Hot & Spicy: Improving Productivity with Python and HLS for FPGAs

Sam Skalicky, Joshua Monson, Andrew Schmidt, Matthew French
Information Sciences Institute, University of Southern California, Arlington, VA
{skalicky,jmonson,aschmidt,mfrench}@isi.edu

Abstract—We present Hot & Spicy an open-source infrastructure and tool suite for integrating FPGA accelerators in Python applications, provided entirely as Python source code and available at https://spicy.isi.edu. This suite of tools eases the packaging, integration, and binding of accelerators and their C/C++ based drivers callable from a Python application. The Hot & Spicy tools can: (1) translate Python functions to HLS-suitable C functions, (2) generate Python C wrapper bindings, (3) automate the FPGA EDA tool flow, and (4) retarget Python source code to use accelerated libraries. For FPGA experts, this enables increased productivity and supports research on each stage of the flow by providing a framework to integrate additional compilers and optimizations. For everyone else this enables fast, consistent, acceleration of applications on FPGAs. We describe the design principles and flows for supporting high-level Python abstractions in an FPGA development flow. Then we evaluate the overheads of calling C/C++ routines from Python. Lastly, we show the results of accelerating a kernel in a Python image processing application and achieve a 39,137x speedup over the original Python implementation, and 6x speedup over an high-performance, hand-optimized OpenCV library implementation.

I. INTRODUCTION

Since the introduction of the Virtex II Pro with PowerPC processors 15 years ago [19], FPGAs have evolved into more sophisticated heterogeneous processing devices with full ARM-based processor subsystems, hardened floating-point units and memory controllers. The FPGA community has been down this road before when first incorporating embedded processors into FPGA devices. However, this time FPGAs with ARM processors are capable of booting full Linux systems without first requiring the FPGA logic to be programmed. This enables developers to first run software applications on an FPGA, such as in C or more recently Python, without writing any HDL. Moreover, tools and frameworks now more reliably exist to provide high-level synthesis (HLS) support to aid in creation of accelerators from higher-level languages [3][4]. As a result, these devices are attracting the attention of non-traditional FPGA developers: software developers. These HLS tools still rely on C/C++ which is suitable for conventional embedded platforms, but do not yet support more modern programming languages.

In this work, we consider a use case that is increasing in interest and popularity, Python on FPGAs. Since 2014 Python has grown in popularity from #4 on IEEE Spectrum’s Programming Language Survey to #1 in 2017 [7]. In addition, in 2017 Xilinx released PYNQ [20] as an open-source project to provide Python productivity on Zynq devices. While PYNQ has been a catalyst for this work, one limitation that became apparent is the lack of Python HLS support. Specifically, if an application exists in Python someone must manually profile and identify suitable functions for acceleration then use either conventional FPGA development tool flows or convert the application to C/C++ and use modern HLS tools to perform FPGA implementation, as illustrated in the top two approaches in Figure 1.

Instead, this work simplifies the translation process by leveraging the decades of research in HLS [3][4] and more recent open-source projects for Python development on Zynq (PYNQ) by building higher-level productivity and run-time tools for Python applications. Our approach, as shown in the bottom of Figure 1, provides an end-to-end development flow that automates the acceleration of existing Python applications on FPGAs. Ultimately, we are asking the simple question: “Can we support accelerating portions of a pure Python application in FPGAs?” This would not only include support for new FPGA users, but also for advanced users. The development flow will need to extend from Python to the bitstream for acceleration, but also integrating the bitstream, drivers, and runtime support to use the accelerator from Python.

We have developed custom tools to quickly and easily integrate FPGA-based accelerators into Python applications
for internal use, which are releasing to the research community as an open-source project. One of our goals in this is to increase adoption of FPGAs for acceleration and improve results for non-experts. Another goal is to provide concepts and design principles to support higher level abstractions in an FPGA development flow enabling researchers to focus on their contributions rather than infrastructure development.

Our design flow is made up of four independent tools that together form a comprehensive design flow. The Synthesis of Python-to-C (sPyC, pronounced spicy) tool supports source-to-source translation for functions from Python to HLS-suitable C/C++. The Python Linker (Pylon) tool generates wrapper bindings that link the Python application to C/C++ accelerator drivers. The Python rewriter (Pyrite) tool refactors the original source code to use the accelerators via the generated wrappers. The Pyramid tool produces scripts to drive the EDA tool flow.

Our work is architected such that these tools work together seamlessly to support a simple flow that can accelerate functions from a Python application. They also support advanced use cases such as integrating an existing accelerator from C/C++ via HLS or custom RTL into a Python application. This enables users to get something working quickly and easily integrate existing EDA flows.

The contributions of this work are:
1) Open-source Python to HLS-suitable C translator
2) Open-source tools to support integrating FPGA-based accelerators into Python applications
3) Analysis of overheads of using Python to control FPGA-based accelerators compared to C/C++
4) Experimental results showing realizable speedup for Python implemented algorithm accelerated via source-to-source translation to C and implemented via HLS

This rest of this paper is organized as follows. Section II discusses the relevant related work. Section III presents an overview of the flow and describes the tools. Section IV presents the experiments and results. Section V summarizes our contributions.

II. RELATED WORK

In this work, we present a suite of tools to integrate support for FPGAs in Python. We support the following operations: translating Python to C, automating the EDA flow, and generating Python C API wrapper bindings. There are many existing tools for translating Python code to C, however none of them produce HLS-suitable C code. Cython [2] translates Python code to C but still makes calls into the CPython interpreter and standard libraries. ShedSkin [8] is experimental and only translates a restricted subset of Python to C++ and does not support the NumPy [17] package. Numba [5] compiles Python to machine code directly rather than producing C/C++ code using its own LLVM compiler. Pythran [10] also translates Python to C++ but uses other high-performance libraries (OpenMP, boost, STL) in the background for speed. Lastly, PyPy [14] has a JIT compiler that translates Python into C code but still runs in an interpreted fashion and cannot generate complete C files.

There are also tools for generating Python C API wrapper bindings, the most widely used is SWIG [1] which generates STL arrays and vectors for Python lists and arrays. Current HLS tools, such as Vivado HLS used in this work, do not support STL types on function interfaces.

Although there are a few examples of HLS support for Python, they are not suitable for translating pure Python to HDL. The Polyphony tool [11] requires writing a stylized hardware suitable Python syntax. Another Python compiler [22] was designed to synthesizes to a custom multi-core processor architecture and execute custom microinstructions rather than traditional hardware logic. The Three Fingered Jack tool [16] transforms a restricted set of Python loop nests to HDL but only supports a 32-bit data width. None of these tools support user control of interface types (AXI, FIFO, BRAM, etc.) making it difficult to integrate the generated HDL into complete hardware systems. MyHDL allows hardware description in the Python language [6] but is just another HDL (like VHDL or Verilog) and is not suitable for migrating a regular Python application to hardware. High-level synthesis (HLS) tools bridge a very complex gap from high-level languages to low level hardware description. Implementing a quality HLS tool that produces production grade HDL is not an easy task. And supporting the complexities of the entire language is a long process. The Vivado HLS tool is one example of the best currently available and yet, it is only capable of synthesizing a subset of the C/C++ language to high quality HDL. In this paper, we refer to this subset of the C/C++ language as “HLS-suitable C”. Taking advantage of these capabilities is the motivation for the sPyC translator tool.

Currently there are no FPGA-based design tools that support applications written in the Python language. SysPy [13] is a tool that generates scripts to simplify hardware development using Python as the scripting language. It does not add any support for Python-based applications. The PYNQ [20] project provides a regular Linux experience on FPGA-based SoCs in addition to libraries and APIs that are available in Python. The PYNQ overlay methodology presents a design pattern for using FPGA accelerators in Python. But there are no tools that automate any of the manual and error prone steps necessary to design an overlay or integrate it into with an application. The Software-defined System-on-chip (SDSoC) Development Environment [12] provides tools and automation to ease the design of heterogeneous hardware/software systems by automating the hardware system creation and driver interfacing of accelerators to C/C++ applications, but there is no Python language frontend. There is currently no integration between PYNQ and
SDSoC. In this paper, we integrate PYNQ and SDSoC using our suite of tools.

III. THE HOT & SPICY APPROACH

The Hot & Spicy tool suite is designed to support accelerating portions of a Python application in FPGAs. In any application, there are certain functions that represent the critical path or bottleneck of the application, and accelerating these functions provides the most benefit to the overall application performance. Given an application written entirely in Python, our tools support integrating accelerators with Python applications.

A. Supported Flows

Our tools support various design flows for integrating accelerators into a Python application. These accelerators can be implemented as Python functions, C/C++ functions, or HDL IPs. Figure 2 shows an high-level view of the tools used in various flows.

The Python-based flow operates by translating a single Python function to C so that an high-level synthesis (HLS) tool can generate an accelerator IP. Then, scripts to drive the vendor tool implementation flow are produced, the design is implemented, and a bitstream generated. Next, the Python C API wrapper is generated to call the accelerator driver function in C, which is then used to produce a Python module that can be imported into a user’s application. Finally, the original application is rewritten to use the accelerator by importing the new module.

The C/HDL-based flow operates very similarly to the Python-based flow except that no Python code is translated to C. Instead, users provide existing C/C++ code destined for synthesis using an HLS tool or custom HDL code. Then the vendor tools perform implementation and generate the bitstream. Our tools still generate the Python C API wrapper, and a Python module is produced. In this flow, the sPyC translator is not used.

B. sPyC Translator

Just as HLS tools increased productivity for HDL developers by transforming C implementation to HDL, our Python to C source-to-source translator, sPyC, improves productivity for hardware developers looking to accelerate functions written in Python with HLS. sPyC generates synthesizable C/C++ code targeting the Vivado HLS tool.

The goal when implementing sPyC was to translate pure Python syntax with the guiding principle being any sPyC-specific syntax requirement must not modify the original behavior of the Python code. Python provides a built-in AST package that gives access to the same routines used by the Python interpreter to parse source code. We use this module as our front-end to extract the abstract syntax tree (AST) of the program so that we can guarantee that we are not adding any custom syntax.

<table>
<thead>
<tr>
<th>Accelerators</th>
<th>Hot &amp; Spicy Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL</td>
<td>C</td>
</tr>
<tr>
<td>Py</td>
<td>sPyC</td>
</tr>
<tr>
<td>Pyramid</td>
<td>C</td>
</tr>
<tr>
<td>Pylon</td>
<td>C</td>
</tr>
<tr>
<td>Pyrite</td>
<td>HLS</td>
</tr>
<tr>
<td>bit</td>
<td>S</td>
</tr>
<tr>
<td>Py App</td>
<td>Mod</td>
</tr>
<tr>
<td>Acc App</td>
<td>Drvr</td>
</tr>
<tr>
<td>script</td>
<td>EDA</td>
</tr>
</tbody>
</table>

Figure 2: Implementation flows using the Hot & Spicy tools.

In addition to the regular basic Python syntax, sPyC requires that users implement type annotations on function arguments and return [18]. There is some type inference support for primitive variables, but users can also define specific types using variable annotations [9]. These annotations are supported in Python versions 3.6 and up. Listing 1 shows an example matrix multiplication function implemented in the traditional triple for loop style and Listing 2 shows the same code with annotations to specify types (lines 1-3, 9) and docstrings for pragmas (lines 4-5, 8). sPyC supports a subset of Python syntax inside the function that is being translated. However, this does not restrict the rest of the user’s application; all Python syntax is supported in the rest of the source file.

The supported Python syntax within the accelerated function still includes all the necessary structures to support complex computation. It supports arithmetic operators like +, -, x, ÷, %, conditionals like if, elif, else, both for and while loops, and the built-in range() function. Variables must be primitive types like char, short, int, long, float, double or NumPy ndarrays [17]. Additionally, the shape attribute on ndarrays is also supported so that it can be used in expressions. NumPy arithmetic functions like round, power, arctan, and sqrt are supported by calling the appropriate math.h function. Calling custom user defined functions in the function body is supported by also translating these sub-functions. There are no fundamental technical limitations for only supporting what is described above. Rather, these are the currently supported features and development is still on-going. Section III-F describes future work goals.

sPyC can be called from the command line like:

```
$ spyce my_app.py -func foo
```

This will call the translator to process the file my_app.py and translate the function foo from Python to C. Listing 3 shows the result of translating the matrix multiplication code in Listing 2 to HLS-suitable C code. Notice that the dimensionality of the arrays on the function arguments (lines 1-3 in Listing 2) has been transformed into a single
C. Pyramid Implementation Flow

Once a Python function has been translated into C, it can be implemented in HDL using an HLS tool. The Pyramid tool generates scripts for SDSoC to drive HLS and generate a complete system design including the drivers for transferring data and controlling the accelerator. In this flow, there is no main function therefore we use the SDSoC library flow to produce a static library. After accelerating a C function through HLS, SDSoC rewrites the source code to call a new driver function that interacts with the accelerator, rather than the original C function.

SDSoC provides some important features necessary for this work: contiguous memory management, high-performance data movers and drivers, and full system generation. Memory allocators and data mover drivers are provided as a static library that is normally integrated into the final software binary during linking. Within this library are memory management routines and data structures that manage the memory reserved via Linux CMA. In the dynamic PYNQ environment, users need to be able to allocate memory without using SDSoC and thus PYNQ provides a dynamic version of this regularly static library. In order to integrate SDSoC generated accelerators with this dynamic library, our flow diverges from the regular SDSoC flow such that after a regular SDSoC build completes, we must regenerate the binary without including SDSoC’s static library. Then, when the Python module is loaded the regular Linux shared library framework will load the dynamic library.

Pyramid can be called from the command line like:

```
$ pyramind -func foo my_app.py -source my_acc.cpp
```

Notice that users still specify the Python application source code and accelerated function. This is used in the invocation of SDSoC to specify the top-level accelerator function whose implementation is in the file specified by the -source <file.cpp> option. The primary output of Pyramid is a script that drives the SDSoC implementation flow. This is the same process that happens when an Eclipse-based IDE generates a makefile using CDT. Users manually execute the generated script after Pyramid completes. In the Python-based flow, the source and header files are generated by sPyC. In the C/HDL-based flow, users specify their own custom source files.

D. Pylon Wrapper Generator

The Python interpreter and run-time are written in C++ and therefore can be extended to execute custom user programs by linking against the run-time. However, using the Python C API requires writing another wrapper to interface custom C code with Python’s internal class representation. Pylon generates this wrapper file automatically by analyzing the function arguments and return type annotations.

Pylon can be called from the command line like:

```
$ pylon -func foo my_app.py
```

Pylon generates C code to interface the accelerator driver function with Python data types and classes. It also produces a script to execute the compilation step, producing a shared object. We use this capability to link in the custom driver code produced by SDSoC so that an accelerator can be used from Python. This shared object is loaded by the Python interpreter when importing the module containing the accelerator driver function. When the function is executed in Python, it executes this wrapper which sets up input/output
data pointers to the physical memory inside of the Python data structures and then calls the SDSoC-produced driver function to use the hardware accelerator.

E. Pyrite Rewriter

Pyrite refactors the original Python code to use hardware accelerators. Given a Python module containing a function that interacts with an accelerator, users must modify their application code to import this module and call the function from the module at the appropriate location. Additionally, they must also take care to choose appropriate allocators to initialize their arrays. The standard NumPy ndarray allocator creates arrays in virtual memory. This requires a Scatter-Gather DMA to transfer the data. PYNQ provides access to the SDSoC contiguous allocator which is compatible with the Simple DMA and Zero-copy data movers. In Section IV-C we experimentally evaluate the trade-offs of using virtual versus contiguous memory with different DMAs.

Pyrite can be called from the command line like:

```
$ pyrite -func foo my_app.py
```

Pyrite refactors the user’s original Python code to import the hardware module and replaces calls to the original Python function with calls to the driver function in the new module. It can also replace calls to ndarray constructors with calls to an helper function that uses the contiguous allocator. This is an optional feature and is experimental.

F. Current Capabilities & Future Work

We have developed the Hot & Spicy tool suite from scratch to bridge the gap in supporting applications written in the high-level Python language using FPGA accelerators. As such we have focused on the most necessary support and features initially. In this section, we describe future extensions in 3 areas: Python-to-C translation, application-level static analyses, and application to additional domains outside of the embedded space.

The sPyC translator currently supports a limited set of Python syntax, however this can be extended with more development effort. Vivado HLS only supports a restricted set of C/C++ for synthesis, which also restricts the capabilities of sPyC. Currently there is support for parsing a single Python file and accelerating a single function. In order to target real-world codebases, we will need to add support for multiple files, imported module tracking, and accelerating multiple functions. Xilinx has recently released the xfOpenCV library [21] which provides HLS-compatible C code for many OpenCV functions. Integrating these into Python with our tools would require adding support for OpenCV’s cv::Mat data structure in the generation of Python C API wrappers. Currently wrappers can only be generated for primitive arrays.

To generate high-performance systems, users will want to take advantage of SDSoC’s ability to directly connect multiple accelerators together in a dataflow style to reduce the amount of data movement between the processor and the fabric. This will require application-level static analyses to perform dependency tracking to determine which functions can be directly connected. This is more of an advanced feature, but one that cannot be enabled without the strong foundation of tools presented in this work. Additionally, other analyses can be added to automatically insert the necessary pragmas for configuring interfaces. Similarly, our tools can support HLS pragma insertion for automatically accelerating functions.

Our work is highly dependent on system design tools from Xilinx, namely SDSoC in the embedded space targeting SoC-based FPGAs. There is not currently anything as mature as SDSoC with a C/C++ front end available from any other source (vendor or academic). As new tools emerge, our work can be applied to those as well. Another application of our tools is in the support of Python with the OpenCL-based vendor tools and PCIe platforms that enable higher performance on larger devices and access to cloud-based applications where Python is being used more heavily than in embedded systems.

IV. EXPERIMENTS & RESULTS

While developing the Hot & Spicy tools, we have created a suite of unit tests to validate each tool individually as well as tests for integration of the various tools into the Python-based and C/HDL-based flows. We present 4 experiments that evaluate the performance of the tools, and the overheads that are incurred. First, we evaluate sPyC by analyzing the time it takes to translate various functions. Then we evaluate the overhead using a dynamic library versus the default SDSoC static library. Next, we evaluate the overhead to call an accelerator from Python versus calling it from a regular C program. Finally, we present the acceleration of a pure Python Canny edge detection application using existing source code from widely available open source resources.

We evaluated the applications by running through the implementation flow using the Hot & Spicy tool suite and the 2017.2 version of the Xilinx SDSoC tool (which includes 2017.2 versions of Vivado, HLS, and SDK). We targeted the PYNQ board which has a Zynq-7000 7020 dual ARM Cortex A-9 processor with integrated FPGA fabric and a processor clock of 650MHz. We ran the accelerators at

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Lines of Python</th>
<th>Code C</th>
<th>Translate time [seconds]</th>
<th>HLS time [seconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>3</td>
<td>7</td>
<td>0.066</td>
<td>56.47</td>
</tr>
<tr>
<td>For</td>
<td>4</td>
<td>9</td>
<td>0.073</td>
<td>55.17</td>
</tr>
<tr>
<td>IfElifElse</td>
<td>7</td>
<td>14</td>
<td>0.064</td>
<td>53.09</td>
</tr>
<tr>
<td>MMult</td>
<td>61</td>
<td>31</td>
<td>0.070</td>
<td>67.67</td>
</tr>
<tr>
<td>Canny</td>
<td>250</td>
<td>107</td>
<td>0.078</td>
<td>249.15</td>
</tr>
<tr>
<td>Huge1k</td>
<td>1,000</td>
<td>1,503</td>
<td>0.111</td>
<td>57.90</td>
</tr>
<tr>
<td>Huge10k</td>
<td>10,000</td>
<td>15,003</td>
<td>0.554</td>
<td>151.16</td>
</tr>
</tbody>
</table>
100MHz for simplicity even though many of the cores can achieve higher performance. We leave system customization to max out the performance to future work. We used a modified version of the PYNO 2.0 Ubuntu image with an updated SDSoC library from version 2016.1 to version 2017.2 to be compatible with the latest tools.

A. Evaluation of sPyC Translations

To evaluate the performance of sPyC to translate Python to C, we used a series of functions to see how capable our tool is to support real-world functions. We translated an empty function (with no body), a simple for loop, a set of conditional statements, a matrix multiplication function with triple for loops, the Canny edge detection code, and artificial functions with function bodies containing from 1,000 to 10,000 lines of code. Table I shows the results of translating these functions to C using sPyC on a PC with a 2.2GHz Intel Xeon E5-2630 and 512GB of memory (note that only up to 16GB of memory was ever used). For each test, we recorded the number of lines of code in the entire source file, not just the function that was translated, since the whole file is parsed by the tool. We also recorded the number of lines of C code in the output file and the time necessary for both: sPyC translating Python-to-C, and Vivado HLS synthesizing C-to-HDL. You can see that sPyC runs much faster than Vivado HLS and can still handle large real-world functions.

B. Evaluation of a Centralized Library

At runtime, Python buffers are allocated on-the-fly and will be accessed from SDSoC’s DMA drivers. This required making the SDSoC library a dynamic shared object rather than a static library. The caveat of doing this is a potential additional overhead of each call into the library. We evaluated this overhead by executing various C/C++ programs and collected the time it took to call into the library.

In this experiment, we used a 32x32 floating-point matrix multiplication application with the same source code between each experiment. We varied pragmas to direct SDSoC to generate systems with different data movers, coherency, and memory management in order to exercise many of the library functions for completeness. We collected the time it took to call into the library and setup the DMA to transfer data. For each test, we compiled two executables: one statically linked, and another that was linked against the dynamic shared library. Each test executes the accelerator 1024 times and we averaged the runtimes for each function.

Table II shows characteristics of each test and the additional overhead of the dynamic library.

Notice that all tests require more time to use the dynamic library to setup the DMA versus the static library. Overall, these overheads are very reasonable for the benefit of using a dynamic library since the overhead is not dependent on the amount of data being transferred. Also notice that the amount of time it takes to setup the DMA to transfer data varies wildly depending on which DMA is used, how memory is allocated, and whether the DMA is connected to a coherent port. Users will need to keep this in mind when determining which pragma to use in their code.

C. Evaluation of Python Overheads

Calling C/C++ routines from Python incurs additional overhead compared to calling the same routines from a C/C++ application. This experiment used the C/HDL-based flow, where we modified custom C accelerator code with different pragmas, but the Python code remained unchanged. We analyzed these overheads for the same matrix multiplication applications described previously, but on a much courser level. In this experiment, we capture the time it takes to call the same accelerator function as a whole. This includes setting up the DMAs, starting the accelerator, and waiting for it to complete using the dynamic library. Table III shows the amount of time it took to call an accelerator from Python verses C/C++. These tests use the exact same bitstream and drivers analyzed in the previous experiment with the dynamic library.

In each of these tests the same accelerator is used, but each one has a different data transfer configuration. Using a Scatter-Gather DMA (mmult_sg) versus a Simple DMA (mmult) results in 1.8x longer execution of the function. Not using coherency (mmult_hp) results in a 2.1x longer execution.

Table III: Overhead of calling an accelerator from Python for different DMA scenarios and their relative performance
ecution and using non-contiguous memory (*mmultMalloc*) results in a 18.3x longer execution. Yet, for each of these tests the overhead of calling the function from Python averages only 42.3 microseconds. This is because data is not copied from the Python domain to the C/C++ domain, instead the same memory is accessed and only addresses are passed.

These results showed large performance benefits to transferring contiguous memory over virtual memory. Another important factor is the performance of the allocation. The time to allocate a 4KB buffer in Python using a non-contiguous allocator takes approximately 32 microseconds compared to 283 microseconds to allocate a contiguous buffer. Even with the additional latency, in many cases higher performance is achieved using contiguous memory by carefully managing allocations and reusing buffers.

**D. Canny Edge Detection**

For this experiment, we sought to accelerate a complex real-world application in Python with the goal to use an existing implementation in the style of Python rather than C/C++ rewritten in Python. We chose a Canny edge detection application that was available open source, designed and written in Python [15] in order to evaluate our tools. First, we took the unmodified source code and tweaked it to use syntax supported by sPyC. This required modifying 11 lines of code out of 279 with annotations necessary for typing as described in Section III-B. Then we attempted to implement the function as-is via HLS but the placement of the design in the small 7020 device failed because the algorithm as-written was using too many copies of the current frame, which resulted in too many BRAMs being used. In order to get the design to fit, we combined the various loops together to operate on line buffers rather than whole frames. Finally, we added directives to improve performance.

Table IV shows the performance results of running the Canny edge detection algorithm on the PYNQ board for a 320x240 sized image. The *Original Python* is the unmodified open source implementation and the *Refactored Python* is the version modified to meet the sPyC syntax and refactored to fit on the device, but executed in software. The *Unoptimized HLS* uses the refactored code implemented in hardware with no performance pragmas (pipeline, unroll, etc.). The *Pipelined HLS* version has some loops pipelined, and the *Partitioned HLS* has both pipelining and some arrays partitioned. We also implemented a version of the application that used the OpenCV library function in software for comparison.

These results show that modifying source code to be perform- in HLS is normally counter-intuitive to improving software performance as the refactored code resulted in a significant slowdown running in Python. Yet implementing that refactored code in hardware gave a nice 820x speedup over the original Python implementation. After adding some HLS pragmas (pipeline, array_partition, etc.) without modifying the algorithm, we were able to achieve a 39,137x speedup which was also 6x faster than the OpenCV version. These results show that it is possible to, from Python, manipulate an algorithm to achieve performance in an FPGA using our tools.

Table V shows the post-place and route resource results of implementing the various systems. The *Original Python* failed to implement due to too many BRAMs unable to be placed on the device (151.5 out of 140 available) and as such these results are post-synthesis only. After refactoring the code, the *Unoptimized HLS* system was able to be implemented successfully. Notice that the *Pipelined HLS* example uses less BRAMs than the *Unoptimized HLS*. This is because some arrays were implemented in flip-flops instead of BRAMs and as a result the pipelined design used 5,415 more flip-flops. The final *Partitioned HLS* design used more BRAMs as arrays were partitioned and more LUTs and flip-flops were used to generate muxes to interface all of these memories. In addition, more DSPs were able to be fed thanks to the memory partitioning.

**V. Conclusion**

We presented Hot & Spicy, a tool suite for integrating FPGA accelerators in Python applications. We evaluated the capabilities of the tools and showed the overheads of accessing accelerators from Python to be minimal. In this paper, we evaluated the tools by directly accelerating Python code for a Canny edge detection application and achieved a 39,137x speedup over the initial software design and a 6x speedup compared to an high-performance, hand-tuned OpenCV implementation. In the future, we will evaluate integrating existing C/C++ HLS accelerators such as the xfOpenCV library and existing RTL into Python applications.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Performance</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Python</td>
<td>48.14 sec</td>
<td>1.0x</td>
</tr>
<tr>
<td>Refactored Python</td>
<td>139.28 sec</td>
<td>0.3x</td>
</tr>
<tr>
<td>Unoptimized HLS</td>
<td>35.68 ms</td>
<td>820.0x</td>
</tr>
<tr>
<td>Pipelined HLS</td>
<td>12.22 ms</td>
<td>3,939.0x</td>
</tr>
<tr>
<td>Partitioned HLS</td>
<td>1.23 ms</td>
<td>39,137.0x</td>
</tr>
<tr>
<td>OpenCV</td>
<td>7.19 ms</td>
<td>6,695.0x</td>
</tr>
</tbody>
</table>

Table V: Canny edge detection resource results
The tools are open source and available online at https://spicy.isi.edu with all of the examples and tests mentioned in this work.

ACKNOWLEDGMENT
This work was supported by the National Aeronautics and Space Administration (NASA) under grant 80NSSC17K0286. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of NASA or the U.S. Government.

The authors would like to thank the Xilinx University Program for their donation of boards and licenses. Thanks to both the PYNQ and SDSoC teams at Xilinx for their support.

REFERENCES