PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow

Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, *Fellow, IEEE,* Wen-mei Hwu, *Fellow, IEEE*

Abstract—The exploding complexity and computation efficiency requirements of applications are stimulating a strong demand for hardware acceleration with heterogeneous platforms such as FPGAs. However, a high-quality FPGA design is very hard to create and optimize as it requires FPGA expertise and a long design iteration time. In contrast, software applications are typically developed in a short development cycle, with high-level languages like Python, which is at a much higher level of abstraction than all existing hardware design flows. To close this gap between hardware design flows and software applications, and simplify FPGA programming, we create PyLog, a high-level, algorithm-centric Python-based programming and synthesis flow for FPGA. PyLog is powered by a set of compiler optimization passes and a type inference system to generate high-quality design. It abstracts away the implementation details, and allows designers to focus on algorithm specification. PyLog takes in Python functions, generates PyLog intermediate representation (PyLog IR), performs several optimization passes, including pragma insertion, design space exploration, and memory customization, etc., and creates the complete FPGA system design. PyLog also has a runtime that allows users to run the PyLog code directly on the target FPGA platform without any extra code development. The whole design flow is automated. The evaluation shows that PyLog significantly improves FPGA design productivity and generates highly efficient FPGA designs that outperform highly optimized CPU implementation and state-of-the-art FPGA implementation by $3.17\times$ and $1.24\times$ on average.

Index Terms—FPGA, high-level synthesis, Python, design optimization.

1 Introduction

The last decade has witnessed an explosive growth of new applications in terms of quantity, diversity, and demands for computing capability and energy efficiency. As an example, deep learning algorithms, which have been shown to be successful in many domains, are driving the revolutionary changes in computer system design. According to Dean et al. [1], the number of machine learning papers on arXiv [2] doubles in less than two years, which has outpaced Moore's Law. The rapid growth of diverse applications poses immense challenges to many aspects of computing systems, including compiler, architecture, storage, etc.

These challenges have motivated new computing systems for the new decade. The FPGA-based computing platform is an emerging platform that provides reconfigurability, along with high performance, low latency, and high energy efficiency. FPGA's unique computing capability makes it a promising platform to tackle the rising computation challenges. FPGA accelerators have been deployed in both cloud servers and edge devices at scale. However, as FPGAs are getting used in increasing number of emerging applications and scenarios at a rapid pace, programming FPGA and optimizing FPGA design gradually become the main barriers in FPGA development.

- *Sitao Huang, Kun Wu, Hyunmin Jeong, Deming Chen, and Wen-mei Hwu are with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. E-mail: {shuang91, kunwu2, hyunmin2, dchen, w-hwu}@illinois.edu*
- *Chengyue Wang is with Zhejiang University/University of Illinois at Urbana-Champaign Institute, Zhejiang University, China. E-mail: cw19@illinois.edu*

Manuscript received December 15, 2020.

The most widely-adopted FPGA development flow today starts with programming FPGA at the register transfer level (RTL) in hardware description languages (HDL) such as Verilog and VHDL. Then designers use FPGA synthesis tools from FPGA vendors to synthesize RTL designs into FPGA bitstreams, which are used to configure FPGA. Programming FPGA at this level requires rich expertise in digital circuit design and the FPGA architecture. Besides, programming at this level is non-intuitive, error-prone, and hard to reuse code compared with modern programming languages, leading to long development, optimization, and verification cycles.

High-level synthesis (HLS) aims to simplify FPGA programming. Elevating the abstraction level of FPGA programming to that of C/C++/OpenCL [3], [4], [5], [6], HLS tools enable FPGA designers to express their algorithms in more familiar high-level languages. Developers are expected to use HLS pragmas or directives to guide the HLS tools to optimize and generate desired RTL design. Compared with RTL design flow, HLS allows FPGA developers to develop, optimize, verify, and reuse their design at a higher level, thereby greatly improving productivity. However, as C/C++/OpenCL are initially designed for general-purpose processors and start with an inherent sequential execution model inside each kernel/function definition of these languages, they are essentially different from the FPGA's fine-grained parallel processing nature (OpenCL model can describe parallel work-items but it is at thread-granularity and not very well supported in current HLS tools). The existing HLS tools are also designed in a way that accommodates the sequential execution model of input languages.

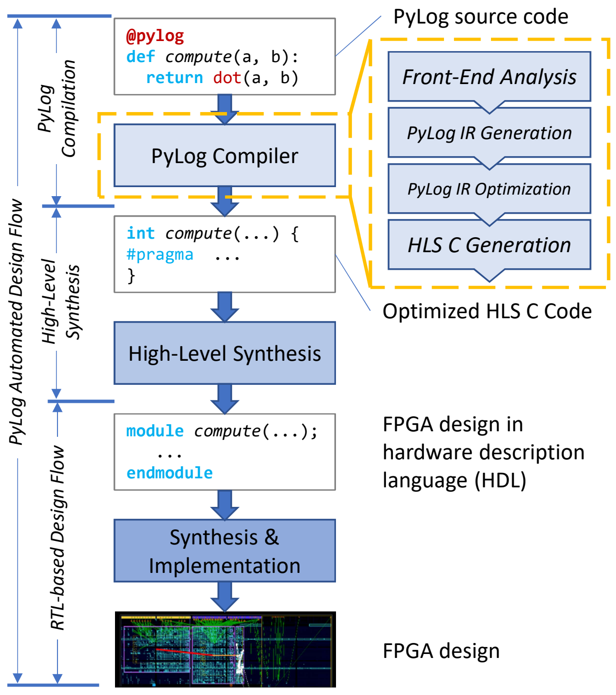


Fig. 1. FPGA Design Flow with PyLog.

As a compromise between the HLS programming model and the FPGA execution model, HLS users often manually annotate their code with HLS pragmas or directives to give HLS compiler hints on parallelism and desirable synthesis approaches. These pragmas have a significant impact on the performance and energy efficiency of the synthesis output. Oftentimes code transformation and rewriting are also needed to improve the outcome. In the end, the quality of synthesized design from HLS highly depends on how the code is written and how the HLS pragmas are added to the code. This requires a considerable amount of engineering time to iteratively adjust the source code and pragmas used. Complicated applications or thorough optimizations may lead to long source code that is difficult to read and maintain. For example, the HLS C code for convolution kernel from CHaiDNN [7] library has nearly 8,000 lines, which is much longer than the well-optimized convolution code for CPU or GPU.

Apart from the difficulties in creating and optimizing designs with current FPGA programming flow, the gap in the abstraction level between application programming and FPGA programming is another challenge in FPGA design. Applications are typically developed with languages at a much higher level of abstraction, where the programming models and styles focus more on describing the algorithm itself, instead of low-level implementation details. In the current HLS flow, when there is a need to accelerate the application, FPGA developers typically need to first lower the abstraction level of the application, re-implement the application in plain C code, and use it as the starting point for HLS. This lowering step is time-consuming and error-prone, and it also makes the FPGA design cycle longer.

These challenges in current FPGA design flows urge us to further elevate the abstraction level of FPGA programming. Among the existing programming languages, Python is one of the most popular and widely-used languages. It has been well adopted in various domains such as machine learning, scientific computing, data analysis, education, etc. Python is also easy to learn. We propose PyLog, an algorithm-centric Python-based programming and synthesis flow for FPGA. PyLog uses general Python compatible syntax, and it provides a set of handy low-level and high-level built-in operators that are capable of describing most of the common computation patterns in a natural way.

The PyLog compiler takes Python code as input, and compiles the code into optimized HLS-synthesizable C code with HLS pragmas. Fig. 1 shows the FPGA design flow with PyLog. We design PyLog in a way that the Python language allows the developers to focus on algorithm and computation flow description without much implementation details, while the PyLog compiler takes over the traditional FPGA developers' burden of exploring possible implementations and optimizations. The functional nature of the Python language also preserves some algorithm-level design information that is helpful for PyLog analysis and transformation. In the PyLog flow, algorithm and implementation are separated as much as possible. The goal of this design is to relieve FPGA programmers from manual design tuning while giving the PyLog compiler maximum information about computation patterns and therefore maximum freedom of design optimization. PyLog will be open-sourced for future research in this field.

The key contributions of this work are:

- We design and implement PyLog, a high-level Python-based programming and synthesis flow for FPGA, which greatly simplifies FPGA programming. The expressiveness of Python allows developers to achieve high design quality with much fewer lines of code compared with previous C/C++ based high-level synthesis flows.
- PyLog compiler is an ahead-of-time compiler and it is capable of doing various types of program analysis and optimizations. It features PyLog intermediate representation, type inference engines, and a set of compiler optimizations, which are all designed to create highly efficient FPGA systems.
- PyLog provides a set of high-level operators that ease algorithm description. These operators are general enough to describe computation patterns across different domains, and their implementation can be configured and optimized by PyLog to meet different design requirements.
- PyLog automatically generates optimized design from high-level algorithm specification, based on hardware resource constraints. Evaluation shows that PyLog generates highly efficient FPGA designs that outperform highly optimized CPU implementation and state-of-the-art FPGA implementation by 3.17× and 1.24× on average.

The rest of the paper is organized as follows. Section 2 reviews related works in literature. Then, the paper presents PyLog programming model in detail in Section 3, and elaborates on PyLog compiler design in Section 4. Section 5 evaluates PyLog with several real-world workloads. Finally, Section 6 concludes the paper.

2 Related Works

Recently there are growing interests in the research community on high-level programming languages for FPGAs. HeteroCL [8] is one recent work that builds on the TVM framework [9]. HeteroCL is built as an API library of the Python Language, and its compiler is a runtime compiler. When HeteroCL code runs, it makes API calls to construct computation patterns and computing schedules from Python-syntax statements and generates synthesizable C code for Merlin HLS compiler [10]. For example, HeteroCL exposes APIs to perform code transformations such as loop pipelining, loop unrolling, quantization, etc. Although both HeteroCL and PyLog use Python syntax, their approaches are very different. HeteroCL is more like a Python library that is used to describe computation flow, instead of an implementation of a subset or extension of the Python language.

On the contrary, PyLog directly implements a subset of the Python language, PyLog code is compiled by ahead-of-time Python language compiler and the code that programmers write is what is the input to the compiler. PyLog's ahead-of-time compilation setting has several advantages. First of all, this enables maximum flexibility of the language, and makes it very easy to be extended. There is no constraints or limitation on the language design. Second, PyLog is designed to be compatible with standard Python grammar, and Python programmers can immediately start to code in PyLog. The existing Python code can be simply synthesized without much modification. Another difference between HeteroCL and PyLog is that, with HeteroCL, programmers still need to manually apply transformations and optimizations using HeteroCL APIs, while in PyLog, the compiler is responsible for implementation and optimization by default. Programmers can use pragmas to force the compiler to generate a specific implementation of the PyLog code, if they choose to do so.

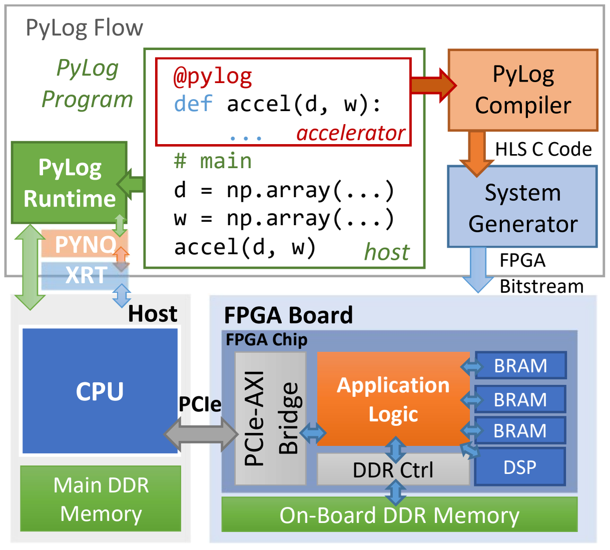
Dahlia [11] is another high-level programming language that compiles to HLS C code. Dahlia uses Scala-like syntax and it uses a type system to enforce design constraints in the HLS code so that unrolling and memory partitioning factors match. Similar to HLS flow, Dahlia requires users to specify design pragmas. PyLog doesn't require manual annotation of HLS pragmas. On the contrary, PyLog compiler automatically inserts pragma to optimize the design.

There are also several works that use Python and other high-level languages like Scala, and Haskell as hardware-description language (HDL) [12], [13], [14], [15]. The biggest difference between PyLog and these works is that PyLog flow is a high-level synthesis flow, and it does not require hardware knowledge to program in PyLog. These previous high-level HDLs elevate the syntax of the input languages, but users still need hardware expertise to use these HDLs.

3 PyLog Programming Model

3.1 Overview

PyLog presents a unified programming model for host and accelerator logic with consistent syntax and semantics.



```
@pylog
def accel(d, w):
    ...
# main
d = np.array(...)
w = np.array(...)
accel(d, w)
```

Fig. 2. The PyLog Flow and Example System Architecture.

This seamless host-accelerator programming model enables agile system design, convenient functional simulation, and flexible design space exploration.

Listing 1 shows a high-level example of PyLog program that describes both host and accelerator. This example contains two functions, `preprocess` and `compute`. Function `compute` is decorated with a Python decorator `@pylog`, therefore it is a PyLog kernel function and will be synthesized into a hardware accelerator on FPGA by PyLog. With `@pylog` decorator, programmers can easily specify accelerator function in the existing Python code.

As shown in the example, both host and accelerator are programmed with Python at the same abstraction level. The host and accelerator interact with each other seamlessly in a natural way. PyLog closes the gap between the abstract level of host programming and FPGA accelerator programming, and enables efficient system-level host-accelerator co-design.

```
def preprocess(data):
  ... # data pre-processing that runs on the host

@pylog
def compute(inputs): # top FPGA kernel function
    def do_work(data): # user defined function
        ...
    for d in inputs:
        do_work(d)
  ...

inputs = preprocess(data) # data pre-processing
result = compute(inputs)
# call FPGA (or run synthesis)
```

Listing 1. A High-Level PyLog Example.

Figure 2 illustrates the overall PyLog flow and the target FPGA system architecture. When PyLog user runs PyLog program with a standard Python interpreter, the `@pylog` decorator calls PyLog compiler to compile the decorated PyLog kernel function into HLS C code. Then, system generator syntesizes generated HLS C code and integrates all the system components to create a complete FPGA design. The generated FPGA bitstream is used to configure the resource and interconnects on FPGA. The rest of PyLog program is interpreted by the standard Python interpreter, and this part is the host code that runs on the host CPU.

TABLE 1
PyLog's Built-in Modes

Mode	Description
`cgen`	Generates optimized high-level synthesis C code
`hwgen`	`cgen` then run FPGA synthesis
`deploy`	Program and call FPGA then collect results
`pysim`	Simulate PyLog code with Python interpreter

When the decorated function is called, PyLog runtime is invoked to program FPGA, allocate and populate memory and invoke FPGA to accelerate the computation. The lower half of Figure 2 shows an example of PCIe-based FPGA platform. Note that PyLog can support both PCIe-based high-performance FPGAs and low-power SoCs and MPSoCs. CPU and FPGA interact through memory-mapped I/O ports as well as configuration registers on FPGA side.

PyLog requires the arguments of the PyLog functions to be NumPy arrays or NumPy scalars. From these NumPy object inputs, PyLog collects type and shape information of the arguments to the top function, which is the initial information for the type inference engine in PyLog.

To run FPGA synthesis or run FPGA accelerator, users simply run the whole program with standard Python interpreter. When the decorated Python function `compute` is called (line 9), *PyLog compiler is invoked to look for the synthesized design for the decorated function.* If the function has not been synthesized before, PyLog will compile the decorated `compute` function, generate `compute` FPGA IP, integrate IPs into a complete FPGA design, and finally synthesize FPGA hardware design and get FPGA bitstream and configuration files. If there exists a synthesized design for the decorated function for the target FPGA platform but the design has not been deployed in the target FPGA, then PyLog will program the FPGA, allocate memory, populate memory space with input arguments, and call FPGA accelerator and collect results. To run the FPGA accelerated programs, users do not need to write any extra FPGA specific code, and they will not notice any difference in the way of running the code on CPU or on FPGA. All the underlying CPU-FPGA interactions are taken care of by PyLog runtime. PyLog synthesis and execution share the same piece of code, which is also very similar to a regular Python code, except the decorator `@pylog`. Both synthesis flow and execution flow are fully automated.

PyLog users can also pass a "`mode`" string to `@pylog` decorator to configure PyLog mode. For example, `@pylog(mode='cgen')` runs only HLS C code generation flow. The list of possible PyLog modes is shown in Table 1.

PyLog uses Python syntax and it is friendly to both software and hardware developers. PyLog has a built-in type inference engine that can infer the types of objects in the program. PyLog supports basic Python operations and expressions as well as several high-level operators, which makes decription of computation flow intuitive, efficient, and natural. (Section 3.2). Besides, PyLog allows programmers to nest operators to express complicated computation patterns. Nested operators significantly simplify code and greatly increase the expressiveness of PyLog (Section 3.3). In addition to the expressive high-level PyLog operators, PyLog also supports data type customization and computation customization (Section 3.5). Besides, PyLog naturally allows users to simulate accelerator behavior in Python (Section 3.6). These capabilities make PyLog expressive and flexible enough to describe many different computation patterns across application domains. The rest of this section will describe all the features in detail.

3.2 High-Level Operators

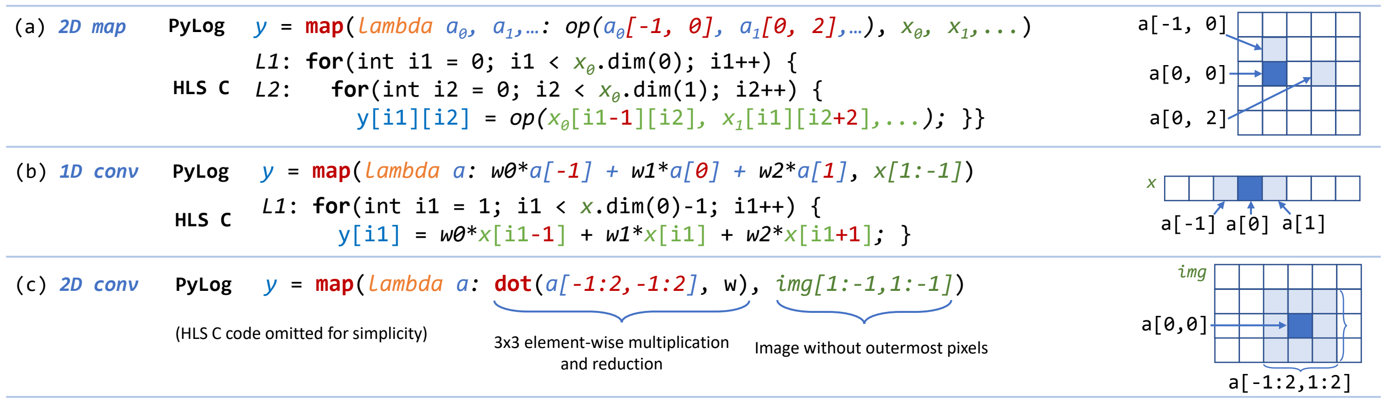
In addition to the frequently used standard Python keywords and built-in functions, PyLog provides a set of high-level operators that describe common computation patterns.

Array operators. PyLog supports a set of commonly used NumPy-style binary array operators. Listing 2 shows two examples of array operators. The first example (line 2) assumes that `a` and `b` are multi-dimensional arrays with the same shape. PyLog can infer the types and shapes of operands (`a` and `b`) as well as target (`c`), and further infer the binary operations "+" and "=" (assignment) to be array operators. PyLog generates the optimized parallel for loops to implement this array operation. PyLog also supports indexing and slicing that adds flexibility to expressions as in NumPy and native Python.

PyLog slicing style is the same as that in Python. Slice expression "`start:end:step`" means a sequence of indices with "`start`" as the first index, "`start+step`" as the second index, etc. The index continues until it reaches `end` (but not including `end`). Note that this is consistent with Python slice's left-inclusive and right-exclusive definition. Any of `start`, `end` or `step` can be left out. If `step` is left out, its value is understood to be 1. If `start` is left out, it means that the slice starts with 0. If `end` is missing, it means that the end value is the dimension. Using -1 for `end` means that the end is dimension-1. the PyLog compiler can infer the actual range of dimensions and indices according to the shape of the array under consideration. For example, `a[::2]` means a sub-array consisting of every other element in `a`.

Line 5 in the Listing 2 shows an example of using slicing and indexing to apply array operations to sub-arrays of the original arrays. In this example, elements in every other row from row 2 to row 64 of `z` (a total of 32 rows) receive the product of variable `x` and all elements in `y` that have index 6 in the second dimension and all the indices except the last one in the third dimension. The compatibility between the input and output arrays of this operation is checked by PyLog's type system. Also, the alignment between the input and output elements is automatically set by PyLog. For example, assume that `x` is a scalar variable. If `z` is a 64×8 array and `y` is a $32 \times 16 \times 9$ array, there will be $32 \times 8 = 256$ `z` elements and the same number of `y` elements involved. `z[2,0]` will receive `x * y[0,6,0]`, and `z[4,4]` will receive `x * y[1,6,4]`. In general, for all the 256 elements involved, `z[i,j]` will receive `x * y[m,6,n]` if $((i/2-1)\cdot 8)+j = m\cdot 8+n)$.

Note also that `x` can be a scalar or a vector, which will make the meaning of `*` different (vector linear scaling and vector element-wise multiplication, respectively). Again, PyLog will be able to infer the types and shapes of all the arrays with indexing and slicing, and it will also check whether the operations are valid by comparing the shapes



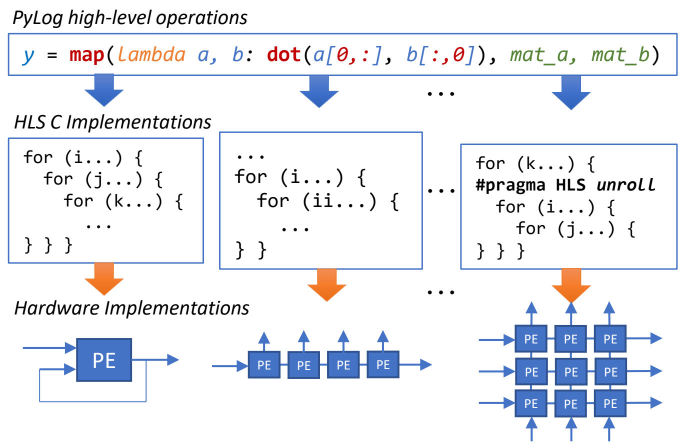
```
(a) 2D map    PyLog   y = map(lambda a0, a1,…: op(a0[-1, 0], a1[0, 2],…), x0, x1,...)
                      L1: for(int i1 = 0; i1 < x0.dim(0); i1++) {
              HLS C   L2:   for(int i2 = 0; i2 < x0.dim(1); i2++) {
                              y[i1][i2] = op(x0[i1-1][i2], x1[i1][i2+2],...); }}

(b) 1D conv   PyLog   y = map(lambda a: w0*a[-1] + w1*a[0] + w2*a[1], x[1:-1])
              HLS C   L1: for(int i1 = 1; i1 < x.dim(0)-1; i1++) {
                            y[i1] = w0*x[i1-1] + w1*x[i1] + w2*x[i1+1]; }

(c) 2D conv   PyLog   y = map(lambda a: dot(a[-1:2,-1:2], w), img[1:-1,1:-1])
              (HLS C code omitted for simplicity)
```

3x3 element-wise multiplication and reduction
Image without outermost pixels
a[-1, 0]
a[0, 0]
a[0, 2]
x
a[-1] a[0] a[1]
img
a[0,0]
a[-1:2,1:2]

Fig. 3. PyLog `map` operator examples. (a) 2D stencil, (b) 1D convolution, (c) 2D convolution.



```
y = map(lambda a, b: dot(a[0,:], b[:,0]), mat_a, mat_b)
```

```
for (i...) {
  for (j...) {
    for (k...) {
      ...
} } }
```

```
...
for (i...) {
  for (ii...) {
    ...
} }
```

```
for (k...) {
#pragma HLS unroll
  for (i...) {
    for (j...) {
} } }
```

Fig. 4. Different implementations generated from same PyLog code.

of sub-arrays. Array operators plus the slicing expressions support succinct and clear specification of linear algebra algorithms such as convolution and matrix multiplication.

```
# Array operation based on operators "+" and "="
c = a + b

# Array operation with slicing
z[2:66:2,:] = x * y[:,6,:-1]
```

Listing 2. Array operation examples.

```
1  # Vector add
2  out = map(lambda x, y: x + y, vec_a, vec_b)
3
4  # 1D convolution
5  out = map(lambda x:w0*x[-1]+w1*x[0]+w2*x[1], vec)
6
7  # Inner product
8  out_vec[i] = dot(matrix[i,:], in_vec)
9
10 # Square matrix multiplication
11 out = map(lambda x,y: dot(x[0,:],y[:,0]), ma, mb)
```

Listing 3. PyLog `map` and `dot` examples.

map operator. PyLog supports a built-in `map` operator, which is an extended version of the Python `map` function. Similar to the `map` function in Python, `map` operator in PyLog can be used as `map(f,`o_1`,...,`o_n`)` to repeatedly apply a function `f` to the n iterable objects o_1, o_2, ..., o_n where all objects must have the same shape. By default, the PyLog `map` operator behaves the same way as the Python `map` operator. In this case, f is defined with n formal parameters p_1, p_2, ... p_n, each of which refers to an element of the corresponding object that f is to be applied to in the map operator. For example, in Line 2 of Listing 3, `x` (p_1) refers to an element of `vec_a` (o_1) and `y` (p_2) refers to an element `vec_b` (o_2). In the i^{th} iteration of the `map` implementation, `f` takes the i^{th} element of `vec_a` and the i^{th} element of `vec_b`, add them together, and assign to sum to the i^{th} element of `out`. Typically, function `f` is a `lambda` function (anonymous function), as shown in Listing 3. For example, in Line 2 of Listing 3, the `lambda` function is an addition function whose output is produced by adding the values of two formal parameters together.

Beyond the basic features, PyLog `map` further supports an extension to allow function `f` to access any number of elements in the iterable objects by specifying an offset or a offset slice expression that specifies a collection of offsets with each reference in the function body. The offsets are defined based on the iteration index. For example, in Line 5 of Listing 3, in the i^{th} iteration of `map`, the `lambda` function accesses `vec[i-1]` (specified as (`x[-1]`), `vec[i]` (`x[0]`), and `vec[i+1]` (`x[1]`) in the function body. Thus the `lambda` in this example is a 1D convolution function with a 3-element filter of `w0`, `w1`, and `w2`. Figure 3 visualizes a few `map` examples. Note that this offset extension can naturally describe stencil operations. PyLog compiles stencil code described with `map` operator and connects to SODA [16] to generate highly efficient stencil accelerators.

dot operator. In PyLog, `dot` is defined as element-wise multiplication followed by a sum reduction. In other words, `dot(a, b)` is equivalent to `sum(a*b)`, where `a*b` is elementwise multiplication and the `sum` operator calculates the sum of all elements in the iterable object. For example, Line 8 of Listing 3 performs a dot product between row `i` of `matrix` and `in_vec` and assign the output value to the i^{th} element of `out_vect`. `dot` operator is introduced in PyLog to simplify programming and expose more optimization opportunities to the compiler. This operation is frequently used in many different applications, e.g. image filtering, matrix multiplication, stencils, etc.

Custom operators. PyLog allows users to define custom operators as PyLog functions, inside a PyLog decorated top function. Similar to other operators, PyLog can infer the types and shapes of operands and output of custom functions by propagating type information from input to output through the whole function. These user-defined functions can be reused and simplify programming. These custom functions will be synthesized into HLS C functions.

3.3 Offset-Slicing and Operator Chaining

The PyLog `map` offsets and offset slice expressions can be used for accessing higher dimensional formal parameter arrays. For each dimension, the offset can be a single number (e.g., 1, which means an input element in that dimension with offset 1 is accessed in an iteration), a slice (e.g., `-1:2`, which means three elements in that dimension with offsets -1, 0, and 1 are accessed in an iteration, or the entire dimension (`:`, which means all elements in that dimension are accessed in an iteration). For example, in Line 11 of Listing 3, the `map` operator will apply the `lambda` function to all positions of the 2D arrays involved. In iteration (`i, j`) of `map`, the offset expression `x[0,:]` means that the function accesses all elements of the i^{th} row of `mat_a` and `y[:,0]` means that the function accesses all elements of the j^{th} column of `mat_b`. That is, the map and lambda functions, chained together in Line 11, perform a dot product between the i^{th} row of `mat_a` (`x[0,:]`) and the j^{th} column of `mat_b` (`y[:, 0]`) and assigned the dot product value to `out[i,j]`, i.e., a matrix multiplication.

Note that the `map` offset slices for accessing formal parameter arrays should not be confused with the slicing of PyLog arrays. Assume that `x` is a formal parameter of a `lambda` and `vec` is a PyLog array, offset slice `x[-1:2]` specifies three accesses to the formal parameter with offsets -1, 0, 1 whereas `vec[1:-1]` produces a slice that consists of all elements of `vec` except the first and the last elements (i.e., all internal elements of `vec`).

The PyLog offset extension and operator chaining enables Python developers to intuitively and succinctly express common computation patterns in various application domains.For example, in Figure 3(c), `img` is the input to a 2D convolution, and `w` is 3×3 convolution filter. The slicing `[1:-1,1:-1]` applied to `img` extracts out the sub-array excluding the outermost elements at the edges. The `lambda` function is applied to each element in the extracted array, `img[1:-1,1:-1]`. The parameter to the lambda function is `a`, which corresponds to each element in the array. The offset-slicing expression applied to `a`, `a[-1:2,-1:2]`, expresses a sub-array consisting of neighbor elements around the current element `a` as well as `a`.

`dot` operator multiplies this square with convolution weight `w` (which is also a 3×3 square) element-wisely, and does a summation reduction to get the single convolution output at the current element `a`. This `dot` operation is repeatedly applied to each element position in `img[1:-1,1:-1]`, and this completes the whole 2D convolution. Note that PyLog can infer the shape of each object involved in the computation and users do not need to give any hints about object shapes. Details of PyLog type inference will be presented in Section 4.

```
1 # Dilated convolution where dilation = 2
2 map(lambda x:dot(x[-2:3:2,-2:3:2],w),img[2:-1,2:-1])
3
4 # Strided convolution where stride = 2
5 map(lambda x:dot(x[-1:2,-1:2],w),img[1:-1:2,1:-1:2])
```

Listing 4. Variants of 2D in PyLog.

In addition to basic 2D convolution, variants of more general convolution can also be expressed easily in PyLog, as shown in Listing 4. Dilated convolution with dilation equals to 2 can be described (line 2) by simply replacing `a[-1:2,-1:2]` with a[-2:3:**2**,-2:3:**2**] in the basic example above. Index "`-2:3:2`" means the sequence of indices of "{`-2, 0, 2`}", therefore a[-2:3:**2**,-2:3:**2**] represents a dilated convolution filter. Similarly, convolution with non-unit stride can also be described by specifying `step` in the index slices of `img` (line 5).

TABLE 2
Examples of NumPy Operators in HLS C

Operator	Tunable Performance Parameters
`argmax`, `argmin`	p_i: number of parallel inputs of comparison tree
`max`, `min`	p_i: number of parallel inputs of comparison tree
`matmul`	p_o: number of output processed in parallel
`convolve`	p_o: number of output processed in parallel p_m: parallel multiplications for each output
`sort`	q_o: partition factor of the output buffer

Note that these high-level operators only describe the arithmetic relationships between array objects without specifying any actual implementation details. For example, the `map` operator describes the repeated application of an operator to all the elements in an array, but it does not prescribe any ordering when iterating through these elements. In traditional HLS, the similar computation would be expressed with nested for loops, which actually implies an iteration order. The HLS quality heavily depends on how the computation is expressed, a key reason why it is hard to create optimal hardware design with HLS flow. In the PyLog flow, the actual order of iterating in `map` operation is left for the PyLog compiler to decide. This approach not only simplifies programming, but also enables better design optimization and code generation. Given the high-level operators, PyLog compiler gets full information about the computation pattern. It knows exactly where data dependencies are in the code, which allows it to perform aggressive loop transformation and other code optimization. How PyLog performs code optimization will be discussed in Section 4.

3.4 HLS C Library Integration

In addition to the high-level operators and Python modules described above, PyLog also supports integration of external HLS C library functions. This allows users to leverage the existing highly optimized HLS libraries. We develop an extensible library of HLS C operators that implement widely used NumPy functions. The interface of these operators is compatible with NumPy functions. PyLog users can call these HLS C operators in the same way as NumPy function calls. A few examples are shown in Table 2.

In spite of the similar interface, the specific implementation of our operator library is very different from NumPy. These PyLog external operators are developed in HLS C language and highly optimized for hardware. These operators are implemented as HLS C code templates and are highly parameterized and configurable, and can be configured by PyLog according to data types and shapes, as well as design goals. Similar to other PyLog operators and function calls, PyLog type inference engine will also do type inference and

TABLE 3
Computation Customization in PyLog

Operator	Customization	Description
`for`	unroll	Unroll for loop
	pipeline	Pipeline for loop
`map`	reorder	Interchange for loops
	tiling	Tile for loops
generic	pragma	Insert HLS pragma

type checking for these external operators, to figure out the configurations of these operators and ensure the arguments and return of these operators to be valid. Type inference and checking is done based on the inference rules customized for each of these operators. Taking operator `matmul(A,B,C)` that performs matrix multiplication `C=A*B` as an example, PyLog type engine checks if the shapes of `A`, `B`, and `C` has a pattern of (m,k), (k,n) and (m,n) or not. If yes, PyLog will configure the operator based on the inferred type and shape and instantiate the operator template and generate HLS C implementation. Otherwise, PyLog will stop and output error messages accordingly.

Besides functionality parameters, these external operators also have configurable performance parameters, which configure the implementation of the operator. PyLog tunes these parameters to balance the performance and resource utilization of the entire design based on the design goals. The performance parameters are listed in the second column of Table 2. Each operator can also be configured as pipelined or non-pipelined according to design needs.

3.5 Bitwidth and Compute Customization

In FPGA designs, integers and fixed-point data types are widely used to improve computation efficiency. PyLog allows users to specify integer and fixed-point data types with arbitrary precision. Listing 5 shows a few examples. The PyLog type system supports the propagation and compatibility checking of user-defined data types.

```
a = pl_int8(0)     # 8-bit integer
b = pl_uint512(0) # 512-bit unsigned integer
c = pl_fixed(8,3)(0.0) # 8-bit fixed-point number,
                       # 3 bits above decimal point
```

Listing 5. PyLog Arbitrary Precision Type Examples.

Aside from the internal optimization passes, PyLog also allows users who want to have more control to customize computation and memory in the code. Table 3 summarizes the computation customization types in PyLog. For loops can be customized with `unroll` or `pipeline`. Operator `map` can be customized with `reorder` or `tiling`, which will be applied to the for loops generated from `map` operation. Loop reordering and tiling are safe in `map` operation since there is no loop-carried dependence in `map` operation.

3.6 Functional Simulation Support

With the unified and seamless host-accelerator programming model provided by PyLog, programmers are not only able to program both host and accelerator efficiently, but also simulate the functionality of both host and accelerator easily. PyLog provides a `pysim` mode that allows the PyLog code to be interpreted by the standard Python interpreter, and all the PyLog specific operations and customizations will be removed or simulated. `pysim` can be used to simplify debugging and improve development efficiency.

To summarize, Table 4 compares the features of existing high-level FPGA design languages and flows with PyLog.

TABLE 4
High-Level FPGA Design Flow Comparison

Features	Dahlia [11]	HeteroCL [8]	PyLog
Hardware customization	✓	✓	✓
Data type customization	✗	✓	✓
Ahead-of-Time compilation	✓	✗	✓
Host programming	✗	✗	✓
System generation	✗	✗	✓
HLS C library integration	✗	✗	✓

4 Compilation and Synthesis Flow

PyLog flow is a fully automated FPGA programming and synthesis flow. It consists of three parts, PyLog compiler, PyLog system generator, and PyLog runtime.

PyLog compiler is a source-to-source compiler that translates PyLog source code to optimized HLS C code which can be synthesized by high-level synthesis (HLS) tools. The current supported HLS tool is Xilinx Vivado HLS [3] and Merlin compiler [10]. However, the analysis and optimization used in PyLog are not restricted to these HLS tools. The code generator of PyLog can be extended to support other HLS tools without much difficulty. The compilation steps of PyLog compiler can be categorized into four stages: (1) front-end analysis and PyLog intermediate representation (PLIR) generation, (2) type inference, (3) optimization, and (4) HLS C code generation.

PyLog system generator calls FPGA vendor's tools to synthesize generated HLS C code, integrates the FPGA application kernel with other system components, and generates FPGA configuration bitstream. PyLog runtime configures and invokes FPGA to accelerate computation in user's application.

4.1 Front-End Analysis and PLIR Generation

When a `@pylog` decorated function is called with NumPy arrays and scalars as parameters, PyLog compilation begins. In the first step, PyLog collects information about the data types and shapes of input arguments to the top function. This information is passed to PyLog sub-modules. The source code of the top function is parsed by Python built-in abstract syntax tree (AST) module `ast`, outputting the AST of PyLog code. AST is a low-level representation of the input code, which only represents the code syntax structure without semantics information. AST is the starting point for the following PyLog compilation steps. `ast` is the only module from standard Python interpreter that PyLog depends on. The following compilation steps do not depend on existing Python interpreter.

TABLE 5
PLIR Node Categories

Category	Example Node Types
Low-Level Ops	`PLUnaryOp`, `PLBinOp`, `PLAssign`, etc.
PyLog High-Level Ops	`PLMap`, `PLDot`, `PLPragma`, etc.
Code Objects	`PLConst`, `PLVariable`, `PLArray`, etc.
Code Structures	`PLFuncDef`, `PLCall`, `PLFor`, etc.

In the first stage, PyLog front-end traverses the PyLog AST, analyzes code structure, collects code information, and generates PyLog intermediate representation (PLIR).

PLIR nodes include nodes representing high-level computation patterns and code structures, e.g. `PLMap`, `PLDot`, etc., as well as nodes representing lower-level generic statements and operations, e.g. `PLFor`, `PLBinOp`, etc. Compared with the nodes in the PyLog AST, each node in PLIR is coarser in granularity, and the attributes of PLIR nodes carries more information. Each PLIR node has multiple attributes that either point to the other PLIR nodes or store the related information about this node. Essentially, the PLIR generation can be considered as a process where the PyLog front-end analyzer aggregates the sub-trees and structure information in the AST to form PLIR nodes.

Table 5 lists the representative categories of PLIR nodes and examples. Low-level operations and expressions are the nodes that represent basic arithmetic operations, indices, basic expressions, etc. PLIR high-level operations are the nodes that represent high-level primitive operations in PyLog, e.g. `map`, `dot`, etc. These are PyLog specific nodes. Code structures are the nodes that represent control flow and structure of the code, e.g., loops, branches, function definition, function calls, etc. The data fields of a PLIR node can point to another PLIR node; therefore, the whole PLIR is a tree that represents the code at high-level.

4.2 Type Inference and Type Checking

One of the biggest challenges in compiling Python code is that Python is a dynamically typed language and there is no explicit type declaration in the Python code, which makes it hard for compiler to understand the actual semantics of some operations. For example, consider a simple expression "`a + b`". Since `a` and `b` can be scalars or can be multi-dimensional arrays, this expression can mean scalar addition, or vector element-wise addition. The corresponding C code for this expression will be very different in these two cases. Without a context of types and shapes of `a` and `b`, it is not possible to know the actual meaning of this expression. To solve this problem, we implement a type inference engine in PyLog that infers the types and shapes of each object in the PyLog code. With PyLog type inference support, PyLog users do not need to provides explicit type hints in PyLog code.

In PyLog compiler, the type information of an object includes the type of data elements in the object as well as the number of dimensions of the object. PyLog compiler uses `PLType(ty, dim)` to denote types, where `ty` is the type of data elements and `dim` is the number of dimensions. For simplicity, we use notation T_t^d to represent `PLType(t, d)`. For example, the type of a three-dimensional array consisting of floating-point numbers can be represented as `PLType(float, 3)`, or T_{float}^3. Algorithm 1 shows the pseudo-code of the type inference algorithm in PyLog. Type inference starts with type and shape information of function arguments in the PyLog top function (which is carried by the input NumPy objects), and type information propagates across the whole PyLog code. Type inference is performed on PLIR and the result type and shape information is annotated to PLIR nodes. Type information propagation happens by performing type inference at each PLIR nodes, which is done according to the type inference rules. Table 6 list a few examples of type inference rules.

As an example, when inferring types of objects in a `map` operation, type inference engine first retrieves the type of operands $T_{t_1}^n$ from current context, which stores the types and shapes of visible variables at current point in the code. Then, the type engine is able to tell that the type of the argument to function `f` is $T_{t_1}^0$, because `map` operator iterates through each element in the operand and `f` takes

Algorithm 1 Type Inference and Checking

```
Input: Set of all variables $S$; set of input variables $S_{\text{in}} \subset S$; set of output variables $S_{\text{out}} \subset S$; Type mappings $T$ defined on $S_{\text{in}} \cup S_{\text{out}}$, PLIR with root $N_{\text{root}}$.
Output: Type mappings $T$ defined on $S$, or TYPEERROR.
 1: $S_{\text{typed}} \leftarrow S_{\text{in}} \cup S_{\text{out}}$
 2: for each node $n \in$ POSTORDERTRAVERSAL($N_{\text{root}}$) do
 3:     if $n \in S_{\text{typed}}$ then
 4:         $n_p \leftarrow$ PARENT($n$)
 5:         if $n_p \notin S_{\text{typed}}$ then
 6:             $T(n_p) \leftarrow$ TYPERULE($n \rightarrow n_p, T(n)$)
 7:             $S_{\text{typed}} \leftarrow S_{\text{typed}} \cup \{n_p\}$
 8:         else if $T(n_p) \neq$ TYPERULE($n \rightarrow n_p, T(n)$) then
 9:             return TYPEERROR
10:         for each $n_c \in$ CHILDREN($n$) do
11:             if $n_c \notin S_{\text{typed}}$ then
12:                 $T(n_c) \leftarrow$ TYPERULE($n \rightarrow n_c, T(n)$)
13:                 $S_{\text{typed}} \leftarrow S_{\text{typed}} \cup \{n_c\}$
14:             else if $T(n_c) \neq$ TYPERULE($n \rightarrow n_c, T(n)$) then
15:                 return TYPEERROR
16: if $S \subset S_{\text{typed}}$ then return $T$
17: else return TYPEERROR
```

TABLE 6
Type Inference Rules Examples

Operations	Types
`out = UnaryOp(a)`	`a` $: T_t^n,$ `out` $: T_t^n$
`out = BinOp(a, b)`	`a` $: T_{t_1}^n,$ `b` $: T_{t_2}^n,$ `out` $: T_{t_1}^n$
`out = dot(a, b)`	`a` $: T_{t_1}^n,$ `b` $: T_{t_2}^n,$ `out` $: T_{t_1}^0$
`out = map(f, a)`	`a` $: T_{t_1}^n,$ `f` $: T_{t_1}^0 \rightarrow T_{t_2}^m,$ `out` $: T_{t_2}^{m+n}$

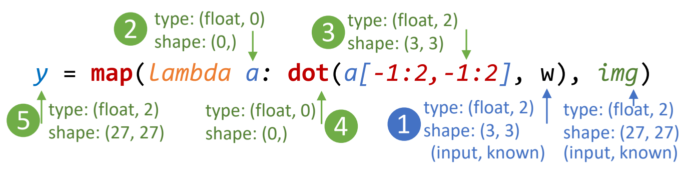


Fig. 5. An Example of Type Inference on Chained High-Level Operators.

```
code: {
  loop_i: {},
  loop_map_1_tile_1: {
    loop_map_0: {
      loop_map_1_tile_0 : {}
    }
  },
  loop_dot_0: {
    loop_dot_1: {
    }
  }
}
```

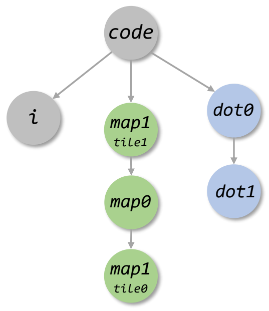


Fig. 6. An example `for` loop structure and its tree representation. `for` loops of different types are in different colors.

one element as input. Next, the type engine infers types of objects inside `f` and gets the return type of `f`, $T^m_{t_2}$. Finally, as a `map` operator, its return value is the aggregated results of `f`'s return values, so the type is $T^{m+n}_{t_2}$.

The shapes of objects are also inferred in the similar way, and happens at the same time as type inference. After type inference, the semantic of operations and expressions in the PyLog code is determined. When inconsistency is detected by the type system, a compilation error message is generated for the user. This makes debugging more user friendly than pure interpreter-based Python implementations. PLIR with type information is then ready for optimization and code generation.

4.3 Compiler Optimization

PLIR characterizes the high-level computation patterns in the input PyLog code, making it easier for PyLog optimizer to optimize the computation flow. Before the optimization stage, all the compiler analysis and transformation are independent of actual implementation. In this optimization stage, the compiler starts to consider and optimize the implementation for better performance. PyLog optimization consists of three steps, high-level operators lowering, loop transformation, and HLS pragma insertion.

High-level operator lowering. In this first step, PyLog optimizer traverses through PLIR, and replaces high-level operators in PLIR with functional equivalent groups of generic low-level operators. For example, vector operators and `map` operators will be replaced with a group of for loop nodes that represent nested for loops. The information about the type of original high-level operators (e.g. map, dot, etc.) are kept and annotated to the generated for nodes. This information is useful in the following optimization.

Loop transformation. For each high-level operator, PyLog is capable of generating multiple styles of nested for loops, including plain sequential version, loop reordering version, loop tiling version, and mixed optimization version where loop reordering and tiling are combined. The type of loop transformation used depends on the operator type and the available hardware resources. For example, loop reordering and loop tiling are safe and possible in `map` operation since there is no loop-carried dependence in `map` operation. Note that each of these versions also has one or more tunable parameters.

HLS pragma insertion. After PyLog optimizer replaces the high-level operators with nested for loops, the PyLog optimizer traverses the whole PLIR tree again and identifies the `for` loops in the code, then, it generates a loop structure tree that represents all the for loops in the code and their structure information. Each node in the tree is a `PLOptLoop` object that represents a for loop and its information. Each `PLOptLoop` object has actions of `unroll(n)` and `pipeline()`. Figure 6 shows an example of for loop structure and its tree representation. With this representation, the loop structure in the code becomes very clear. Then, PyLog optimizer starts to insert HLS pragmas according to the optimization algorithm. Algorithm 2 shows the pseudo-code for one of the optimization heuristics. In this algorithm, the algorithm traverses the loop structure in the post-order. That is, the optimizer starts with the leaf nodes in the loop structure, which corresponds to the innermost for loops in the code. Then it moves to the parents of the traversed nodes. For each node, determined by the type of for loop (whether it belongs to `map` nodes or `dot` nodes or regular for loops), the optimizer tries a set of candidate HLS pragmas (or actions). In each trial, it evaluates the area and latency changes after applying that pragma. If the benefits is higher than a threshold, it accepts the change, otherwise it discards the change. Then it continues and moves on to next action or next node. Note that right now we are using a basic heuristic to guide the optimization. Other more sophisticated optimization/search algorithms can be used and plugged into the PyLog optimizer and guide the optimization. We leave this as a part of future work.

Algorithm 2 HLS Pragma Insertion Algorithm

```
Input: Original loop structure $L$, improvement threshold
    $T$, total available area $A_{total}$.
Output: Loop structure $L$ with HLS pragmas configured.
 1: $latency, area \leftarrow$ EVALUATE($L$)
 2: for each loop $L \in$ POSTORDERTRAVERSAL($L$) do
 3:     if $L$ has attribute $unroll$ or $pipeline$ then
 4:         continue
 5:     else
 6:         if $L$ is from map then
 7:             $A \leftarrow \{unroll, pipeline, unroll.pipeline\}$
 8:         else $A \leftarrow \{pipeline\}$
 9:         for each $action \in A$ do
10:             $L.action()$
11:             $latency', area' \leftarrow$EVALUATE($L.action()$)
12:             if $\frac{latency-latency'}{area'-area} < T$ or $area' > A_{total}$ then
13:                 Undo $L.action()$
14:                 continue
15:             else break
```

Performance and resource models. We use the performance and resource models proposed in [17] to estimate the latency and resource utilization of the design points of each source code version. Based on these estimates, the compiler identifies the optimal design points. These optimal design points become the candidates of global design optimization. We use an example in Section 5.3 to further demonstrate the optimization mechanism.

4.4 Global Design Optimization

The optimizations presented in Section 4.3 focus on fine-grained operation-level and loop-level optimization, while in this section, we discuss system-level optimization in PyLog flow. Operation-level and loop-level optimizations identify the top implementation candidates for each of the high-level operators in PyLog. At the system-level optimization stage, PyLog optimizer finalizes the low-level design choices based on the global constraints and optimization targets. We formulate this global optimization problem as an integer linear programming (ILP) problem, solve the problem with an ILP solver, and finally get the optimal design choices.

At the operation-level optimization stage, for each high-level operator p_i, the optimizer identifies the set of top m design candidates for this operator $\{p_i^{(1)}, p_i^{(2)}, \ldots, p_i^{(m)}\}$, as well as the latency and resource estimates of these candidates. Let's denote the latency and resource estimates with mappings $L : p_i^{(j)} \to \mathbb{Z}^+$ and $A : p_i^{(j)} \to \mathbb{Z}^+$ respectively. Note that for each type of FPGA resource, there will be one estimate function. To simplify the notations, here we only write down the formula for one type of resource. The same formula applies to the other types of FPGA resources. The goal of this global optimization stage is to identify the optimal choice of design candidates for each of the high-level operators so that the overall latency of the whole design can be minimized, while the resource usage meets the FPGA resource constraints. This optimization problem can be formulated as ILP problem as follows.

We define a binary indicator variable $x_i^{(j)} \in \{0, 1\}$ to denote whether or not we choose the j^{th} candidate of the i^{th} operator, i.e. $p_i^{(j)}$. Since only one candidate will be chosen for a specific operator, we have the constraint $\sum_j x_i^{(j)} = 1$ for each operator p_i. Given available resource $A_{\max}$ on FPGA, the optimization problem is:

$$\min_{x_i^{(j)} \in \{0,1\}} \sum_{i,j} x_i^{(j)} L(p_i^{(j)}) \tag{1}$$

$$\text{subject to} \sum_{j} x_i^{(j)} = 1, \forall i, \tag{2}$$

$$\sum_{i,j} x_i^{(j)} A(p_i^{(j)}) \leq A_{\max} \tag{3}$$

Please note that the summation here takes the control flows in the program into account. For example, if a high-level operator is called inside a sequential `for` loop, all the dynamic instances of this operator will be summed up. This formulated ILP problem is then sovled by an external ILP solver, and the solution corresponds to the optimal choices of high-level operators design candidates.

4.5 C Code Generation and System Generation

After optimization, all the nodes in PLIR are low-level operators since the high-level operators have been replaced. PyLog code generator traverses through the optimized PLIR and generates C AST, which is then translated into actual C code.

PyLog system generator calls FPGA synthesis tools to synthesize generated HLS C code into FPGA IP block (Vivado HLS or Merlin compiler), and integrate the IP with all the other system components to create the complete FPGA design (Vitis or Merlin compiler). The whole system generation flow is fully automated.

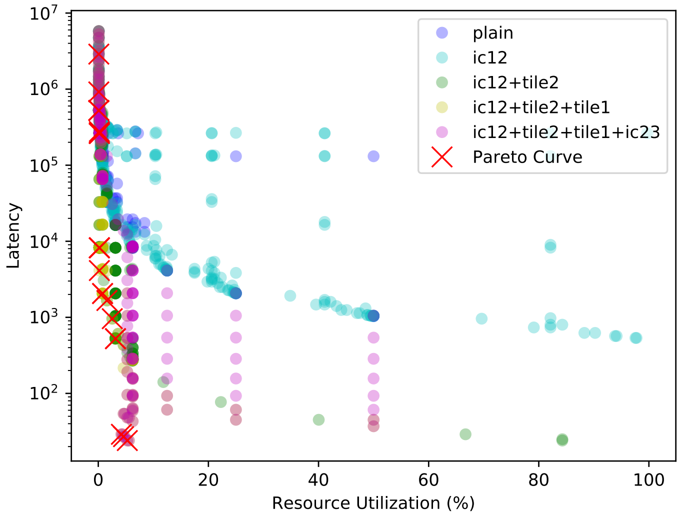


Fig. 7. Design space of various code versions of 2D array addition. ("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

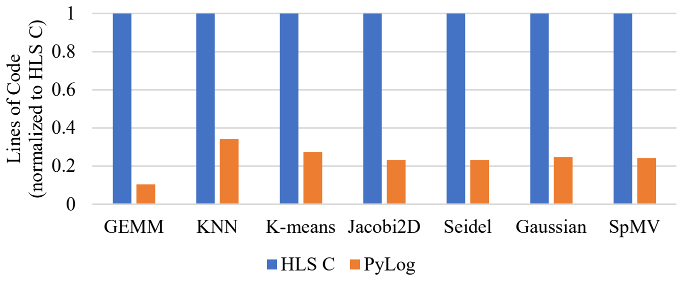


Fig. 8. Length of HLS C code and PyLog code.

4.6 PyLog Runtime

When the PyLog kernel function is called, PyLog automatically invokes FPGA to accelerate the program. First, it programs FPGA, then it allocates and populates arrays in CPU-FPGA shared memory space. Second, it invokes FPGA accelerator, and waits for FPGA to finish. Finally, PyLog collects computing results from FPGA and returns the results to the kernel function caller in the host program. This runtime is built on top of Xilinx PYNQ library [18], which supports both low-power platforms and high-performance platforms.

5 Evaluation

This section evaluates PyLog flow in several different aspects, namely portability, language expressiveness, and performance.

5.1 Portability

PyLog flow is designed to be generic enough to be portable across different FPGA platforms. The whole PyLog flow, including code generation, hardware generation, and PyLog runtime, can be used with a wide range of FPGA platforms. Table 8 lists the FPGA platforms that are currently supported by PyLog flow. When targeting

TABLE 7
Accelerator Performance Evaluation on AWS F1 Instance

Benchmark	LUT	Registers	BRAM	DSP	f (MHz)	P (W)	T_{CPU}	T_{HCL} [8]	T_{PyLog}	$\frac{T_{\text{CPU}}}{T_{\text{PyLog}}}$	$\frac{T_{\text{HCL}}}{T_{\text{PyLog}}}$
KNN	109276	74889	425	0	256.40	37.222	0.48	0.45	0.26	1.85	1.73
K-means	10829	17604	3	7	273.97	37.429	38.16	4.24	4.45	8.58	0.95
Jacobi-2D [19]	93925	111144	96	704	269.03	37.327	11.31	8.25	5.19	2.18	1.59
Seidel [19]	47304	57854	30	304	269.03	37.341	21.37	8.22	5.16	4.14	1.59
Gaussian Filter [19]	56580	75846	48	688	147.15	37.783	23.63	7.34	5.19	4.55	1.41
GEMM	12868	63759	655	1024	250.00	39.657	60.34	8.13	13.05	4.62	0.62
SpMV	8294	12787	25	21	273.97	37.225	0.29	-	0.24	1.21	-
Histogram [20]	4096	7647	13	0	273.97	37.327	5.85	-	2.07	2.83	-
									Geometric Mean	**3.17**	**1.24**

T_{CPU}: Execution time on single-thread CPU; T_{HCL}: Execution time on HeteroCL [8] generated accelerator; T_{PyLog}: Execution time on PyLog generated accelerator; All time values are in milliseconds (ms); '-' means the implementation is not publicly available.

TABLE 8
Current Supported FPGA Platforms in PyLog

Platform Type	FPGA Platforms
Low Power	ZedBoard [21], PYNQ [22], Ultra96 [23]
High Performance	Amazon EC2 F1 instance [24], Alveo series (U200, U250, U280) [25]

a specific FPGA platform, one simply passes the platform name to the `@pylog` decorator, and no other code change is needed. For example, `@pylog(mode='deploy', board='aws_f1')` will execute the PyLog code using Amazon AWS F1 instance FPGAs. Exactly the same PyLog code with `@pylog(mode='deploy', board='pynq')` applied instead will run the program with PYNQ FPGA, assuming FPGA bitstreams have been generated with `mode='hwgen'`.

5.2 Expressiveness

To evaluate the expressiveness of PyLog, we compare the number of lines of code between HLS C code and PyLog code. To make comparison fair, for PyLog versions, we only use PyLog built-in high-level operators to express the benchmarks but not using other pre-built functions or libraries. The HLS C versions are HLS C code generated by PyLog from the corresponding PyLog version. This guarantees the FPGA designs of two versions are equal. Figure 8 shows the results. With PyLog, on average only 30% length of code is needed to express computation, compared to the Vivado HLS flow.

5.3 Design Space Exploration and Search

We evaluate the effectiveness of our compiler optimizations by profiling the latency and resource utilization of real-world workloads. Fig. 7 shows all the valid design points of a 2D array addition example, as well as the optimal design points. These design points are identified by PyLog compiler automatically. First of all, PyLog identifies the four valid versions of implementation, that is, "ic12", "ic12+tile2", "ic12+tile2+tile1", and "ic12+tile2+tile1+ic23". Here "ic12" corresponds to the version that interchanges loop 1 and loop2, while "tile2" is the version that tiles loop 2. Second, PyLog explores all the valid design points for each of these valid code versions. These design points correspond to different ways of inserting HLS pragmas. Fig. 7 uses circles with different colors to mark the design points from different code versions. The optimal design points on the pareto curve are also marked in the figure. As we can see from Fig. 7, the optimal design points come from different code versions. Our compiler is able to identify the optimal design points from all the code versions. These optimal design points becomes the design candidates for the global design optimization, which is then solved by ILP solver.

5.4 Accelerator Performance

We evaluate PyLog performance with real-world benchmarks on Amazon EC2 F1 f1.2xlarge instance [24]. Amazon EC2 F1 f1.2xlarge instance is a cloud computing platform with 8-core Intel Xeon E5-2686 v4 CPU and a Xilinx Virtex UltraScale+ XCVU9P FPGA. The benchmarks are from different domains and have varied computation patterns, including linear algebra, data analytics, stencil, sparse operations, etc. Table 7 shows the evaluation results. The table lists FPGA resource utilization (look-up tables, registers, BRAMs and DSPs), design frequency (f (MHz)), design power (P (W)), and kernel execution time (T (ms)) of PyLog generated designs. T_{CPU} is the execution time on AWS F1 CPU with one single thread. The CPU baselines are optimized CPU implementations from [8] and other sources. T_{HCL} is the execution time on HeteroCL [8] generated accelerators. The HeteroCL accelerators are generated from optimized HeteroCL implementations that are available online. T_{PyLog} is the execution time on PyLog generated accelerators. SpMV and histogram benchmarks do not have HeteroCL implementations available yet so we do not compare with HeteroCL versions for these two benchmark. The stencil benchmarks (Jacobi-2D, Seidel, and Guassian Filter) are compiled to generate Vivado HLS C code with IPs from external HLS library SODA [16]. The other benchmarks are compiled to Merlin C code. In terms of compilation time, PyLog HLS C code generation only takes seconds and therefore PyLog compilation time is negligible compared to Vitis synthesis that takes hours.

The last two columns in Table 7 show the speedup achieved by PyLog accelerator over CPU implementation and HeteroCL implementation respectively. On average, PyLog accelerators achieve around $3.17\times$ and $1.24\times$ speedup over CPU baseline and HeteroCL accelerators. PyLog can generate accelerators with better or similar performance compared with HeteroCL in most of benchmarks. Note that we uses PyLog generic backend to compile GEMM benchmark while HeteroCL uses special systolic array backend for GEMM. This is the main reason for the performance gap in GEMM benchmark. After we add support for systolic array backends this gap will be filled. This is left as future work. The main sources of speedup achieved by PyLog are as follows. First, the high-level operators expose parallel processing opportunities and the compiler is able to insert HLS pragma with better insight. Second, PyLog compiles Python code directly and has native support for imperative programming. This enables users to have fine-grained control in hardware generation. Third, PyLog incorporates external highly optimized HLS libraries and it tunes the design parameters of these libraries to achieve good balance of performance and resource utilization.

6 Conclusion

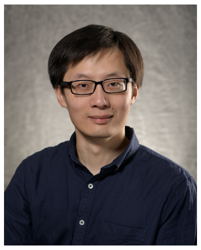
In order to improve FPGA development efficiency and simplify FPGA programming, we built PyLog, an algorithm-centric Python-based FPGA programming and synthesis flow. PyLog flow compiles Python functions into optimized HLS C code, and generates a complete system including FPGA accelerator as well as host-side runtime environment. The built-in PyLog high-level operators and PyLog optimizer automate design implementation and optimization, which reduces the burden of FPGA developers. Evaluation results show that PyLog is expressive to describe different types of applications with few lines of code and it can accelerates high-level software applications effectively and achieve significant speedup.

Acknowledgments

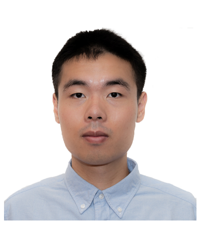
The authors would like to acknowledge the support from Laboratory for Physical Sciences, and the Xilinx Center of Excellence and Xilinx Adaptive Compute Clusters (XACC) program at the University of Illinois Urbana-Champaign.

References

[1] J. Dean, D. Patterson, and C. Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, Mar 2018.
[2] arXiv. arXiv.org e-Print archive. https://arxiv.org/.
[3] Xilinx. Vivado High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[4] Intel. Intel High-Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.
[5] Xilinx. Xilinx SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.htm.
[6] Intel. Intel FPGA SDK for OpenCL Software Technology. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html.
[7] Xilinx. CHaiDNN: HLS based deep neural network accelerator library for Xilinx UltraScale+ MPSoCs. https://github.com/Xilinx/CHaiDNN.
[8] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'19, pages 242–251, New York, NY, USA, 2019. Association for Computing Machinery.
[9] TVM. TVM stack. https://tvm.apache.org/.
[10] Falcon Computing. Merlin compiler. https://www. falconcomputing. com/ merlin-fpga-compiler/.
[11] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types, 2020.
[12] Chisel. Chisel/FIRRTL hardware compiler framework. https://www.chisel-lang.org/.
[13] Clash. Clash: A modern, functional, hardware description language. https://clash-lang.org/.
[14] PyMTL3. PyMTL3 (Mamba), an open-source python-based hardware generation, simulation, and verification framework. https://github.com/pymtl/pymtl3.
[15] PyRTL. PyRTL. https://ucsbarchlab.github.io/PyRTL/.
[16] Y. Chi, J. Cong, P. Wei, and P. Zhou. Soda: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.
[17] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Performance modeling and directives optimization for high level synthesis on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.
[18] Pynq. http://www.pynq.io/.
[19] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, 2012.
[20] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. Parallel programming for fpgas, 2018.
[21] Zedboard. http://zedboard.org/product/zedboard.
[22] PYNQ-Z1: Python productivity for Zynq-7000 ARM/FPGA SoC. https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/.
[23] Ultra96 board. https://www.96boards.org/product/ultra96/.
[24] Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/.
[25] Xilinx Alveo boards. https://www.xilinx.com/products/boards-and-kits/alveo.html.



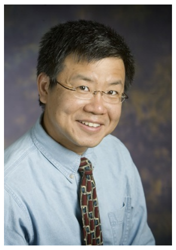
Sitao Huang received his B.Eng. degree in Electrial Engineering from Tsinghua University, Beijing, China in 2014, and M.S. degree in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign in 2017. His current research interests include compilers and high-level synthesis flows for hardware systems, hardware accelerators, and parallel computing architectures.



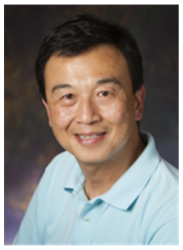
Kun Wu received his B.Eng. degree from Tsinghua University, Beijing, China in 2019. He is currently working toward the Ph.D. degree in the Electrical and Computer Engineering Department at the University of Illinois at Urbana-Champaign. His current research interest includes parallel computer architecture, compilers and lower-level software systems.

Hyunmin Jeong received his B.S. degree in Computer Engineering from University of Illinois at Urbana-Champaign, Illinois, United States, in 2019, where he is currently working towards the Ph.D. degree in Electrical and Computer Engineering. His current research interest includes hardware accelerators for neural networks and high level synthesis.

Chengyue Wang is expected to received the B.S. degree in electrical engineering both from Zhejiang University, Zhejiang, China, in 2021, and from University of Illinois at Urbana-Champaign, illinois, United States, in 2021. Her current research interests include reconfigurable computing, hardware design automation, and hardware AI accelerator design.



Deming Chen (Fellow, IEEE) received the B.S. degree in Computer Science from the University of Pittsburgh in 1995, and the M.S. and Ph.D. degrees in Computer Science from the University of California at Los Angeles in 2001 and 2005, respectively. He is the Abel Bliss Endowed Professor of Engineering in the ECE Department of the Grainger College of Engineering at UIUC. His research interests include system-level and high-level synthesis, machine learning, computational genomics, GPU and reconfigurable computing, and hardware security. He has received numerous research and service related awards, including nine best paper awards. He is an IEEE Fellow, an ACM Distinguished Speaker, and the Editor-in-Chief of ACM Transactions on Reconfigurable Technology and Systems (TRETS).



Wen-mei Hwu (Fellow, IEEE) joined NVIDIA in February 2020 as senior distinguished research scientist, after spending 32 years with the University of Illinois at Urbana-Champaign, where he was a professor, Sanders-AMD endowed chair, Acting Department head and chief scientist of the Parallel Computing Institute. He and his Illinois team developed the superblock compiler scheduling and optimization framework that has been adopted by virtually all modern vendor and open-source compilers today. For his research contributions, he received the ACM SigArch Maurice Wilkes Award, the ACM Grace Murray Hopper Award, the IEEE Computer Society Charles Babbage Award, the ISCA Influential Paper Award, the MICRO Test-of-Time Award, the IEEE Computer Society B. R. Rau Award, the CGO Test-of-Time Award and the Distinguished Alumni Award in CS of the University of California, Berkeley. He has also won numerous best paper awards for major conferences. He is a fellow of IEEE and ACM.