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@isi.edu
Hot & Spicy

• Hot

IEEE Spectrum 2017 Programming language Ranking

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>🌍💻</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>📈📱</td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>🌍📱</td>
<td>99.4</td>
</tr>
<tr>
<td>4. C++</td>
<td>📈📱</td>
<td>97.2</td>
</tr>
</tbody>
</table>

• Spicy

synthesize Python to C/C++ (SPyC)
Motivation: SpaceCubeX

- NASA sponsored effort spanning the past 2+ years to evaluate next-gen space-based compute platforms
- Earth scientists using Python to prototype algorithms
- Need to quickly implement Python apps on FPGAs

### MidAR Algorithm
- High framerate image buffering and processing
- Signal reconstruction and filtering
- Spectral analysis and calibration
- Optical communications

### Implementation
- Python
- OpenCV

Accessible, rapid prototyping of next generation satellite and multi-satellite constellations capabilities via virtual machines deployed in a cloud computing environment.
Problem Statement

• Have Python application, need acceleration
  – Accelerate Python code (custom algorithm)
  – Utilize existing C/HDL based accelerators

• System design is complex
  – DMAs, buffers, interfacing components
  – Controlling, drivers, APIs

• EDA Tools are low-level HDL-based

• HLS tools are C language-based

• Vendor offerings
  – Python Productivity on Zynq (PYNQ)
  – Hardware
  – Libraries & APIs
  – Methodologies
Our Approach: Hot & Spicy

- Raise level of abstraction to target Python
  - source-to-source translation from Python to C
  - automate the system integration
  - generate Python/C API wrappers
  - refactor Python app to use accelerators
Related Work - Translation

[Cython][Cpython][ShedSkin][Numba][Pythran][PyPy]

• **Convert Python code to C/C++**
  – Bloated – includes interpreter or other runtime elements, type checking
  – Incompatible with HLS – have dependencies on other libraries (ie. STL, boost, etc.)

• **C/C++ code destined to run on PC**
  – Not HLS-suitable
**sPyC: synthesizing Python to C/C++**

- **Source-to-source translation**
  - Abstract Syntax Tree (AST) level translation
  - C/C++ Code Gen - custom syntax generator from Python AST
  - Operates at the function level
    - Input is a Python function, output is a C/C++ function

- **Type support**
  - Basic type inference implemented (via propagation)
  - Can manually specify variable types
  - Must manually specify function argument types

- **Goal: Use pure Python syntax**
  - No special syntax that would require be incompatible with Python AST
  - Additional requirements should not change original code behavior

- **Supports a subset of Python syntax**
  - Loops, conditionals, arithmetic, Numpy arrays, lists
Pyramid: System Integration

- **Pyramid** tool generates scripts to drive SDSoc flow
- SDSoc generates comprehensive system architecture
  - Hardware system physically integrates components
  - Software drivers are generated to interface with accelerators
Pylon: Python/C API Extensions

• Given that Python is implemented in C++
• Python has APIs for extension modules
  – Natively integrated (dynamic library loading)
  – Requires C/C++ wrapper to interface

• *Pylon* (*Python linker*) tool generates wrapper(binding)
1. Translate Python function to C/C++
2. Generate EDA script
3. Generate bitstream & drivers
4. Generate wrapper & compile Python module
5. Retarget App
Experiments: Overview

- Raises the level of abstraction to use accelerators
- Quickly/Easily integrate FPGA-based accelerators into Python applications
- Hot & Spicy tools analyses
  - sPyC source-to-source translation
  - Static vs Dynamic driver library
  - Calling accelerator from Python vs C
- Example Application
  - Canny edge detection

Python Productivity on Zynq (PYNQ)
Experiments: sPyC translation

- Translated various functions from Python to C
- Compared time to translate (via sPyC) versus time to synthesize (via HLS)

<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>53.69</td>
</tr>
<tr>
<td>MMult</td>
<td>61</td>
<td>31</td>
<td>0.070</td>
<td>67.07</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>

Translation does not add significant overhead to implementation time
Experiments: Static vs Dynamic Library

- Goal: compare overhead of using dynamic library vs static library
- Evaluated various combinations of DMA, coherency, and memory contiguity (virtual)

<table>
<thead>
<tr>
<th>Test Name</th>
<th>DMA Type</th>
<th>Coherency</th>
<th>Contiguity</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult</td>
<td>Simple</td>
<td>Coherent</td>
<td>Contiguous</td>
<td>Standard</td>
</tr>
<tr>
<td>mmult_sg</td>
<td>Scatter-Gather</td>
<td>Coherent</td>
<td>Contiguous</td>
<td>Standard</td>
</tr>
<tr>
<td>mmult_hp</td>
<td>Simple</td>
<td>Non-Coherent</td>
<td>Contiguous</td>
<td>Cache Flush</td>
</tr>
<tr>
<td>mmult_sg_hp</td>
<td>Scatter-Gather</td>
<td>Non-Coherent</td>
<td>Contiguous</td>
<td>Cache Flush</td>
</tr>
<tr>
<td>mmult_malloc</td>
<td>Scatter-Gather</td>
<td>Coherent</td>
<td>Non-Contiguous</td>
<td>Page Pinning</td>
</tr>
<tr>
<td>mmult_hp_malloc</td>
<td>Scatter-Gather</td>
<td>Non-Coherent</td>
<td>Non-Contiguous</td>
<td>Page Pin+Flush</td>
</tr>
</tbody>
</table>

- Results show ~1.18us average overhead for dynamic library

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult</td>
<td>1.70</td>
<td>1.91</td>
<td>0.21</td>
</tr>
<tr>
<td>mmult_sg</td>
<td>8.06</td>
<td>8.40</td>
<td>0.34</td>
</tr>
<tr>
<td>mmult_hp</td>
<td>18.01</td>
<td>19.20</td>
<td>1.19</td>
</tr>
<tr>
<td>mmult_sg_hp</td>
<td>22.47</td>
<td>23.20</td>
<td>0.73</td>
</tr>
<tr>
<td>mmult_malloc</td>
<td>198.16</td>
<td>201.77</td>
<td>3.61</td>
</tr>
<tr>
<td>mmult_hp_malloc</td>
<td>217.83</td>
<td>218.86</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Using dynamic library does not significantly degrade performance
Experiments: Call Accel. from Python vs C

- Same accelerator configurations from previous experiment
- Same drivers & bitstream between Python & C
- Both using dynamic library
- Average overhead 42.3us of using Python

<table>
<thead>
<tr>
<th>Test Name</th>
<th>C  [μs]</th>
<th>Python [μs]</th>
<th>Overhead [μs]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmult</td>
<td>33.48</td>
<td>56.00</td>
<td>22.52</td>
</tr>
<tr>
<td>mmult_sg</td>
<td>62.41</td>
<td>104.00</td>
<td>41.59</td>
</tr>
<tr>
<td>mmult_hp</td>
<td>71.41</td>
<td>107.00</td>
<td>35.59</td>
</tr>
<tr>
<td>mmult_sg_hp</td>
<td>104.97</td>
<td>152.00</td>
<td>47.03</td>
</tr>
<tr>
<td>mmult_malloc</td>
<td>613.74</td>
<td>673.00</td>
<td>59.26</td>
</tr>
<tr>
<td>mmult_hp_malloc</td>
<td>661.07</td>
<td>709.00</td>
<td>47.93</td>
</tr>
</tbody>
</table>

Reasonable overhead for benefit of running application in Python
Experiments: Canny edge detection

- Algorithm from publicly available open source code
- Pure Python implementation

**Software versions**
- Original version (Python)
- Refactored version (Python)
- OpenCV version (C++)

**Hardware (HLS) versions**
- Original version
- Refactored version
- Pipelined
- Partitioned

### Performance Comparison

<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>58.68 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>

Able to optimize algorithm from Python and achieve a speedup
Experiments: Canny resources

• **Post synthesis**
  – Original Python – used too many BRAMs (*did not fit)

• **Post PAR**
  – Unoptimized – by default all arrays were impl. in BRAMs
  – Pipelined – convolution was pipelined, arrays impl. in FFs
  – Partitioned – line buffer BRAMs partitioned

<table>
<thead>
<tr>
<th>Test Name</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>36kBRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[53,200]</td>
<td>[106,400]</td>
<td>[220]</td>
<td>[140]</td>
</tr>
<tr>
<td>Original Python*</td>
<td>23,246</td>
<td>31,076</td>
<td>108</td>
<td>151.5</td>
</tr>
<tr>
<td>Unoptimized HLS</td>
<td>11,722</td>
<td>15,596</td>
<td>20</td>
<td>27.5</td>
</tr>
<tr>
<td>Pipelined HLS</td>
<td>17,061</td>
<td>21,011</td>
<td>20</td>
<td>26.5</td>
</tr>
<tr>
<td>Partitioned HLS</td>
<td>18,250</td>
<td>22,356</td>
<td>30</td>
<td>28.5</td>
</tr>
</tbody>
</table>
Conclusion

• **Hot & Spicy flow is a cohesive suite of tools, end-to-end support**
  – `sPyC` validated the source-to-source translation approach

• **Showed feasibility of using Python in an embedded system**
  – 42.3us overhead to call an accelerator from Python vs C

• **Accelerated Canny by modifying algorithm in Python**
  – Achieved Speedup of 39,137x over original Python software

• **Tools, source code, applications are available online**
  – Licensed as GPL v3
  – spicy.isi.edu
What’s next?

• **sPyC translation**
  – Extending supported syntax
  – Arbitrary precision annotation
  – Support multiple Python source files, import module tracking
  – Better automatic type inference
  – Automatic interface pragma generation

• **Pylon wrapper/binding generation**
  – Support OpenCV/xfOpenCV datatypes cv::Mat/xf::Mat
  – Arbitrary precision ndarrays (ap_int)

• **Application level analyses**
  – Translate multiple functions
  – Function level dependency/dataflow analysis to support accelerator pipelining

• **HPC platforms**
  – SDAccel, SDK OCL

• **More collaboration**
Questions

Sam Skalicky
skalicky@isi.edu

http://spicy.isi.edu

Come see our demo at 6:30