Noname manuscript No.
(will be inserted by the editor)

A Dynamic to Static DSL Compiler for Image Processing Applications

**Pierre Guillou · Benoît Pin ·
Fabien Coelho · François Irigoin**

the date of receipt and acceptance should be inserted later

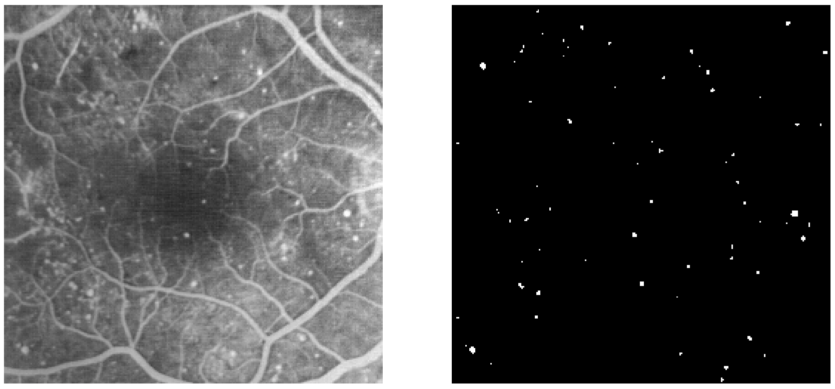
Abstract High-level interpreted programming languages, such as Python, are widely used because of their concise syntax and dynamic type system, which allow programmers to efficiently develop applications. However, they cannot offer the same guarantees provided by lower-level languages such as C in terms of portability on embedded systems. Is it possible for dynamic applications to benefit from lower-level compilation toolchain in order to increase their portability onto specialized hardware targets? We present in this paper (1) a methodology to convert a dynamic Domain-Specific Language (DSL) into a static one that preserves programmability, (2) a working implementation that takes care of types, memory allocation, polymorphism and API adaptation between the two DSLs, (3) and experimental results on portability and performance that show the efficiency of our approach. We illustrate our methodology with two image processing libraries: SMIL and FREIA. The SMIL library is a 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. The approach we suggest here 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.

Keywords Dynamic Language · DSL · Compilation

MINES ParisTech, PSL Research University, France
E-mail: *firstname.lastname*@mines-paristech.fr

1 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 promises and may be available to consumers in the next decade. Figure 1 depicts an image processing application that detects macular degeneration on a retina, speeding up medical diagnoses. To support this innovative field, new image processing libraries are being developed and compete in terms of performance and programmability. Among them, OpenCV [1], GEGL [2], ImageMagick [3] or NumPy [4] are the most well known. These libraries often provide APIs in high-level programming languages, such as Python, in order to offer good programmability.



(a) Input image (b) Output image

Fig. 1: Example of an image processing application: automatic detection of retina damage

Python is a general-purpose programming language with several key features improving programmability: high-level abstractions, dynamic type system, simple memory management with garbage collection and concise and easy-to-learn syntax. It can support several programming paradigms, such as imperative, functional or object-oriented programming. However, Python programs are limited by the interpreter, which causes slower executions and needs more memory than native languages such as C or C++ [5, 6]. The interpreter also limits the parallelism of Python programs through the Global Interpreter Lock [7], which prevents using more than one thread. Fortunately, Python can easily be interfaced with low-level C or C++ code through wrappers to ensure native performance on parallel architectures. As a consequence, image processing libraries are often written C or C++, and provide Python bindings to increase programmability.

However, in our hardware jungle era [8], 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 [9] for shared memory, OpenCL [10] for heterogeneous platforms or MPI [11] 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 compiling high-level interpreted DSLs into lower-level but more portable ones. We validate our approach with two image processing interfaces considered as DSLs, SMIL [12] and FREIA [13, 14], supporting each a different set of hardware targets and providing different levels of programmability. We evaluate our methodology on a set of seven image processing applications.

2 Context

Mathematical Morphology is an image processing theory based on lattice theory initiated in the 1960s [15, 16]. Common applications of this theory consist in detecting geometric structures in images, or partitioning an image in regions. 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.

2.1 The SMIL library

```python
import smilPython as smil

imin = smil.Image("input.png")  # read from disk
imout = smil.Image(imin)        # allocate imout
smil.dilate(imin, imout)        # morphological dilatation
imout.save("output.png")        # write to disk
```

Fig. 2: Morphological dilatation in SMIL

SMIL (*Simple Morphological Image Library*) [12, 17] is a 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 GCC compilation [18] and OpenMP parallelization [9];

– ease of programming, through several Swig [19] auto-generated interfaces to higher-level programming languages such as Python;
– portability on several CPUs and operating systems, using the CMake [20] compilation toolchain;
– maintainability and extensibility, through C++ templates and functors.

Figure 2 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.

2.2 The FREIA framework

```c
#include "freia.h"

int main(void) {
  // initializations...

  freia_data2d
    *imin = freia_common_create_data(/*...*/), // allocate
    *imout = freia_common_create_data(/*...*/);
  freia_common_rx_image(imin, /*...*/);  // read from disk
  freia_cipo_dilate(imout, imin, 8, 1);  // morpho dilatation
  freia_common_tx_image(imout, /*...*/); // write to disk
  freia_common_destruct_data(imin);      // free memory
  freia_common_destruct_data(imout);

  // shutdown...
}
```

Fig. 3: Morphological dilatation in FREIA (excerpt)

FREIA (*Framework for Embedded Image Applications*) [13] 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 [21];
– FPGAs with the SPoC [22] and Terapix [23] backends;
– manycore CPUs such as the Kalray MPPA [24] with Sigma-C [25, 26], a dataflow programming language;
– GPUs using OpenCL [10].

Used in combination with our in-house C source-to-source compiler framework PIPS [27], FREIA applications can be further optimized at the image operator level for the designated hardware target [14, 28]. Performed image optimizations include complex operator unfolding, temporary variable elimination, common sub-expression elimination, backward and forward copy propagation, operator aggregation, and target-specific code generation.

Figure 3 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 afterwards.

2.3 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.

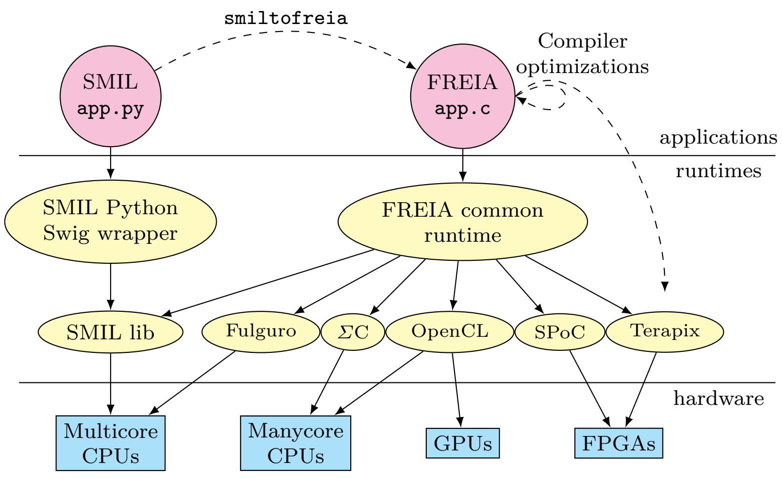


Fig. 4: Compiler toolchain diagram

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 work already 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 4 represents our compilation chain highlighting the benefits of `smiltofreia` in terms of portability.

3 Manipulating and accelerating Python code

In order to analyze and convert SMIL Python applications into FREIA C, we tried two Python tools: RedBaron [29, 30], a Python refactoring framework, and Cython [31, 32], a Python-to-C compiler.

The Python standard library itself provides low-level tools to parse and query Python code. Among them, *inspect* [33], which can be used to inspect and modify running Python code, and *ast* [34], for manipulating Python code Abstract Syntax Trees.

3.1 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 [35] 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 is an invariant transformation:

```
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 2 is represented in Figure 5. 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.

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 2, Line 4, into an image copy is as simple as rewriting the content of the corresponding node

```
fst[3] = "imout = imin"
```

The main purpose of RedBaron is to help refactoring Python code, such as generating classes, methods, or renaming variables or functions in one or several files. A canonical example of this usage can be found in Figure 6: variable `imin` is here renamed in `in`.

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

```
{"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"}}
      ]}]}
```

Fig. 5: FST of `smil.dilate(imin, imout)`

```python
from redbaron import RedBaron
red = RedBaron("smil.dilate(imin, imout)")
for node in red.find_all("NameNode", value="imin"):
    node.value = "in"
print(red.dumps())  # smil.dilate(in, imout)
```

Fig. 6: Example of using RedBaron to rename a variable

3.2 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. We wrapped a subset of the FREIA API in Python using the Cython extension system, and we used RedBaron to convert SMIL applications into FREIA Python. 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 optimizations. Cython introduces a lot of new variables and functions, 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. Moreover, the generated code depends on an external library implementing a Python run-time environment, which may not have been ported on every FREIA hardware target. The Cython approach, which works well for interfacing Python and C code and hence accelerating

```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;
}
```

Fig. 7: Actual C call to FREIA dilatation after Cython compilation

Python applications, is thus not recommended for post-processing the generated C code.

4 The `smiltofreia` SMIL Python to FREIA C compiler

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

int main(int argc, char *argv[]) {
  smil_freia_initialize(argc, argv); // initializations
  freia_data2d *imin;
  imin = freia_create_image();
  freia_data2d *imout;
  imout = freia_create_image();
#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_destruct_image(imout);
  freia_destruct_image(imin);
  smil_freia_finalize(); // shutdown
  return 0;
}
```

Fig. 8: Simplified FREIA C output of our compiler for Figure 2

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 in Figure 8. Some compatibility code and wrapper functions have been placed in a dedicated `smil-freia.h` header file, which explains the main differences with the original FREIA dilatation code in Figure 3.

Algorithm 1: General description of a `smiltofreia` execution over a SMIL Python application

```
Input: src — SMIL Python application source code
Output: dst — corresponding FREIA C application code
  /* Preprocessing: ensure there is a main function          */
1 src = preprocess(src);
  /* Get and process the RedBaron FST                        */
2 fst = RedBaron.generate_fst(src);
3 dst = generate_c(fst, Scope());
4 dst.insert_freia_includes();
  /* Apply clang-format to prettify the C output code        */
5 dst.clang_format();
6 print(dst);
```

Algorithm 1 presents a broad overview of the workings of our compiler. The RedBaron tool is used to get the *Full Syntax Tree* of the input code. Our compiler operates on the nodes of this FST to type variables and generate C code. A pre-processing pass can generate a missing `def main()` function around the input code instructions as a normalized entry point for our applications. To prettify the generated C code, an optional application of the formatting tool `clang-format` [36] is performed.

A more in-depth description of the main routine of our compiler is available in Algorithm 2. This algorithm shows the recursive approach taken to generate the C code corresponding to a FST node. Each FST node is processed according to its RedBaron type, from top-level nodes, which represent Python module-level constructs such as classes, function declarations, global variable declarations or instructions, to bottom-level ones (identifiers, operators, constants or formatting data). For instance, dealing with a Python instruction block consists in processing each sub-node.

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 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

Algorithm 2: `smiltofreia` RedBaron node transformation function

```
Input: src — SMIL Python function FST
Output: dst — corresponding FREIA C function
/* recursive bottom-up code generation and typing */
def generate_c(node, scope):
    cc = Ccode();
    switch node.type() do
        case BlockNode do
            foreach subnode of node do
                cc.add(generate_c(subnode, scope));
            end foreach
        end case
        case AssignNode do
            rcode = generate_c(node.rightarg, scope);
            var, type = node.leftarg, node.rightarg.rettype;
            if var $\in$ scope then
                assert(scope.get_type(var) == type); // check static type
            else
                scope.add_type(var, type);
                cc.add("%s %s;", type, var); // variable declaration
            end if
            cc.add("%s = %s;", var, rcode);
        end case
        case BinOpNode do
            /* in1 $\otimes$ in2 */
            op, in1, in2 = node.value, node.first, node.second;
            node.out, node.rettype = gen_id(), get_rettype(op, in1, in2);
            scope.add_type(node.out, node.rettype);
            cc.add("%s %s;", node.rettype, node.out);
            cc.add("%s = %s;", node.out, gen_init(node.out, node.rettype));
            cc.add("%s(%s,%s,%s);", freia_call(op), node.out, in1.out, in2.out);
        end case
        case WhileNode do
            /* deal with test and loop body separately */
            test = generate_c(node.test, scope);
            body = generate_c(node.value, scope);
            cc.add("while(%s) {%s}", test, body);
        end case
        [...];
    end switch
    return cc;
```

without other Python modules is supported, and the type of all variables must be statically inferable.

4.1 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.

4.2 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 transform the first line version into the second line version.

4.3 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 three-address code 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.

4.4 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.

4.4.1 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 an 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 while keeping track of the structuring element size; this allows our source-to-source compiler to fully forward-substitute these variables.

4.4.2 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

```
freia_status freia_cipo_dilate(freia_data2d *imout, freia_data2d *imin,
                               uint32_t size) {
  unsigned int i;
  int32_t square_strelt[9] = { 1, 1, 1, 1, 1, 1, 1, 1, 1};
  freia_aipo_dilate_8c(imout, imin, square_strelt);
  for (i = 1; i < size; i++)
    freia_aipo_dilate_8c(imout, imout, square_strelt);
  return FREIA_OK;
}
```

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.

5 Performance Evaluation

We evaluated our compilation chain using seven image processing FREIA applications taken from [14], which we rewrote entirely in idiomatic SMIL C++, on the one hand, and SMIL Python, on the other hand. We are thus able to compare the performance of the output of our `smiltofreia` compiler to the original application.

We first compare the number of lines of code used for writing these seven applications in FREIA and in SMIL Python. The results are assembled in Table 1, which shows that SMIL applications are on average three times shorter than FREIA ones, mainly due to the concise syntax and the memory management model of Python. The SMIL C++ applications have roughly the same number of lines of code than their Python counterparts, but the C++ syntax is more complex and intrusive.

Apps	#LoC		Gain
	SMIL	FREIA	
anr999	23	88	3.8
antibio	61	172	2.8
burner	55	140	2.5
deblocking	74	162	2.2
licensePlate	37	202	5.5
retina	40	133	3.3
toggle	40	144	3.6
GMEAN	44.4	144.7	3.2

Table 1: Number of lines of SMIL and FREIA code for seven image processing applications

5.1 General-purpose CPU

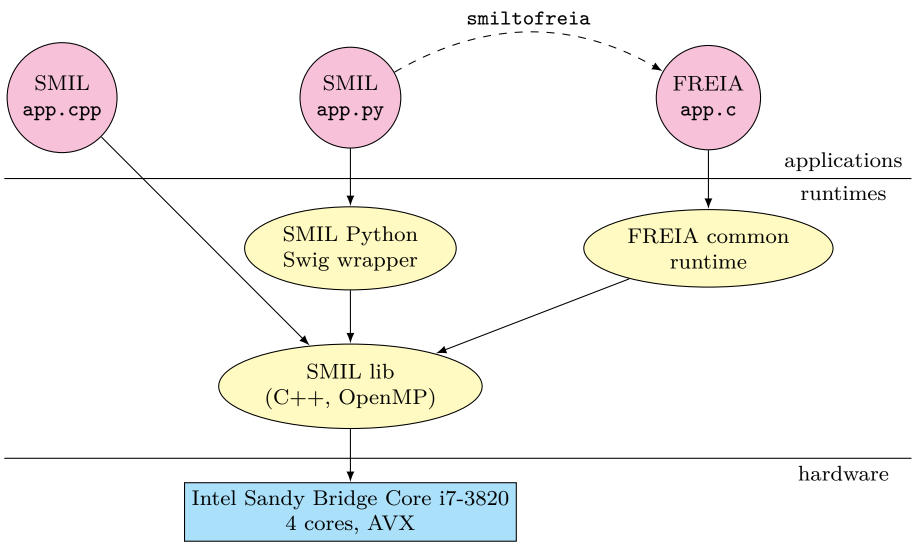


Fig. 9: Compiler toolchain diagram

We then compare the execution times of our seven applications, written in SMIL C++, SMIL Python and FREIA, and the impact of using `smilto-freia` to convert SMIL Python applications into FREIA. 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 executed these applications using the SMIL backend of FREIA on an Intel Sandy Bridge Core i7-3820 CPU. Figure 9 represents the simplified software used for this evaluation. All applications are in the end calling the SMIL library and are being executed on the same hardware.

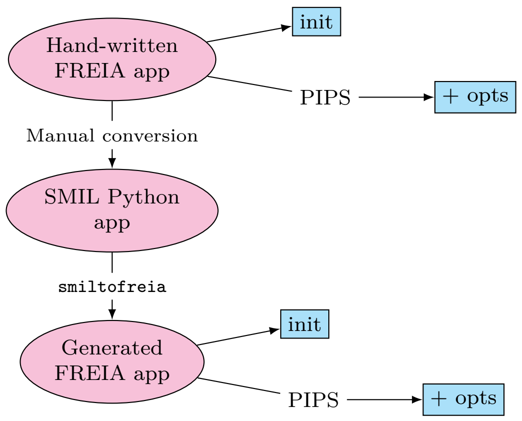


Fig. 10: Evaluation methodology

As shown in Figure 10, FREIA applications are tested with and without PIPS source-to-source optimizations. Execution times of original and `smiltofreia`-generated applications are similarly compared. The applications are executed with a fixed number of OpenMP threads equal to the number of physical cores of the current CPU, which has four of them. Input images are scaled to a 4K resolution of 3840×2160 to ensure correctly fed threads.

Apps	**SMIL**		**FREIA**			
	C++	Python	Hand-written		Generated	
			init	+ opts	init	+ opts
anr999	80.9	84.2	81.5	61.8	81.8	63.2
antibio	2795	2800	3470	3540	3500	3500
burner	1220	1210	1612	1630	1630	1585
deblocking	828	852	864	866	860	861
licensePlate	196	198	195	84.3	195	84.3
retina	1010	1020	1110	964	1115	1105
toggle	47.0	47.0	47.4	48.1	46.6	46.6

Table 2: Execution times (ms) of seven image processing image applications written in SMIL (C++ and Python) and FREIA (Column “Hand-written”). The SMIL Python applications are converted to FREIA using the `smiltofreia` compiler and their execution time is measured as well (Column “Generated”). FREIA applications are executed with and without compile-time optimizations.

The resulting execution times of our seven applications are available in Table 2. 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 both the C++ and 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 “+ opts”. Similarly, Column “Generated” represents the output of our `smiltofreia` compiler. Two sub-columns, “init” and “+ opts”, show original and optimized execution times.

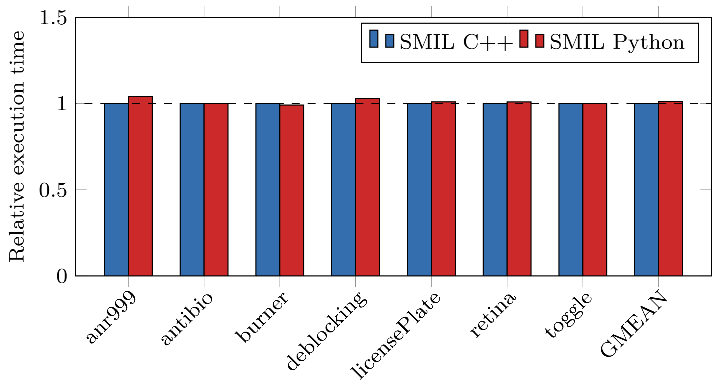


Fig. 11: Comparison of the relative execution times of seven image processing applications written in SMIL using the C++ and the Python API

The three plots in Figure 11, Figure 12 and Figure 13 serve as a visual representation of the data in Table 2. Figure 11 compares the execution times of SMIL C++ and SMIL Python applications, and shows a minimal overhead, amounting to less than 2%, when the Python language is used. The Python wrapper has therefore little to no impact on the global performance of our applications. SMIL applications in Python are as fast as SMIL C++ ones, while benefiting from the Python syntax.

We then compare the execution times of our SMIL Python applications, before and after being converted to FREIA by our `smiltofreia` compiler. We also tested applying our PIPS optimizing compiler on the generated FREIA code. Results are displayed in Figure 12. The “anr999”, “deblocking”, “licensePlate”, and “toggle” applications yield similar performance before and after being converted into FREIA. Source-to-source optimizations performed by PIPS have a major impact on the execution times of the “anr999” and “licensePlate” applications, especially by removing unnecessary copies between operations and merging identical computation sequences. The “antibio” and

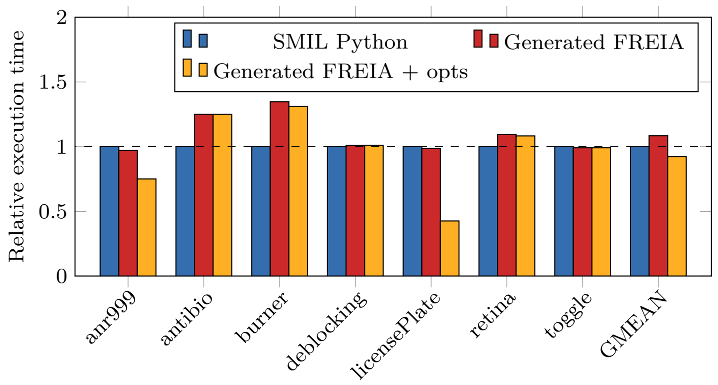


Fig. 12: Comparison of the relative execution times of seven image processing applications written in SMIL Python, executed using the Python interpreter, then converted into FREIA and executed with and without compile-time optimizations

"burner" FREIA applications are somewhat around 30% slower than their SMIL versions. This stems from the use of a peculiar mathematical morphology operator called *geodesic reconstruct by closing*, which has a coarse-grain implementation in SMIL, whereas in FREIA it is decomposed into several other simpler operators. The "retina" application is also affected by this issue. The SMIL implementation uses complex data structures, such as hierarchical queues, which are quite efficient on large images, as in the present case. Implementing algebraic optimizations [37, 38] to detect this coarse-grain operator in a sequence of FREIA atomic operators and replace it by the corresponding SMIL call can help to reduce this slowdown. On average, these result show that FREIA generated code is about 10% slower than SMIL Python code, although applying PIPS optimizations can lead to 10% faster code.

Figure 13 compares the execution times of the original FREIA applications to those generated from Python by `smiltofreia`. Generated FREIA applications perform on average as well as original ones. As seen in the previous Figure, applying PIPS optimizations leads to increased performance in the "anr999" and the "licensePlate" applications in both cases, whereas the generated version of the "retina" application does not seem to benefit from it. This can be explained by the introduction of compatibility code for bridging the gap between SMIL and FREIA, which in this case slows down the generated code. To mitigate these slowdowns, a small speedup of the generated version of the "toggle" application can be linked to the use of newly introduced FREIA functions, such as `freia_aipo_mask`, which allow more direct calls to SMIL functions in the FREIA API. We show through these figures that

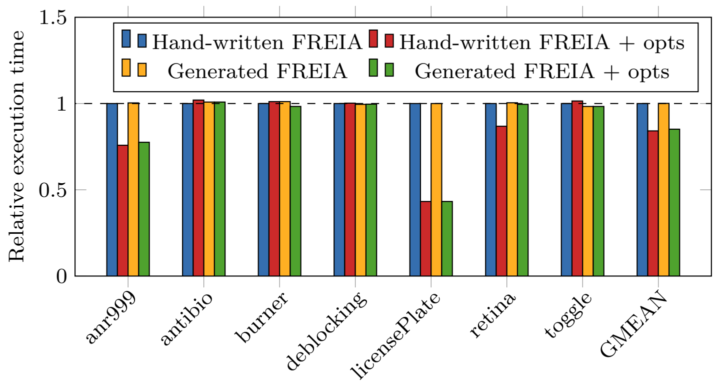


Fig. 13: Comparison of the relative execution times of seven image processing applications written in SMIL Python and converted into FREIA, and genuine hand-written FREIA applications, executed with and without compile-time optimizations

`smiltofreia`-generated FREIA applications perform very closely to original FREIA applications, with or without PIPS optimizations, the average execution times ratio amounting to less than 1% in both cases.

5.2 Specific hardware

Apps	**SPoC**				**Terapix**			
	Hand-written		Generated		Hand-written		Generated	
	init	+ opts	init	+ opts	init	+ opts	init	+ opts
anr999	1.8	0.1	1.8	0.1	0.7	0.4	0.7	0.4
antibio	46.4	5.1	46.4	5.2	17.9	7.5	17.9	9.8
burner	28.9	2.1	28.9	3.6	11.7	8.2	11.6	8.3
deblocking	2.0	0.9	2.4	1.1	0.9	0.3	1.0	0.3
licensePlate	7.5	0.4	7.5	0.4	1.6	1.2	1.6	1.2
retina	16.8	1.0	16.7	1.6	6.6	3.5	6.6	3.5
toggle	0.8	0.4	0.9	0.4	0.4	0.1	0.4	0.3

Table 3: Execution times (s) of seven image processing image applications written in SMIL Python and FREIA (Column “Hand-written”) and executed on the SPoC and Terapix image processing hardware accelerators. The SMIL Python applications are converted to FREIA using the `smiltofreia` compiler and their execution time is measured as well (Column “Generated”). All applications are executed with and without compile-time optimizations.

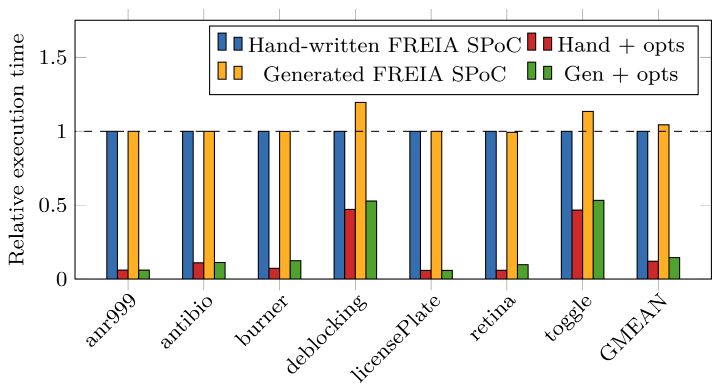


Fig. 14: Relative execution times of hand-written FREIA C and `smiltofreia`-generated FREIA applications executed on the SPoC embedded accelerator, with and without PIPS optimizations

Converting SMIL Python applications to FREIA is also a way to automatically port Python applications onto embedded accelerators dedicated to image processing. Thanks to FREIA portability, we executed our image processing applications onto two hardware accelerators implemented on FPGA: SPoC [22] and Terapix [23]. There is no Python interpreter available for these accelerators, nor any SMIL implementation, which prevent SMIL applications to be directly executed on these platforms. In Figure 14 and Figure 15, we compare the execution times of the original hand-written FREIA applications to the SMIL Python ones after being converted through `smiltofreia`. Raw results are also available in Table 3.

Results show that generated FREIA code performs identically to hand-written code on these accelerators, except for the “deblocking” and “toggle” applications. The corresponding slowdown stems from the original conversion of these applications from FREIA to SMIL Python: a FREIA operator that was not implemented in SMIL has been replaced by a combination of two SMIL operators. `smiltofreia` does not yet look for such a combination to regenerate the original FREIA operator. Due to these two applications, `smiltofreia`-generated code performs on average 4% slower on SPoC and 3% slower on Terapix.

We also studied the impact of PIPS optimizations and target-specific code-generation in both cases. These optimizations are a bit less effective for `smiltofreia`-generated code than for hand-written FREIA. One of the reasons of this slowdown again comes from the original rewriting of FREIA applications to SMIL Python. We translated FREIA application functions into idiomatic Python, and returned the produced image pointers instead of passing them by reference, thus preventing PIPS from performing some useful operator ag-

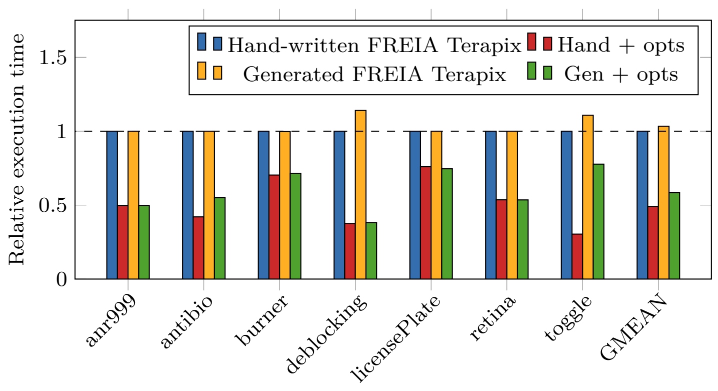


Fig. 15: Relative execution times of hand-written FREIA C and `smiltofreia`-generated FREIA applications executed on the Terapix embedded accelerator, with and without PIPS optimizations

gregation. These slowdowns amount to 20% for SPoC and 19% for Terapix, which is quite acceptable, considering the average speedups of 7.2× for SPoC and 1.8× for Terapix given by PIPS optimizations. Further improvements of `smiltofreia`-generated applications may be attainable by generating more idiomatic C code.

These tables and plots show that SMIL applications easily benefit from our FREIA compilation toolchain with minimal 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. Improved performance can still be achievable in the current case by using PIPS not only to clean the generated FREIA applications, but also by regenerating aggregated SMIL C++ calls, hence getting rid of the FREIA API and the C wrapper intermediary layers.

6 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 [31], which we already described in subsection 3.2, 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 [39] 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 [40] is a Python-to-LLVM JIT compiler dedicated to accelerating Python code, but still needs the Python interpreter to work. Similarly, Parakeet [41] is a JIT compiler to parallelize Python code on CPUs or GPUs.

Theano [42] and Tensorflow [43] are DSL compilers for Python linear algebra applications for deep learning which generate optimized C++ or CUDA. Image processing compilers such as Halide [44] and PolyMage [45] 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.

7 Conclusion

We study in this paper a static to dynamic DSL compilation scheme that preserves programmability. As a use case, 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, such as these provided by the source-to-source compiler PIPS, can lead to improved performance for target DSL applications.

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

References

[1] *OpenCV: Open Source Computer Vision.* URL: `http://opencv.org/`.

[2] *GEGL: GEneric Graphics Library.* URL: `http://www.gegl.org/`.

[3] *ImageMagick: Convert, Edit, Or Compose Bitmap Images.* URL: `https://www.imagemagick.org/script/index.php`.

[4] *NumPy: scientific computing with Python.* URL: `http://www.numpy.org/`.

[5] *Is Python faster and lighter than C++?* URL: `https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c/`.

[6] *Is C/C++ really faster than Python?* URL: https://news.ycombinator.com/item?id=9753366.

[7] *CPython Global Interpreter Lock.* URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[8] Herb Sutter. *Welcome to the Jungle.* 2011. URL: http://herbsutter.com/welcome-to-the-jungle/.

[9] *OpenMP: Open Multi-Processing.* URL: http://openmp.org/wp/.

[10] Khronos Group. *OpenCL: The open standard for parallel programming of heterogeneous systems.* URL: https://www.khronos.org/opencl/.

[11] The MPI Forum. *The Message Passing Interface.* URL: http://www.mpi-forum.org/.

[12] Matthieu Faessel. *SMIL: Simple (but efficient) Morphological Image Library.* 2011. URL: http://smil.cmm.mines-paristech.fr/.

[13] Michel Bilodeau et al. *FREIA: FRamework for Embedded Image Applications.* French ANR-funded project with ARMINES (CMM, CRI), THALES (TRT) and Télécom Bretagne. 2008.

[14] Fabien Coelho and François Irigoin. "API Compilation for Image Hardware Accelerators". In: *ACM Transactions on Architecture and Code Optimization* (Jan. 2013).

[15] Jean Serra. *Image analysis and mathematical morphology.* London New York: Academic Press, 1982. ISBN: 0126372411.

[16] Edward Dougherty. *An introduction to morphological image processing.* Bellingham, Wash., USA: SPIE Optical Engineering Press, 1992. ISBN: 081940845X.

[17] 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.

[18] *Auto-vectorization in GCC.* URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[19] *Swig: Simplified Wrapper and Interface Generator.* URL: http://www.swig.org/.

[20] *CMake: Build, Test and Package Your Software.* URL: https://cmake.org/.

[21] Christophe Clienti. *Fulguro image processing library.* Source Forge. 2008.

[22] 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.

[23] 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.

[24] 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.

[25] Thierry Goubier et al. “ΣC: A Programming Model and Language for Embedded Manycores”. In: 2011.

[26] 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.

[27] 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).

[28] Pierre Guillou, Fabien Coelho, and François Irigoin. “Automatic Streamization of Image Processing Applications”. In: *Languages and Compilers for Parallel Computing*. 2014.

[29] *Redbaron: Bottom-up approach to refactoring in python*. URL: `http://github.com/PyCQA/redbaron`.

[30] Laurent Peuch. *RedBaron, une approche bottom-up au refactoring en Python*. Oct. 2014.

[31] *Cython: C-Extensions for Python*. URL: `http://cython.org/`.

[32] S. Behnel et al. “Cython: The Best of Both Worlds”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 31–39. ISSN: 1521-9615. DOI: `10.1109/MCSE.2010.118`.

[33] *inspect — Inspect live objects*. URL: `https://docs.python.org/3/library/inspect.html`.

[34] *ast — Abstract Syntax Trees*. URL: `https://docs.python.org/3/library/ast.html`.

[35] *Baron: a Full Syntax Tree library for Python*. URL: `https://github.com/PyCQA/baron`.

[36] *clang-format:A tool to format C/C++/Java/JavaScript/Objective-C/Protobuf code*. URL: `http://clang.llvm.org/docs/ClangFormat.html`.

[37] Julien Zory. “Contribution à l’optimisation de programmes scientifiques”. PhD thesis. MINES ParisTech, Dec. 1999.

[38] Julien Zory and Fabien Coelho. “Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Codes”. In: *PACT: Parallel Architectures and Compilation Techniques*. Paris: IEEE, Dec. 1998, pp. 376–384.

[39] Serge Guelton et al. “Pythran: enabling static optimization of scientific Python programs”. In: *Computational Science & Discovery* (2015).

[40] 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`.

[41] Alex Rubinsteyn et al. “Parakeet: A Just-In-Time Parallel Accelerator for Python”. In: Berkeley, CA: USENIX, 2012.

[42] James Bergstra et al. "Theano: a CPU and GPU Math Expression Compiler". In: *Python for Scientific Computing Conference (SciPy)*. Austin, TX, June 2010.

[43] M. Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *ArXiv e-prints* (Mar. 2016). arXiv: `1603.04467 [cs.DC]`.

[44] 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.

[45] 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`.