIL Python applications and PIPS-optimized FREIA applications, which can be seen in Figure 10. The “anr999”, “licensePlate” and “retina” applications benefit from the conversion to FREIA and the subsequent PIPS optimizations, whereas “antibio”, “deblocking” and “toggle” are already competitive in SMIL Python. The “burner” application is a specific case involving a morphological operator called *geodesic reconstruct by closing* that is, in SMIL, implemented using complex (and in this case inefficient) data structures such as hierarchical queues. The FREIA version is simpler and only involves highly-optimized classical operators, which explains the huge performance gap between the two of them. On average, these results show that `smiltofreia` generated FREIA is $\times 1.5$ faster than original SMIL code and has performance very close to hand-written, PIPS-optimized FREIA code.

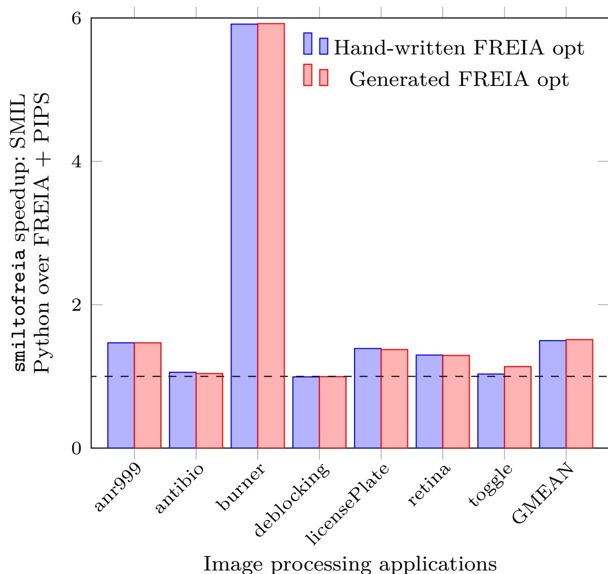

Figure 10. SMIL Python over FREIA + PIPS, handwritten and generated by `smiltofreia`

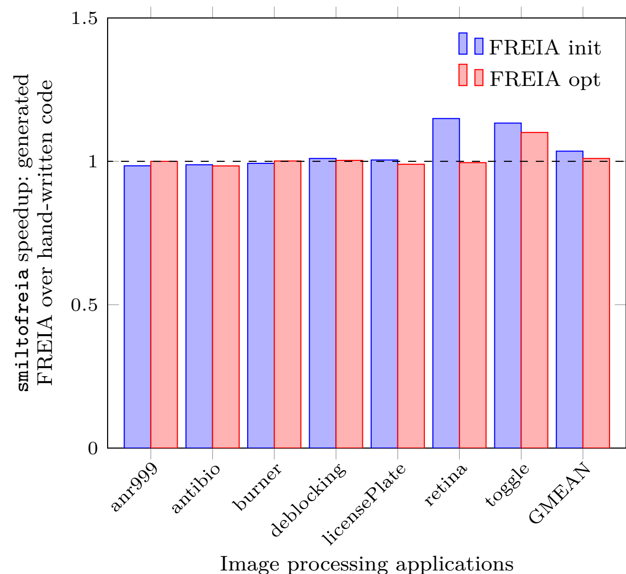

Figure 11. `smiltofreia` generated FREIA code compared to hand-written FREIA, with and without PIPS optimizations

The relative performance of FREIA code between original hand-written applications and `smiltofreia` generated FREIA code, with and without PIPS optimizations, is represented Figure 11. The performance of the two versions is very similar, except for "toggle" and "retina". The discrepancies mainly come from the original translation from FREIA to SMIL Python: in the "retina" case, two FREIA functions were partly converted to match SMIL API. This partial conversion leads to increased performance when converting back to FREIA. The "toggle" application benefits greatly from the `freia_aipo_mask` operator introduced to match SMIL API. On average, `smiltofreia` generated FREIA code is $\times 1.03$ faster than hand-written FREIA, and PIPS-optimized `smiltofreia` output is the same as PIPS-optimized FREIA.

These table and plots show that SMIL applications easily benefit from our FREIA compilation toolchain with no performance impairment compared to hand-written code. What's more, SMIL applications can now directly target the whole set of FREIA hardware backends (FPGAs, manycore and GPUs) without modifying the input code.

VI. Related Work

Python is a versatile general-purpose programming language especially used for fast application prototyping. However, the Python interpreter performance pales compared to native compiled languages such as C or C++. Other research projects use subsets of Python as inputs to accelerate applications on several hardware targets. Cython [2], which we already described in subsection III-B, is both an interfacing tool between Python and C and a Python-to-C compiler. Yet Cython output is overly complex and implements parts of the Python interpreter. Pythran [26] is a Python-to-C++ compiler for scientific programs targeting multicore CPUs with SIMD extensions. Pythran generates C++ source code or shared libraries from Python code, which can be reused directly in a Python application. Numba [27] is a Python-to-LLVM JIT compiler dedicated to accelerating Python code, but still needs the Python interpreter to work. Similarly, Parakeet [28] is a JIT compiler to parallelize Python code on CPUs or GPUs. Theano [29] and Tensorflow [30] are DSL compilers for Python linear algebra applications for deep learning which generate optimized C++ or CUDA. Image processing compilers such as Halide [31] and PolyMage [32] also aim at performing domain-specific optimizations while still offering ease of programming through high-level DSLs. While PolyMage is still limited to CPU execution, Halide is able to generate OpenCL and CUDA code for running onto GPUs.

VII. Conclusion

We study in this paper a static to dynamic DSL compilation scheme that preserves programmability. We developed a fully-functional implementation, called `smiltofreia`, that converts image processing applications written in a high-level DSL running on a small set of hardware targets to a lower-level DSL that supports a greater number of backends. Our proposed solution relies on transformations on an AST-like data structure of the original application for generating corresponding calls and variable declarations in the output language. Since the source DSL is embedded in Python, and the target DSL is embedded in C, typing and polymorphism have been taken care of. Experimental results on a set of seven image processing applications show that generated code is competitive with its input in terms of execution times. Moreover, additional target-specific optimization provided by the source-to-

source compiler PIPS can lead to improved performance for our target DSL applications.

To sum up, our SMIL compiler can port high-level image processing applications onto a variety of hardware accelerators, such as GPUs, manycore or FPGA accelerators, by reusing another lower-level image processing DSL as a target. We benefit from the existing compiler toolchain of our target DSL to provide hardware-specific optimizations.

Keywords: dynamic language, DSL, compilation, image processing

Acknowledgments: This work was funded by Investissements d'Avenir as part of the CAPACITES project.

References

[1] Matthieu Faessel. *SMIL: Simple (but efficient) Morphological Image Library*. 2011. URL: http://smil.cmm.mines-paristech.fr/.
[2] *Cython: C-Extensions for Python*. URL: http://cython.org/.
[3] Michel Bilodeau et al. *FREIA: FRamework for Embedded Image Applications*. French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne. 2008.
[4] Fabien Coelho and François Irigoin. "API Compilation for Image Hardware Accelerators". In: *ACM Transactions on Architecture and Code Optimization* (Jan. 2013).
[5] Herb Sutter. *Welcome to the Jungle*. 2011. URL: http://herbsutter.com/welcome-to-the-jungle/.
[6] *OpenMP: Open Multi-Processing*. URL: http://openmp.org/wp/.
[7] Khronos Group. *OpenCL: The open standard for parallel programming of heterogeneous systems*. URL: https://www.khronos.org/opencl/.
[8] The MPI Forum. *The Message Passing Interface*. URL: http://www.mpi-forum.org/.
[9] Matthieu Faessel and Michel Bilodeau. "SMIL: Simple Morphological Image Library". In: *Séminaire Performance et Généricité, LRDE*. Villejuif, France, Mar. 2013. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-00836117.
[10] *Auto-vectorization in GCC*. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.
[11] *Swig: Simplified Wrapper and Interface Generator*. URL: http://www.swig.org/.
[12] *CMake: Build, Test and Package Your Software*. URL: https://cmake.org/.
[13] Christophe Clienti. *Fulguro image processing library*. Source Forge. 2008.
[14] Christophe Clienti, Serge Beucher, and Michel Bilodeau. "A System On Chip Dedicated To Pipeline Neighborhood Processing For Mathematical Morphology". In: *European Signal Processing Conference*. Aug. 2008.
[15] Philippe Bonnot et al. "Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA". In: *Design Automation and Test in Europe*. IEEE, Dec. 2008.
[16] Benoit Dupont de Dinechin, Renaud Sirdey, and Thierry Goubier. "Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor". In: *Procedia Computer Science 18*. 2013.
[17] Thierry Goubier et al. "ΣC: A Programming Model and Language for Embedded Manycores". In: 2011.
[18] Pascal Aubry et al. "Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor." In: *ICCS*. Ed. by Vassil N. Alexandrov et al. Vol. 18. Procedia Computer Science. Elsevier, 2013, pp. 1624–1633.
[19] François Irigoin, Pierre Jouvelot, and Rémi Triolet. "Semantical interprocedural parallelization: an overview of the PIPS project". en. In: *Proceedings of ICS 1991*. ACM Press, 1991, pp. 244–251. ISBN: 0897914341. DOI: 10.1145/109025.109086. URL: http://portal.acm.org/citation.cfm?doid=109025.109086 (visited on 05/21/2014).
[20] Pierre Guillou, Fabien Coelho, and François Irigoin. "Automatic Streamization of Image Processing Applications". In: *Languages and Compilers for Parallel Computing*. 2014.
[21] *Redbaron: Bottom-up approach to refactoring in python*. URL: http://github.com/PyCQA/redbaron.
[22] Laurent Peuch. *RedBaron, une approche bottom-up au refactoring en Python*. Oct. 2014.
[23] *inspect — Inspect live objects*. URL: https://docs.python.org/3/library/inspect.html.
[24] *ast — Abstract Syntax Trees*. URL: https://docs.python.org/3/library/ast.html.
[25] *Baron: a Full Syntax Tree library for Python*. URL: https://github.com/PyCQA/baron.
[26] Serge Guelton et al. "Pythran: enabling static optimization of scientific Python programs". In: *Computational Science & Discovery* (2015).
[27] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: ACM, 2015, 7:1–7:6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162. URL: http://doi.acm.org/10.1145/2833157.2833162.
[28] Alex Rubinsteyn et al. "Parakeet: A Just-In-Time Parallel Accelerator for Python". In: Berkeley, CA: USENIX, 2012.
[29] James Bergstra et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Python for Scientific Computing Conference (SciPy)*. Austin, TX, June 2010.
[30] M. Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *ArXiv e-prints* (Mar. 2016). arXiv: 1603.04467 [cs.DC].
[31] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *PLDI 2013* (June 2013), p. 12.
[32] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines". en. In: ACM Press, 2015, pp. 429–443. ISBN: 9781450328357. DOI: 10.1145/2694344.2694364. URL: http://dl.acm.org/citation.cfm?doid=2694344.2694364.