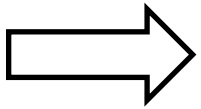
PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, and Christopher Batten

Cornell University
Computer Systems Laboratory

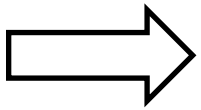
Outline

The Computer Architecture Research Methodology Gap



PyMTL

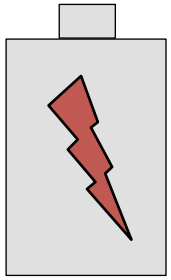
The Performance-Productivity Gap



SimJIT

Trends in Computing Systems

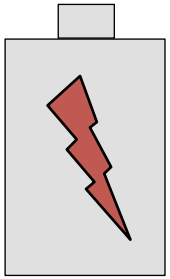
Energy & Power Constrained



Credible Energy and Power Analysis

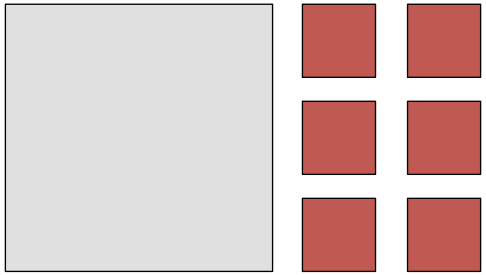
Trends in Computing Systems

Energy & Power Constrained



Credible Energy and Power Analysis

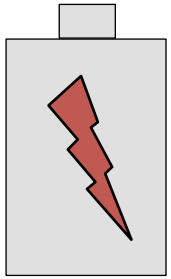
Extensive Specialization



Productive Design Space Exploration of Specialized Units

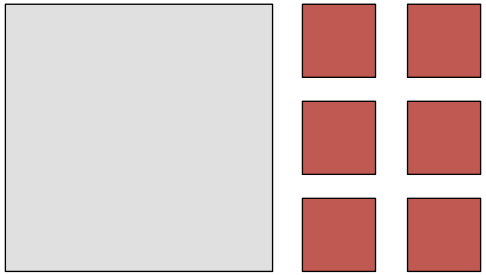
Trends in Computing Systems

Energy & Power Constrained



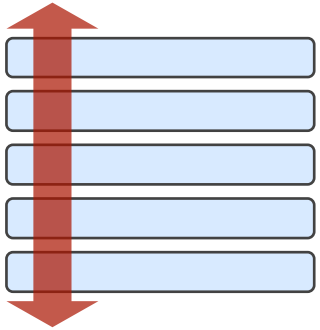
Credible Energy and Power Analysis

Extensive Specialization



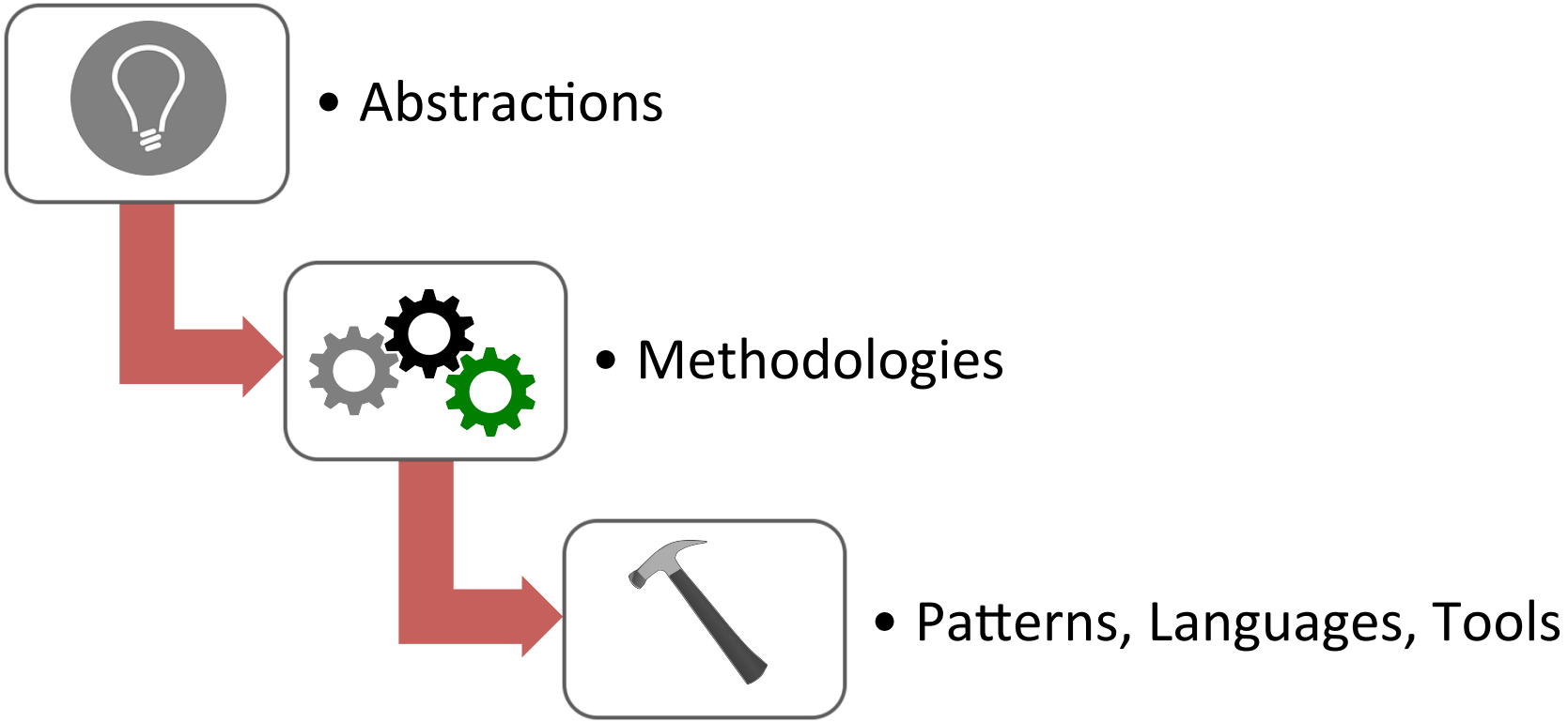
Productive Design Space Exploration of Specialized Units

Cross-Layer Optimization



Effective Strategies for Vertically Integrated Design

Managing Increasing Design Complexity



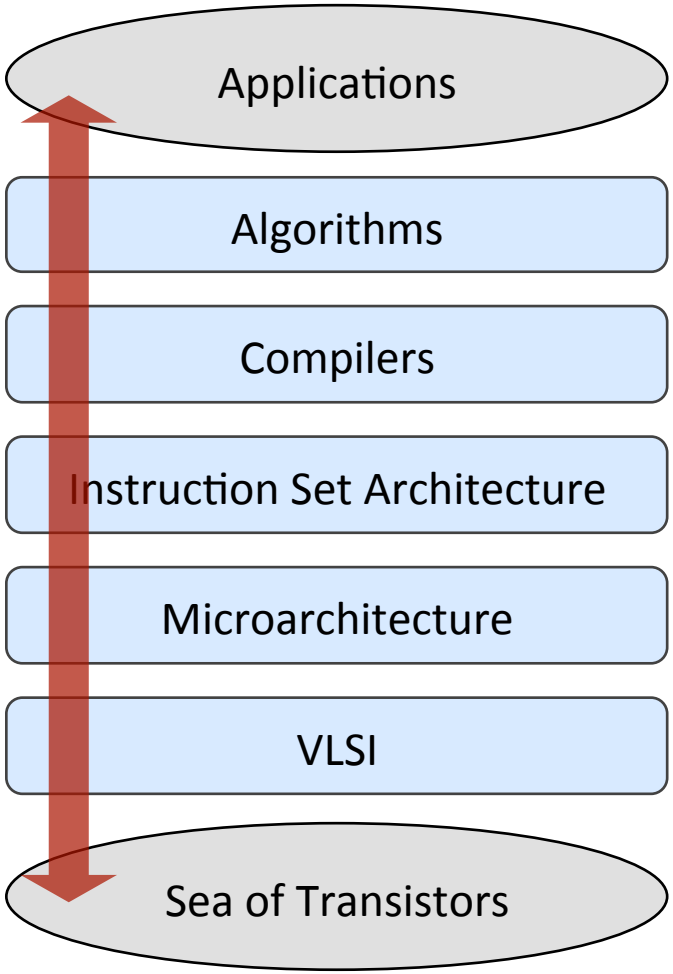
Computer Architecture Research Abstractions

- Abstractions

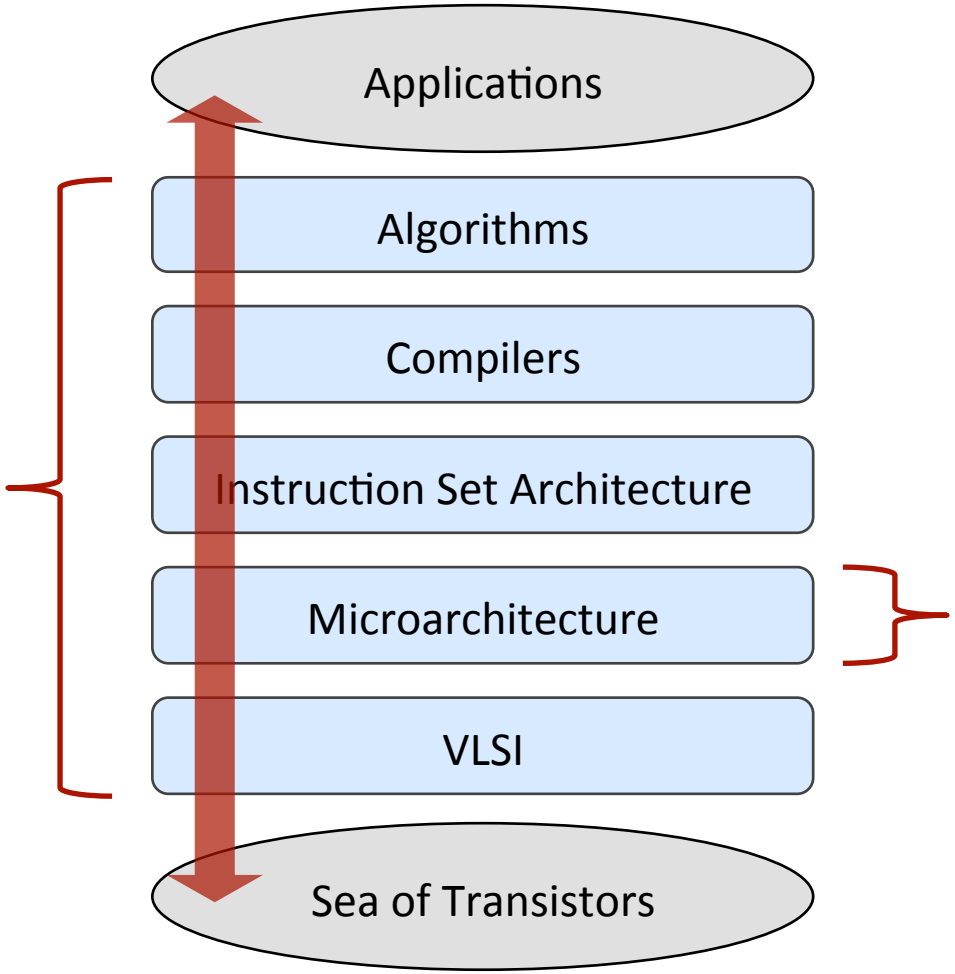
- Methodologies

- Patterns, Languages, Tools

Computer Architecture Research Abstractions



Computer Architecture Research Abstractions



Industry Development

Hundreds of Engineers

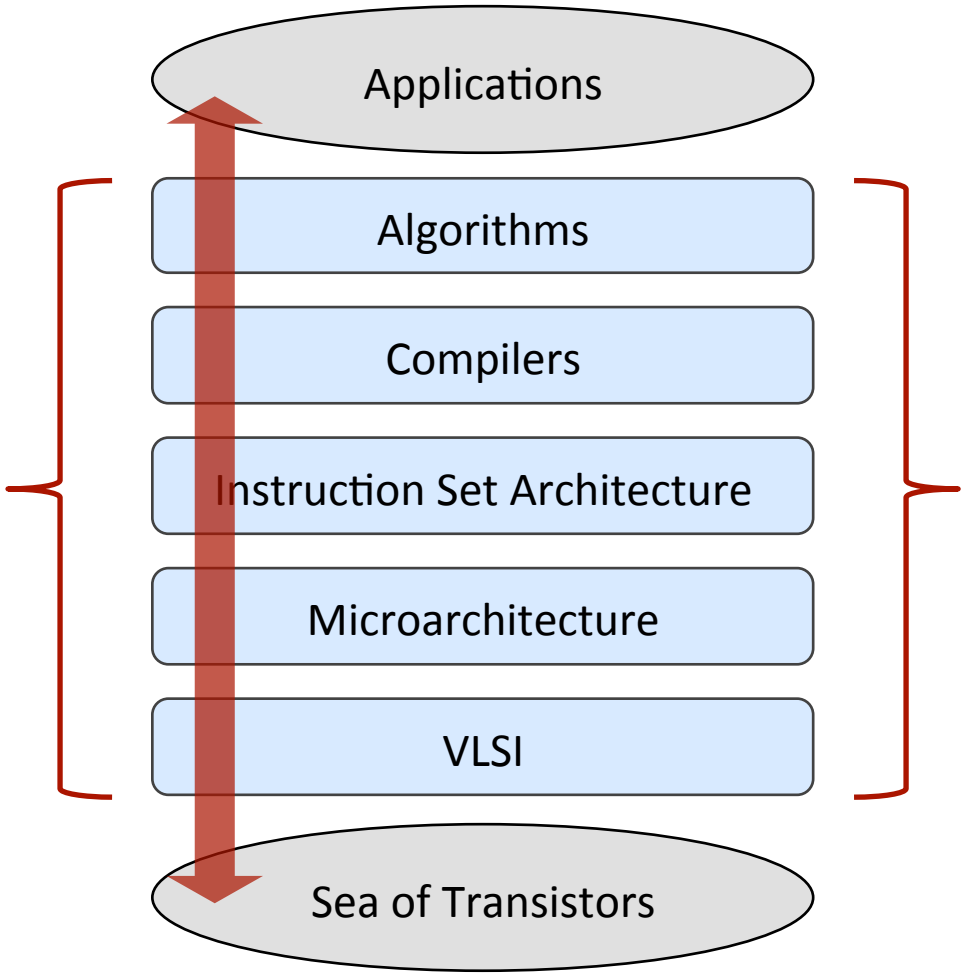
Academic Research

A Few Researchers

Computer Architecture Research Abstractions

Industry Development

Hundreds of Engineers

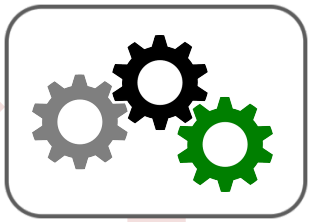


Academic Research

A Few Researchers

Computer Architecture Research Methodologies

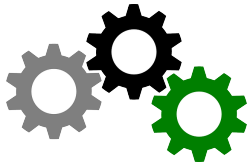
- Abstractions



- Methodologies

- Patterns, Languages, Tools

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

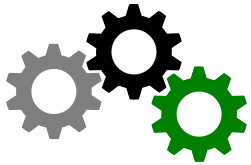
VLSI

Sea of Transistors

Cycle Level

- Behavior
- Timing

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

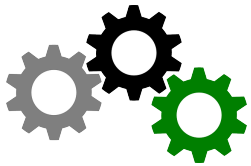
VLSI

Sea of Transistors

Cycle Level

- Behavior
- Timing

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

VLSI

Sea of Transistors

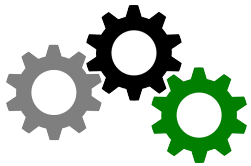
Functional Level

- Behavior

Cycle Level

- Behavior
- Timing

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

VLSI

Sea of Transistors

Functional Level

- Behavior

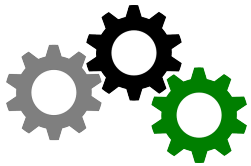
Cycle Level

- Behavior
- Timing

Register Transfer Level

- Behavior
- Timing
- Physical Resources

Computer Architecture Research Methodologies



Applications

Algorithms

Compilers

Instruction Set Architecture

Microarchitecture

VLSI

Sea of Transistors

Functional Level

- Behavior

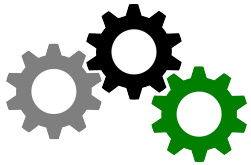
Cycle Level

- Behavior
- Timing

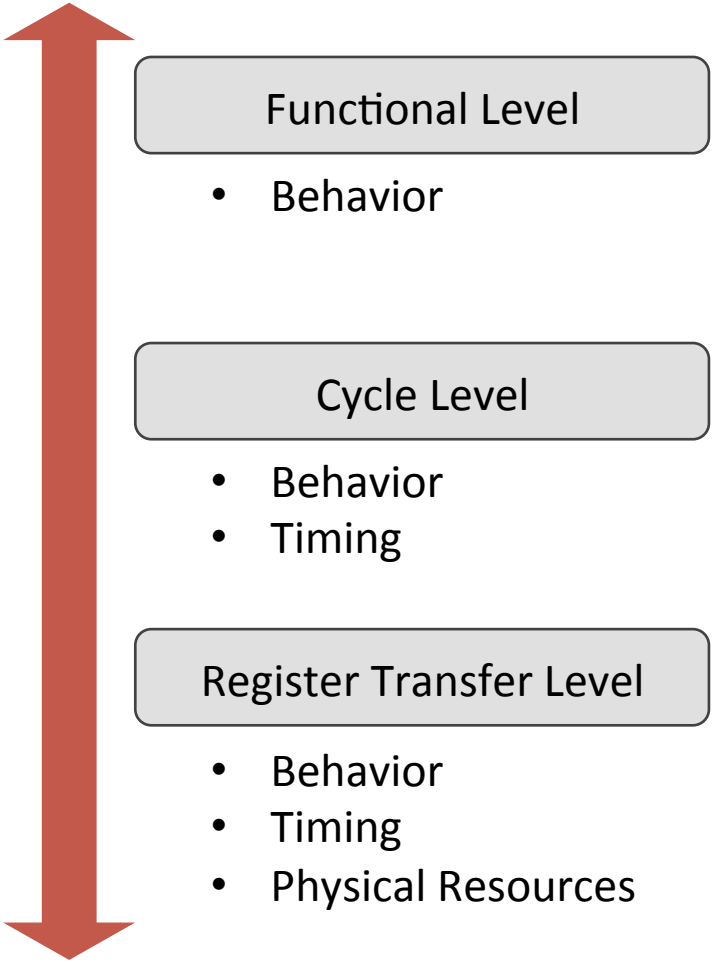
Register Transfer Level

- Behavior
- Timing
- Physical Resources

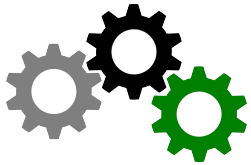
Computer Architecture Research Methodologies



Modeling Towards Layout



Computer Architecture Research Methodologies



Greater Simulation Speed

Modeling Towards Layout

Greater Model Detail

Functional Level

Algorithm and ISA Development

Cycle Level

Design Space Exploration

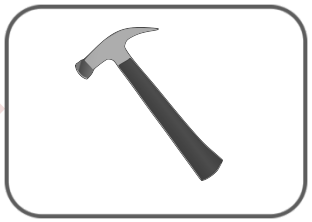
Register Transfer Level

Area/Energy/Timing Validation and Prototype Development

Computer Architecture Research Frameworks

- Abstractions

- Methodologies



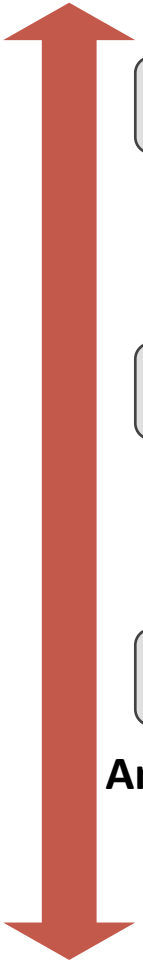
- Patterns, Languages, Tools

Computer Architecture Research Frameworks

MATLAB/Python Algorithm or
C++ Instruction Set Simulator

C++ Computer Architecture
Simulation Framework
(Object-Oriented)

Verilog or VHDL Design with
EDA Toolflow
(Concurrent-Structural)



Functional Level

Algorithm and ISA Development

Cycle Level

Design Space Exploration

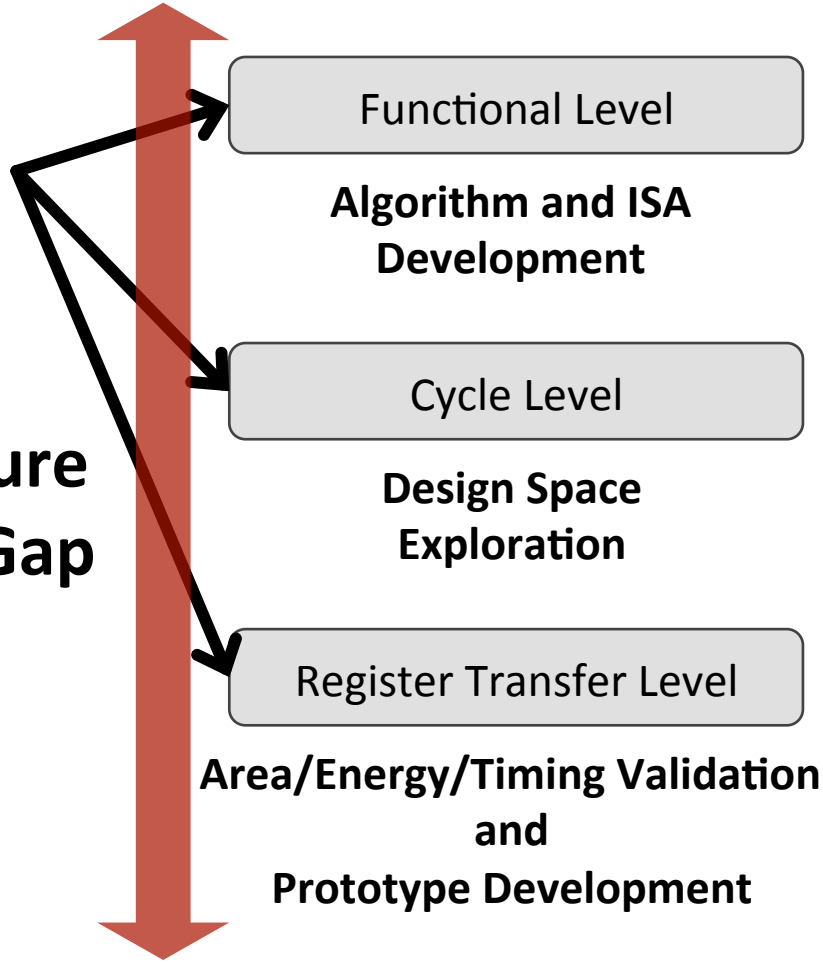
Register Transfer Level

Area/Energy/Timing Validation and Prototype Development

Computer Architecture Research Frameworks

Different languages, patterns, and tools!

The Computer Architecture Research Methodology Gap



Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)

Consistent interfaces across abstractions

Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC) — Consistent interfaces across abstractions
- **Unified Modeling Languages** (SystemC) — Unified design environment for FL, CL, RTL

Great Ideas From Prior Work

- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC) — Consistent interfaces across abstractions
- **Unified Modeling Languages** (SystemC) — Unified design environment for FL, CL, RTL
- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL) — Productive RTL design space exploration

Great Ideas From Prior Work

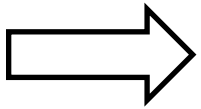
- **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC) — Consistent interfaces across abstractions
- **Unified Modeling Languages** (SystemC) — Unified design environment for FL, CL, RTL
- **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL) — Productive RTL design space exploration
- **HDL-Integrated Simulation Frameworks** (Cascade) — Productive RTL validation and cosimulation

Great Ideas From Prior Work

• **Concurrent-Structural Modeling** (Liberty, Cascade, SystemC)	Consistent interfaces across abstractions
• **Unified Modeling Languages** (SystemC)	Unified design environment for FL, CL, RTL
• **Hardware Generation Languages** (Chisel, Genesis2, BlueSpec, MyHDL)	Productive RTL design space exploration
• **HDL-Integrated Simulation Frameworks** (Cascade)	Productive RTL validation and cosimulation
• **Latency-Insensitive Interfaces** (Liberty, BlueSpec)	Component and test bench reuse

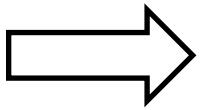
Outline

The Computer Architecture Research Methodology Gap



PyMTL

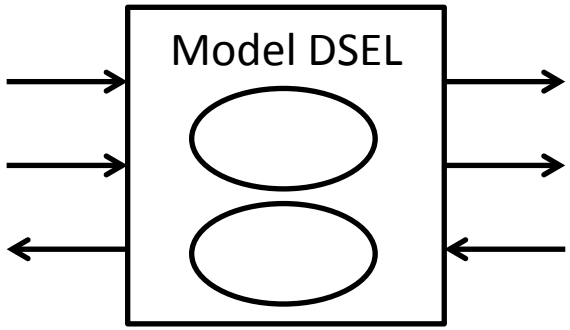
The Performance-Productivity Gap



SimJIT

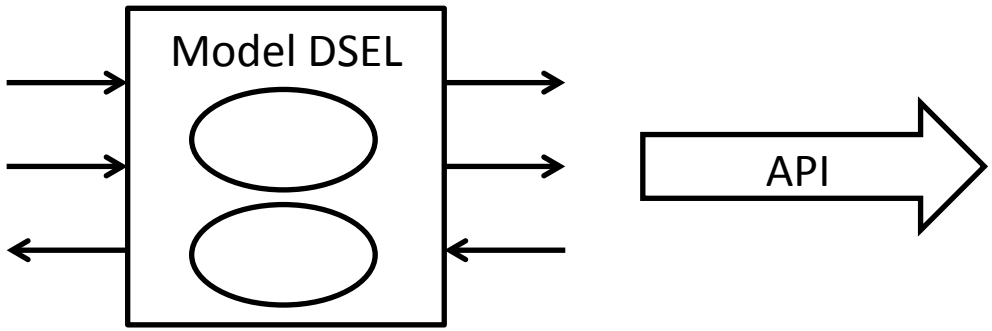
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling



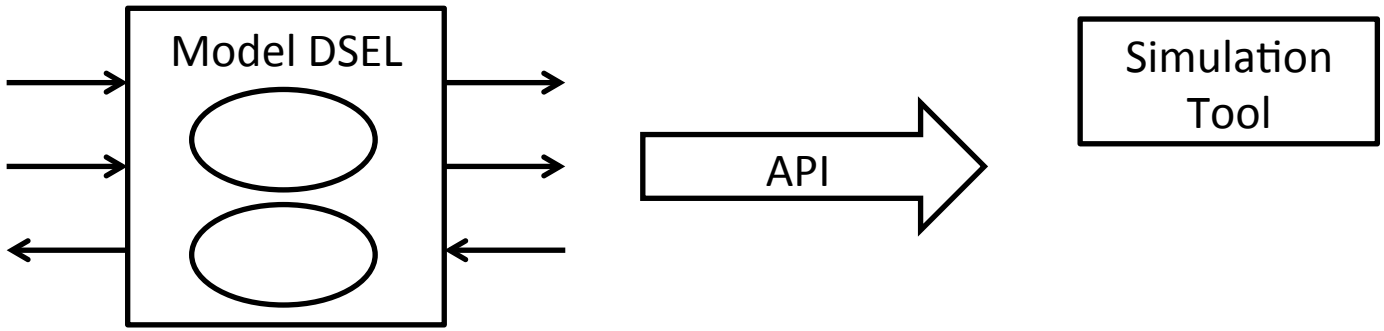
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL



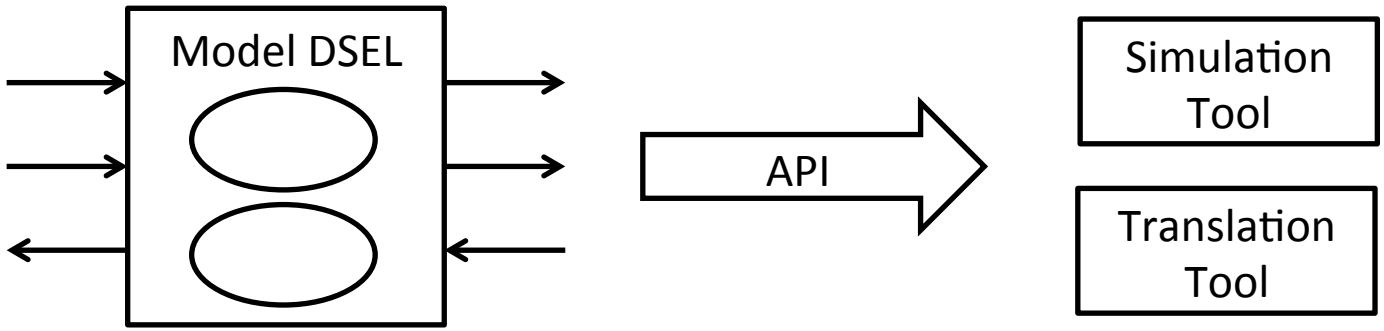
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
- A Python tool for simulating PyMTL FL, CL, and RTL models



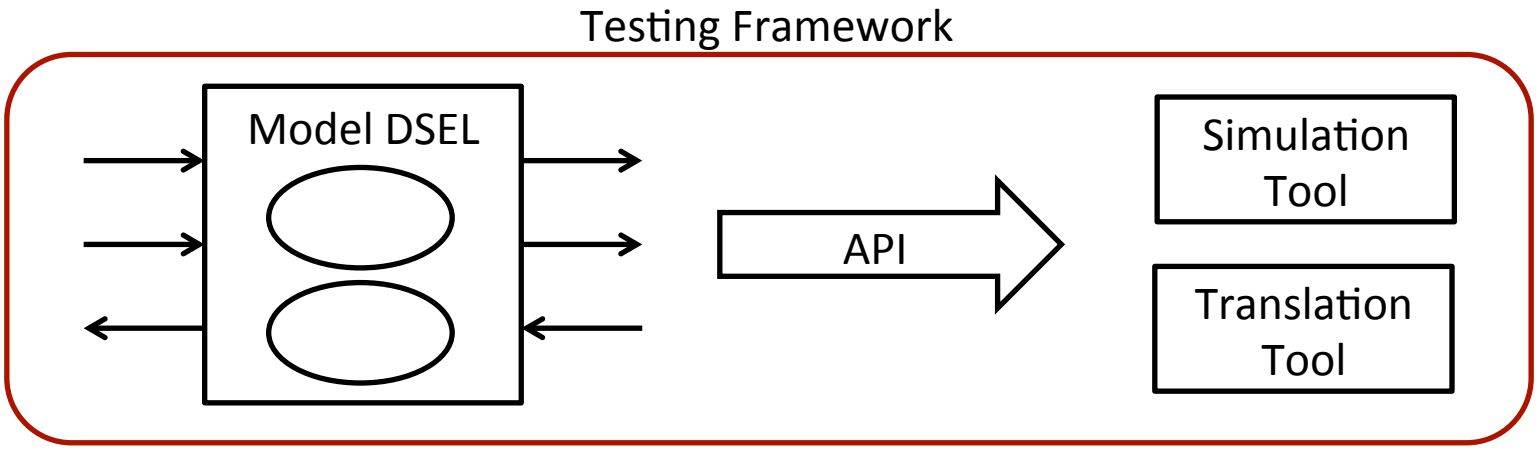
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
- A Python tool for simulating PyMTL FL, CL, and RTL models
- A Python tool for translating PyMTL RTL models into Verilog



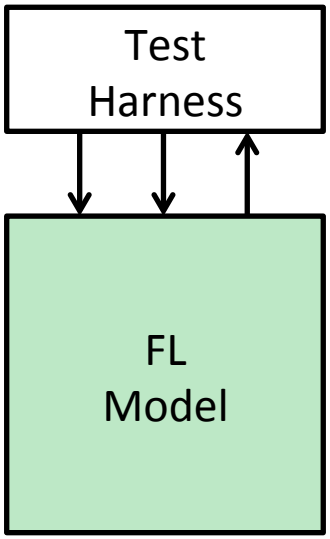
What is PyMTL?

- A Python DSEL for concurrent-structural hardware modeling
- A Python API for analyzing models described in the PyMTL DSEL
- A Python tool for simulating PyMTL FL, CL, and RTL models
- A Python tool for translating PyMTL RTL models into Verilog
- A Python testing framework for model validation



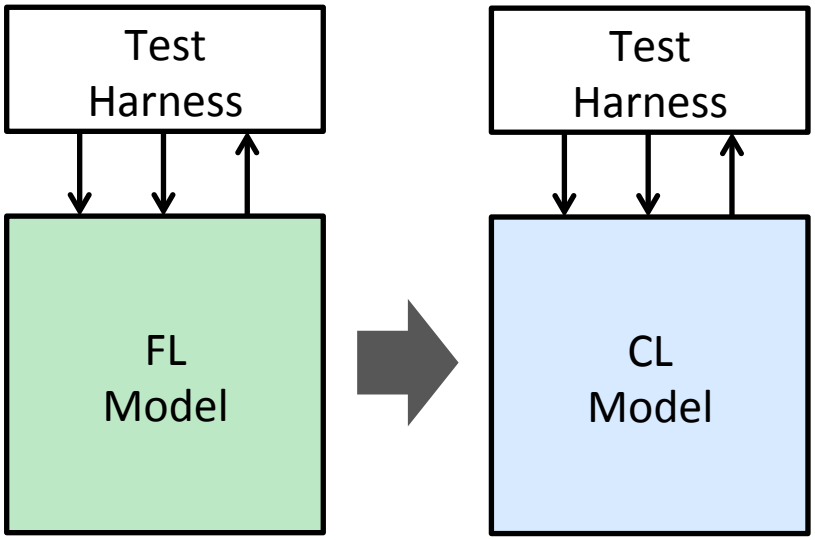
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation



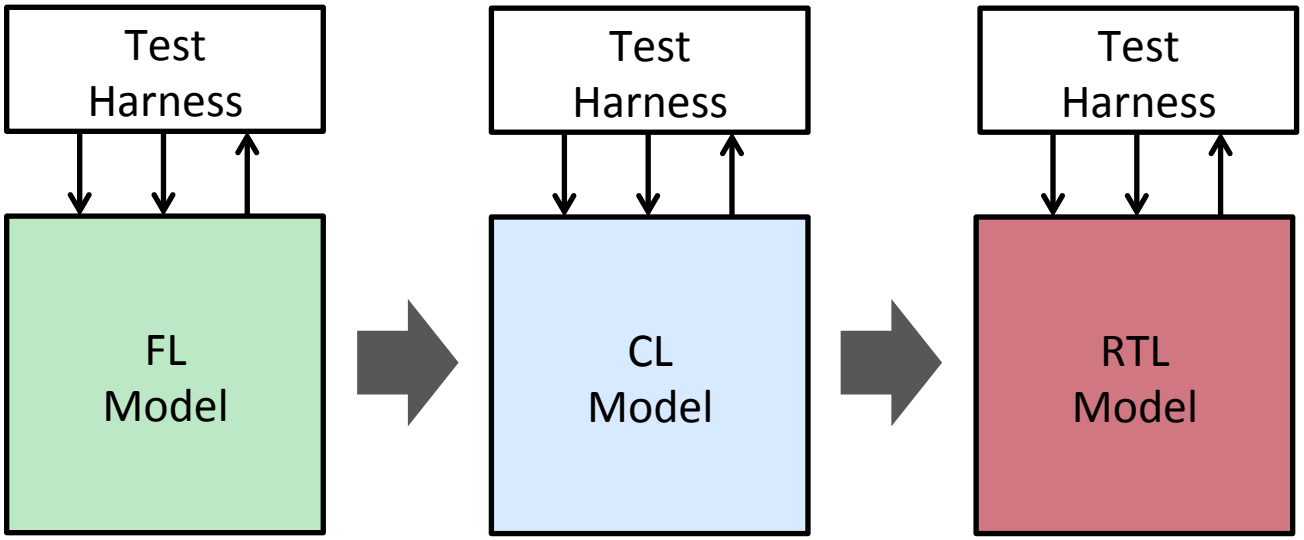
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation



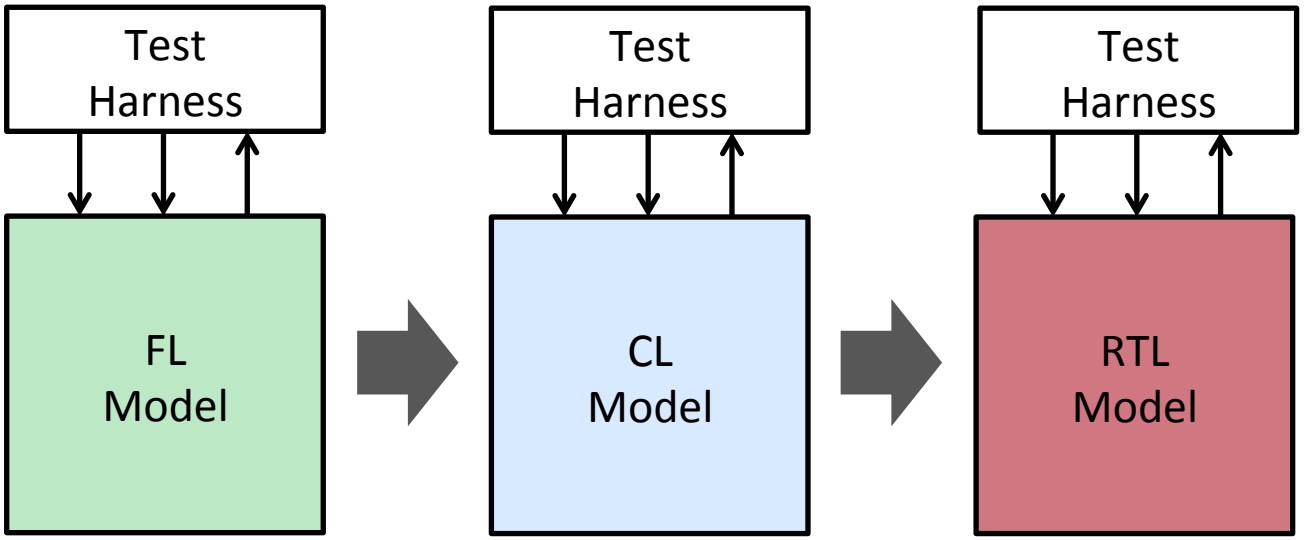
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation



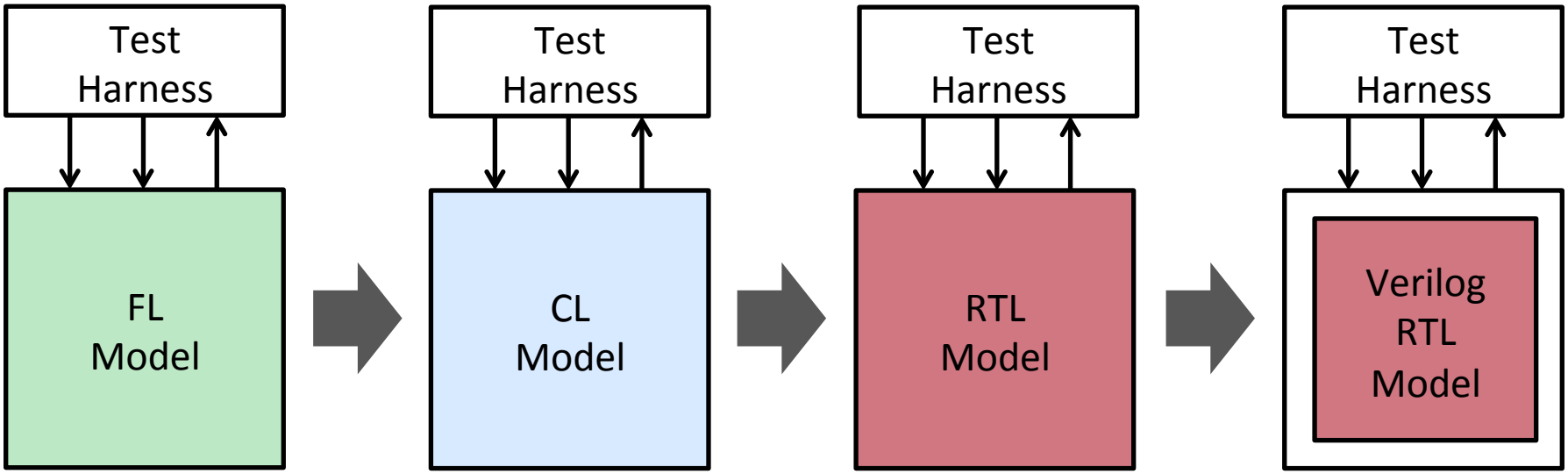
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog



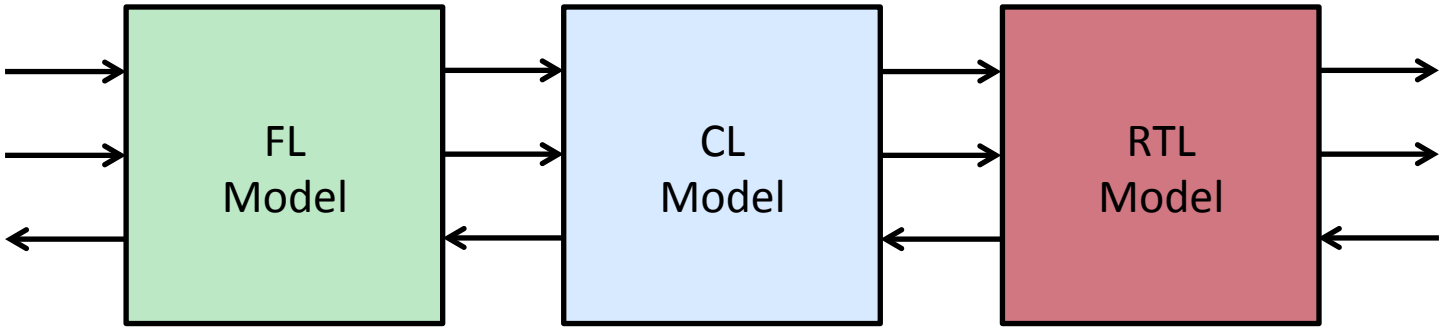
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog



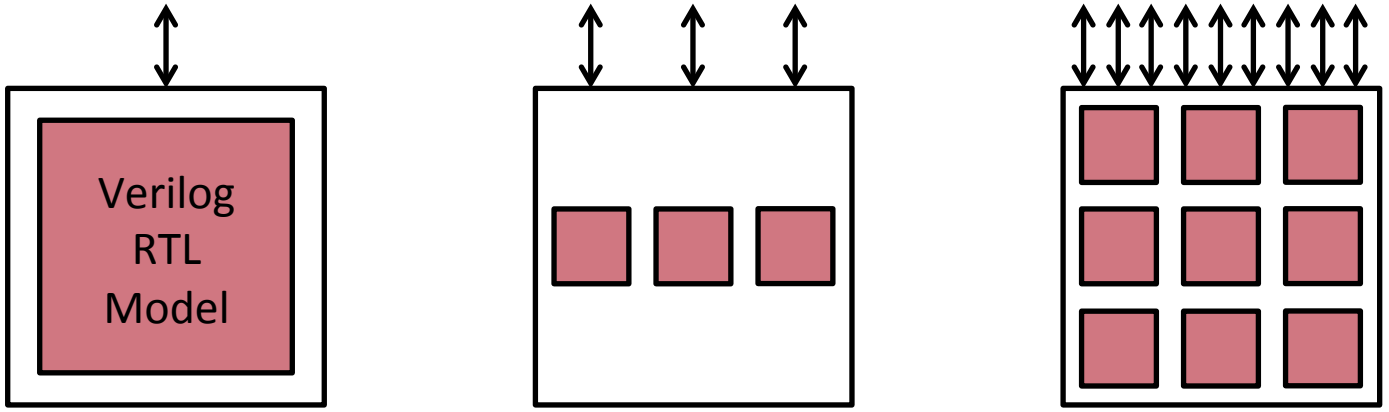
What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models



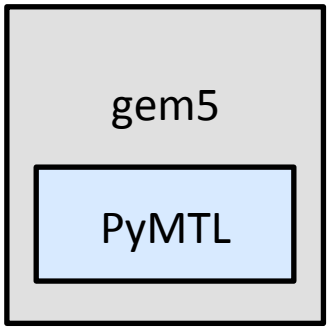
What Does PyMTL Enable?

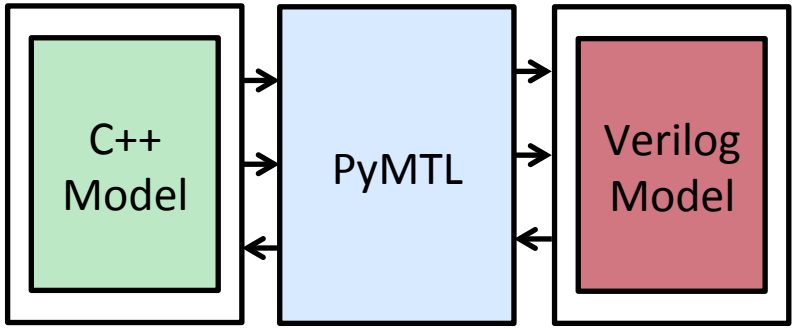
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators



What Does PyMTL Enable?

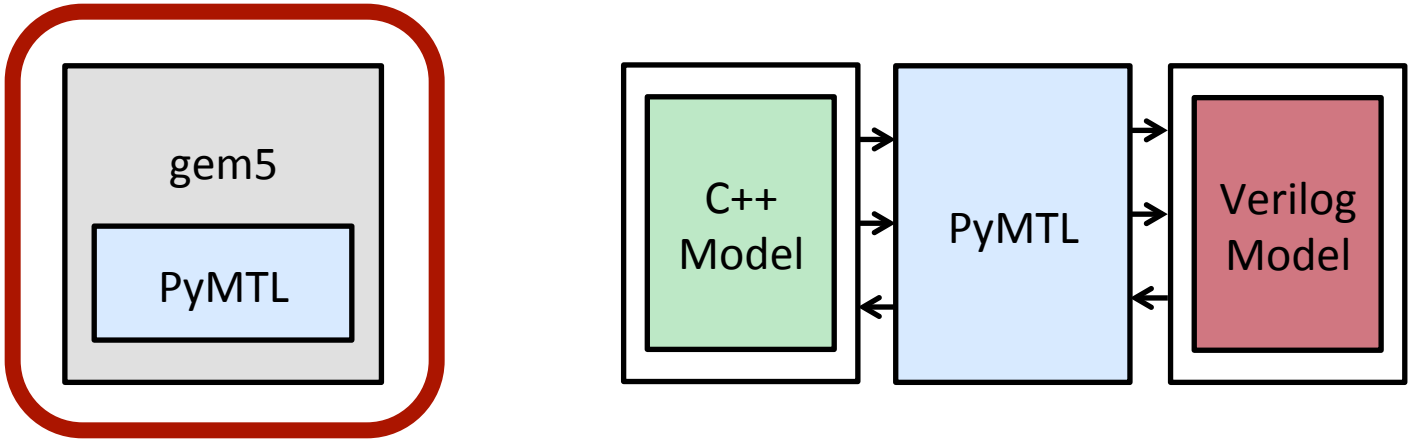
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog models





What Does PyMTL Enable?

- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators
- Embedding within C++ frameworks & integration of C++/Verilog models (*see Srinath et. al. in MICRO-47, Session 6B!)*



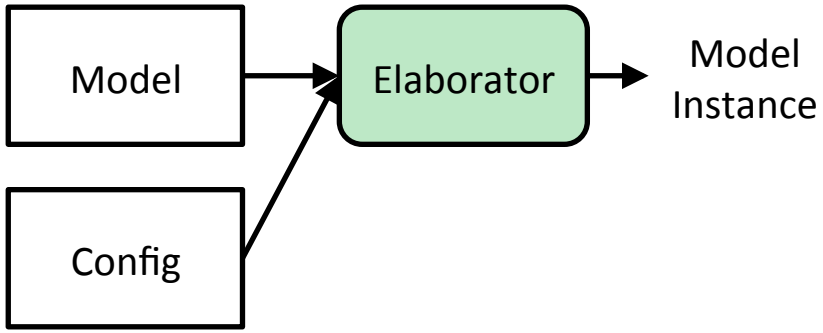
The PyMTL Framework

Specification **Tools** Output

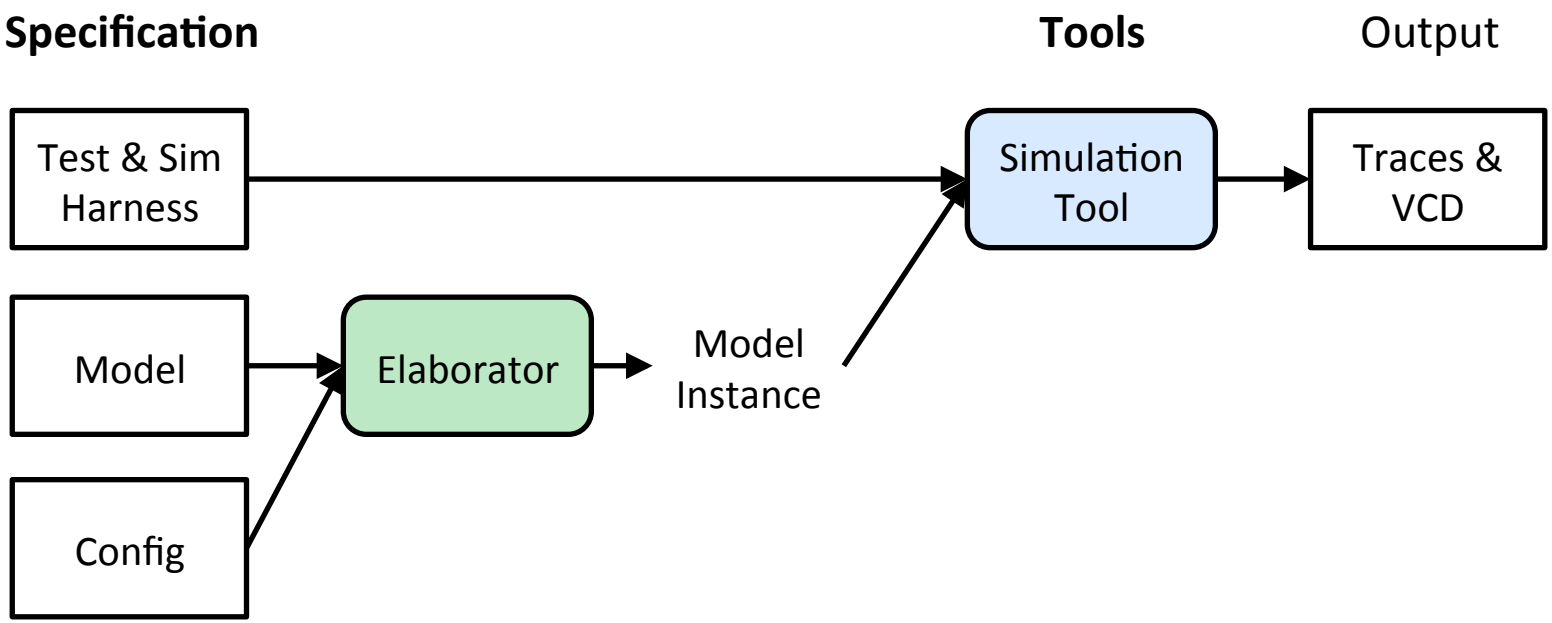
Model

The PyMTL Framework

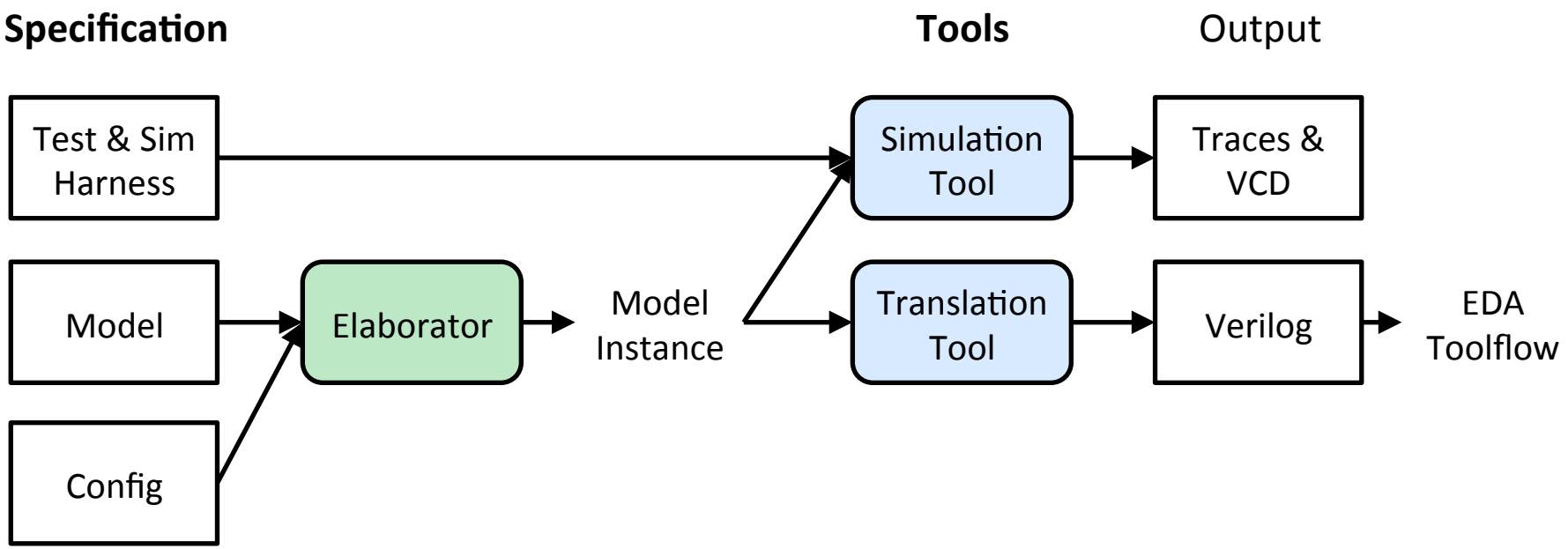
Specification **Tools** Output



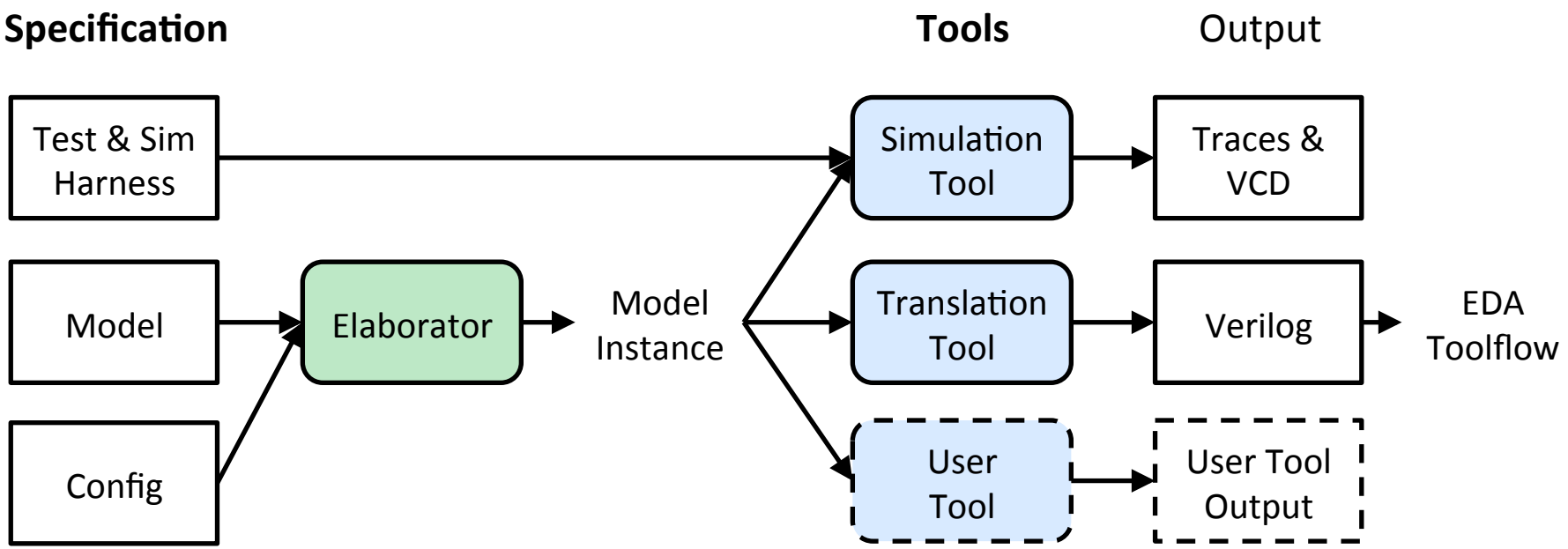
The PyMTL Framework



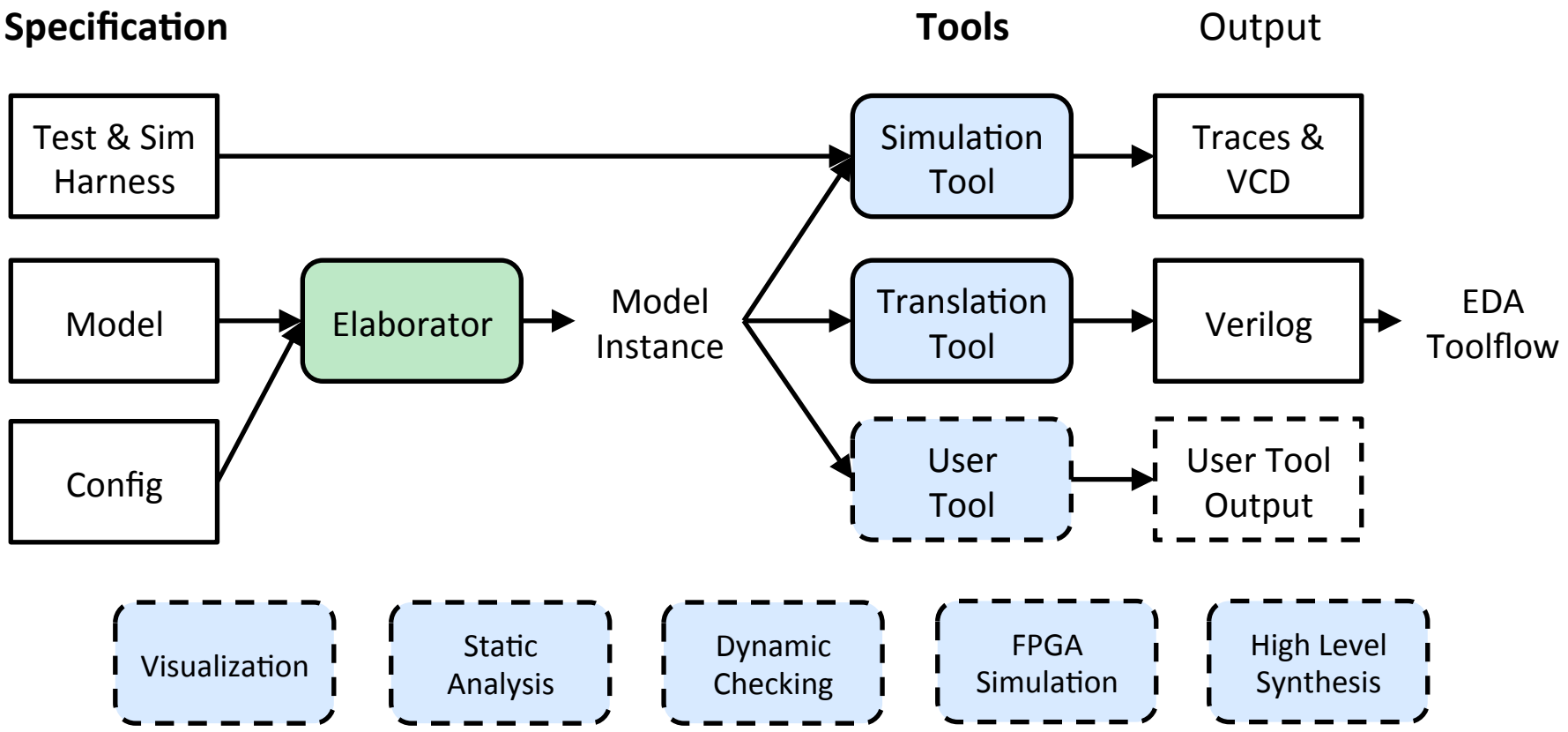
The PyMTL Framework



The PyMTL Framework

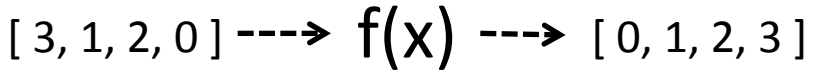


The PyMTL Framework



The PyMTL DSEL

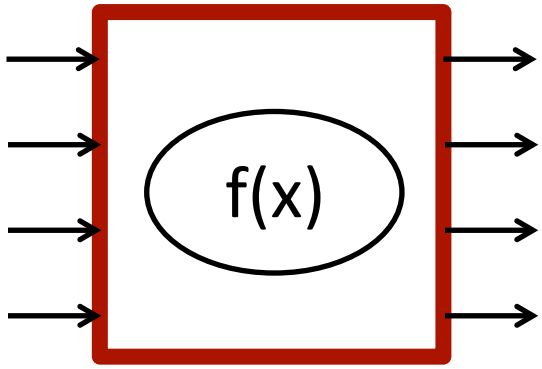
```
def sorter_network( input ):
  return sorted( input )
```



The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )

class SorterNetworkFL( Model )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

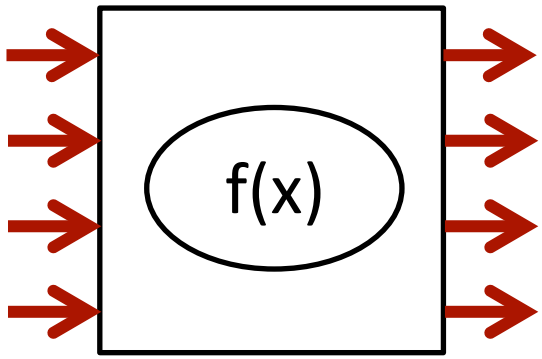


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkFL( Model ):
  def __init__( s, nbits, nports ):
    type   = Bits( nbits )
    s.in_  = InPort [nports]( type )
    s.out  = OutPort[nports]( type )
```

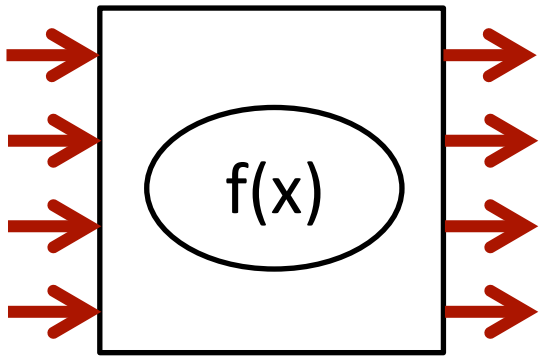


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkFL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)
```

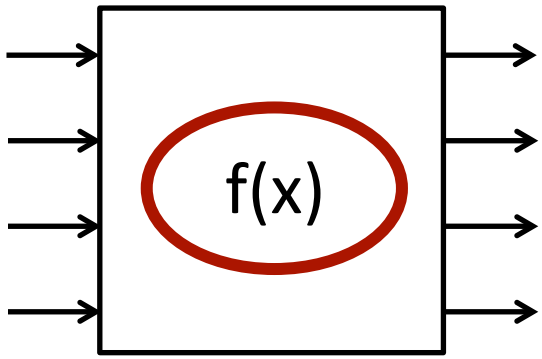


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkFL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)


    @s.tick_fl
    def logic():
```

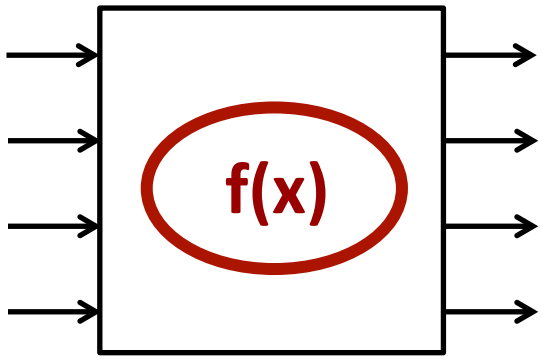


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkFL( Model ):
  def __init__( s, nbits, nports ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)
```



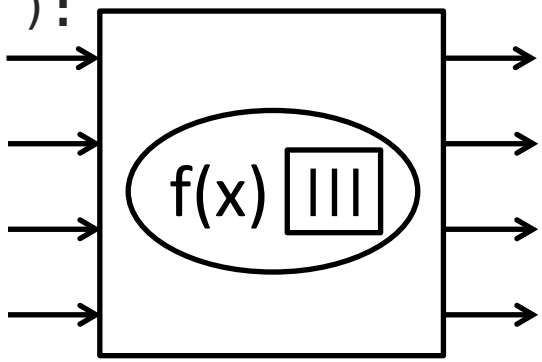

```
    @s.tick_fl
    def logic():
      for i, v in enumerate( sorted( s.in_ ) ):
        s.out[i].next = v
```

The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkCL( Model ):
  def __init__( s, nbits, nports, delay=3 ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)


    @s.tick_cl
    def logic():
```

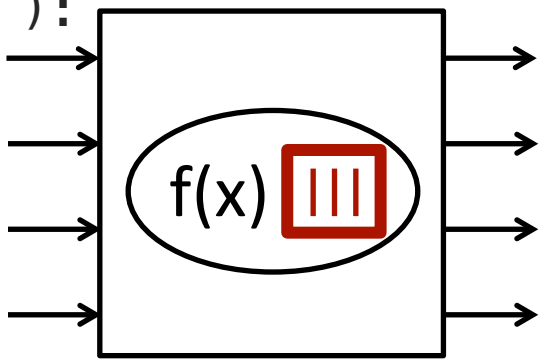


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

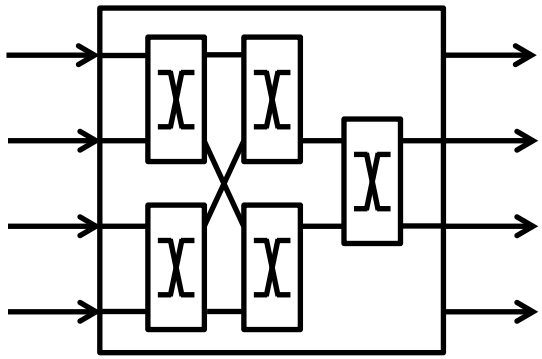
```
class SorterNetworkCL( Model ):
  def __init__( s, nbits, nports, delay=3 ):

    s.in_  = InPort [nports](nbits)
    s.out  = OutPort[nports](nbits)
    s.pipe = Pipeline( delay )

    @s.tick_cl
    def logic():
      s.pipe.xtick()
      s.pipe.push( sorted( s.in_ ) )

      if s.pipe.ready():
        for i, v in enumerate( s.pipe.pop() ):
          s.out[i].next = v
```



The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )

class SorterNetworkRTL( Model )
  def __init__( s, nbits ):

    s.in_  = InPort [4](nbits)
    s.out  = OutPort[4](nbits)
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

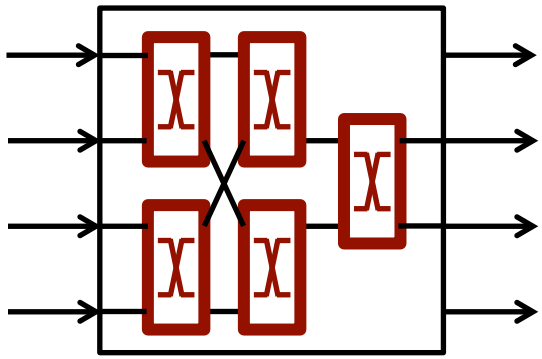


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkRTL( Model )
  def __init__( s, nbits ):

    s.in_  = InPort [4](nbits)
    s.out  = OutPort[4](nbits)

    s.m = m = MinMaxRTL[5](nbits)
```

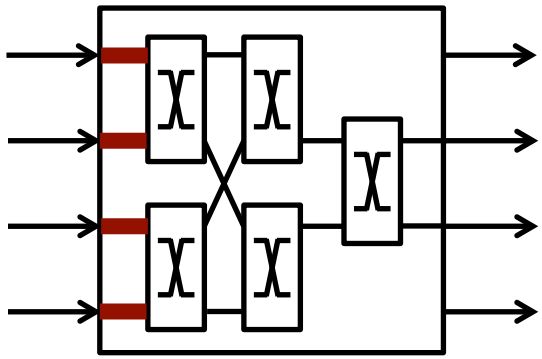


The PyMTL DSEL

```
def sorter_network( input ):
  return sorted( input )
```

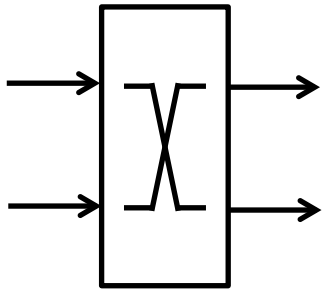
[3, 1, 2, 0] ---> f(x) ---> [0, 1, 2, 3]

```
class SorterNetworkRTL( Model )
  def __init__( s, nbits ):

    s.in_  = InPort [4](nbits)
    s.out  = OutPort[4](nbits)

    s.m = m = MinMaxRTL[5](nbits)

    s.connect( s.in_[0], m[0].in_[0] )
    s.connect( s.in_[1], m[0].in_[1] )
    s.connect( s.in_[2], m[1].in_[0] )
    s.connect( s.in_[3], m[2].in_[1] )

    . . .
```



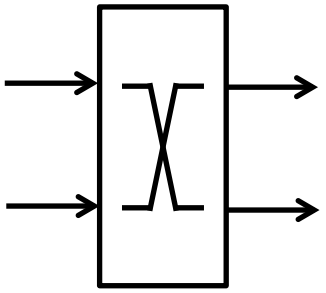
The PyMTL DSEL

```
class MinMaxRTL( Model )
  def __init__( s, nbits ):
    s.in_ = InPort [2](nbits)
    s.out = OutPort[2](nbits)
    @s.combinational
    def logic():
      swap = s.in_[0] > s.in_[1]
      s.out[0].value = s.in[1] if swap else s.in[0]
      s.out[1].value = s.in[0] if swap else s.in[1]
```

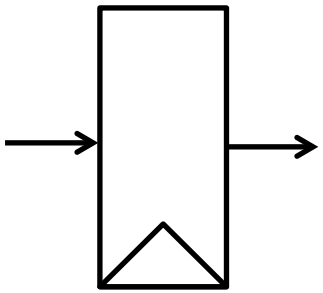


The PyMTL DSEL

```
class MinMaxRTL( Model )
  def __init__( s, nbits ):
    s.in_ = InPort [2](nbits)
    s.out = OutPort[2](nbits)
    @s.combinational
    def logic():
      swap = s.in_[0] > s.in_[1]
      s.out[0].value = s.in[1] if swap else s.in[0]
      s.out[1].value = s.in[0] if swap else s.in[1]
```



```
class RegRTL( Model )
  def __init__( s, nbits ):
    s.in_ = InPort (nbits)
    s.out = OutPort(nbits)
    @s.tick_rtl
    def logic():
      s.out.next = s.in_
```



The PyMTL DSEL

Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.

The PyMTL DSEL

Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.

Productivity helpers:

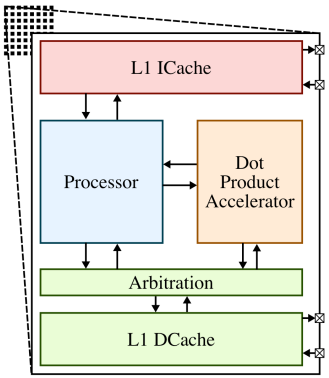
- MemoryProxies
- QueueAdapters
- PortBundles
- BitStructs
- TestSource
- TestSink

The PyMTL DSEL

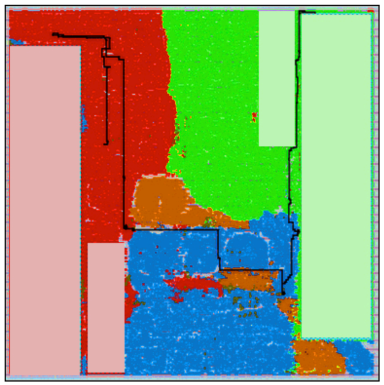
Testing of SorterFL, SorterCL, and SorterRTL can be greatly simplified by using latency-insensitive interfaces.

Productivity helpers:

- MemoryProxies
- QueueAdapters
- PortBundles
- BitStructs
- TestSource
- TestSink



(a) Block Diagram



(b) Post-Place-and-Route Layout

See the paper for more examples!

Why Python?

Benefits:

- Modern language features enable rapid prototyping (dynamic-typing, reflection, metaprogramming)
- Lightweight, pseudocode-like syntax
- Built-in support for integrating C/C++ code
- Large, active developer and support community

Why Python?

Benefits:

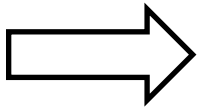
- Modern language features enable rapid prototyping (dynamic-typing, reflection, metaprogramming)
- Lightweight, pseudocode-like syntax
- Built-in support for integrating C/C++ code
- Large, active developer and support community

Drawbacks:

- Performance

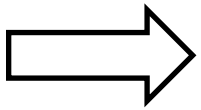
Outline

The Computer Architecture Research Methodology Gap



PyMTL

The Performance-Productivity Gap



SimJIT

Performance-Productivity Gap

Experiment:

- Simple 8x8 Mesh Network Model

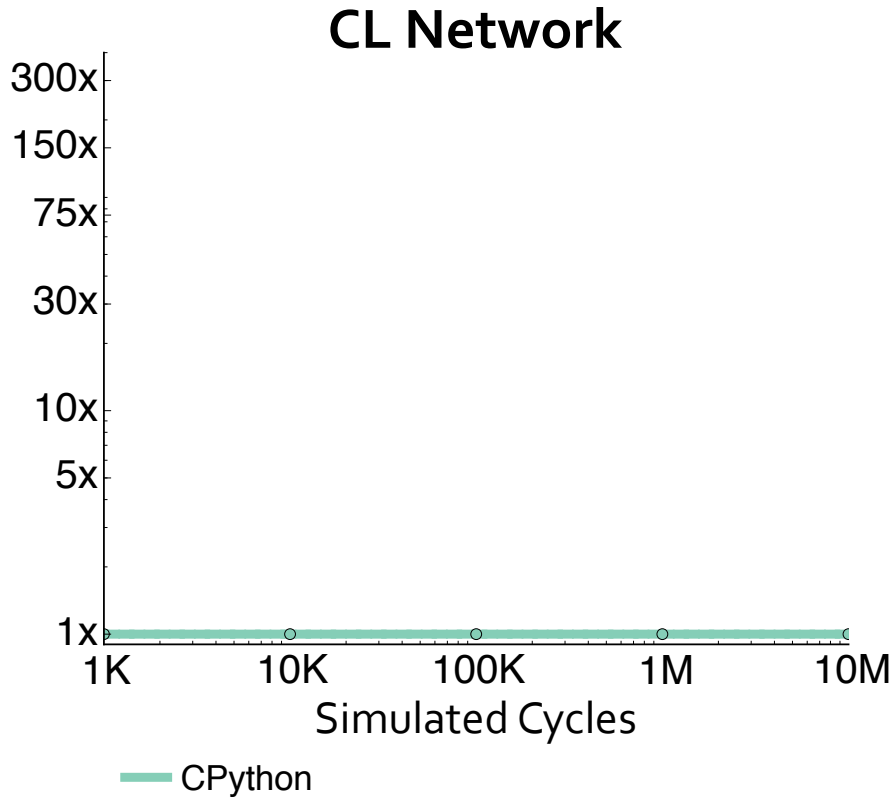
- Cycle-Precise CL Model:
  - PyMTL Model Simulated with the CPython Interpreter
  - Hand-Written C++ Model and Simulator

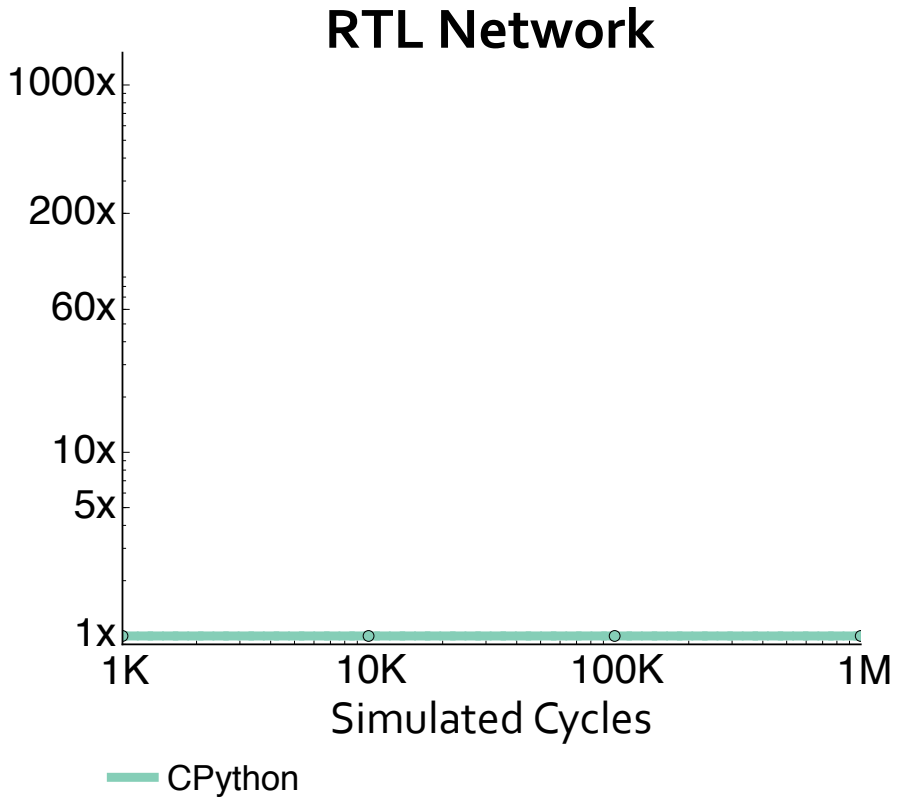
Performance-Productivity Gap

Experiment:

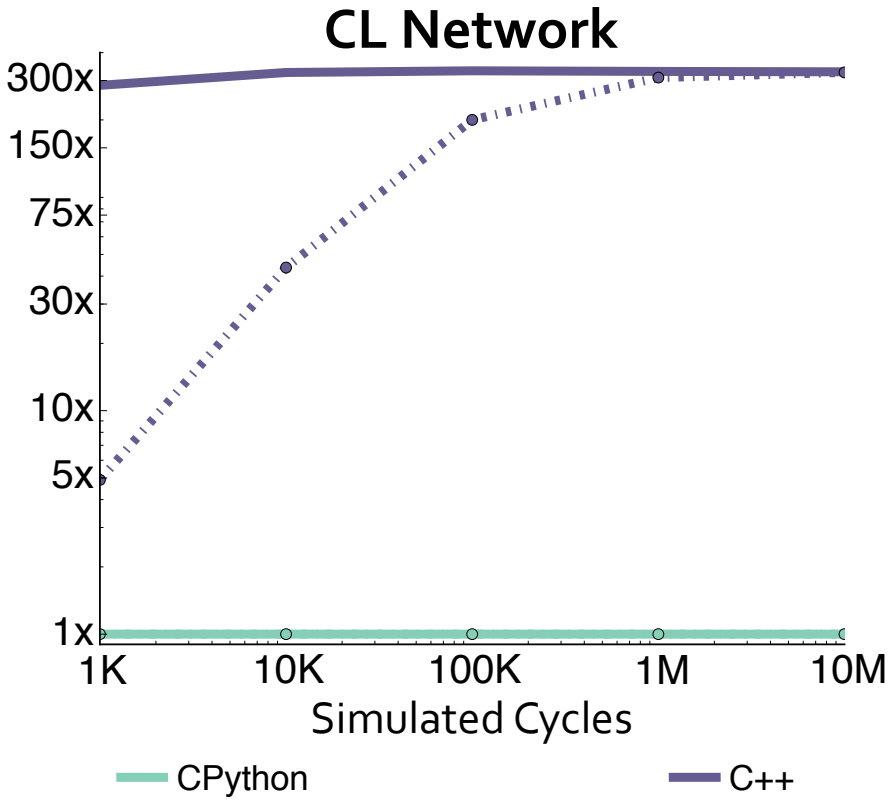
- Simple 8x8 Mesh Network Model
- Cycle-Precise CL Model:
  - PyMTL Model Simulated with the CPython Interpreter
  - Hand-Written C++ Model and Simulator
- Bit-Accurate RTL Model:
  - PyMTL Model Simulated with CPython Interpreter
  - Hand-Written Verilog RTL Simulated with Verilator

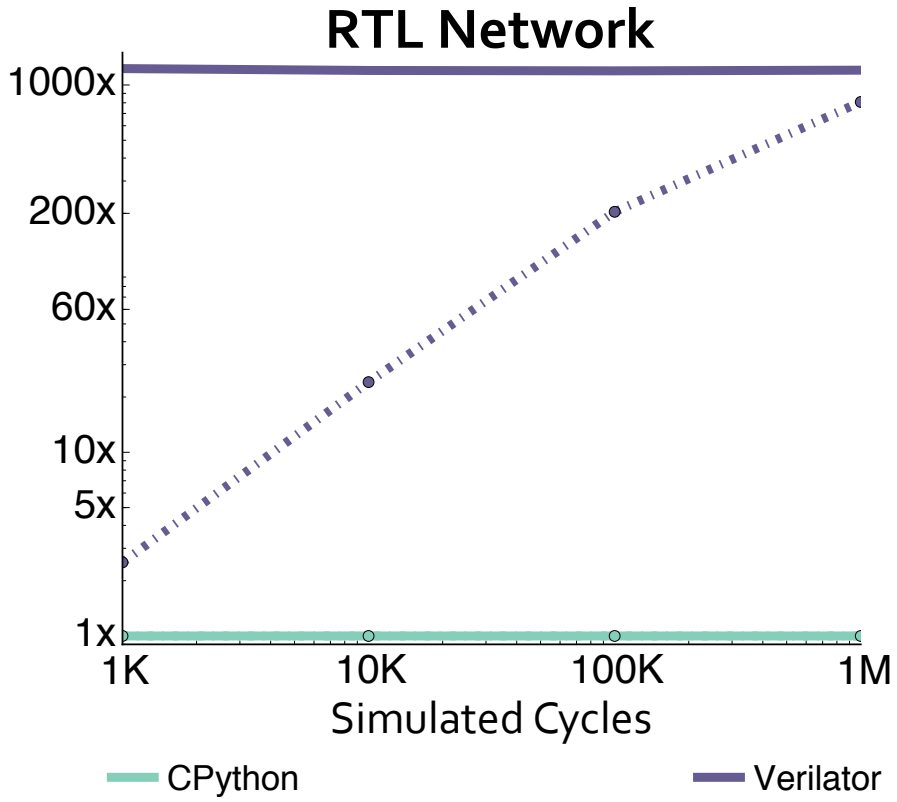
Performance-Productivity Gap





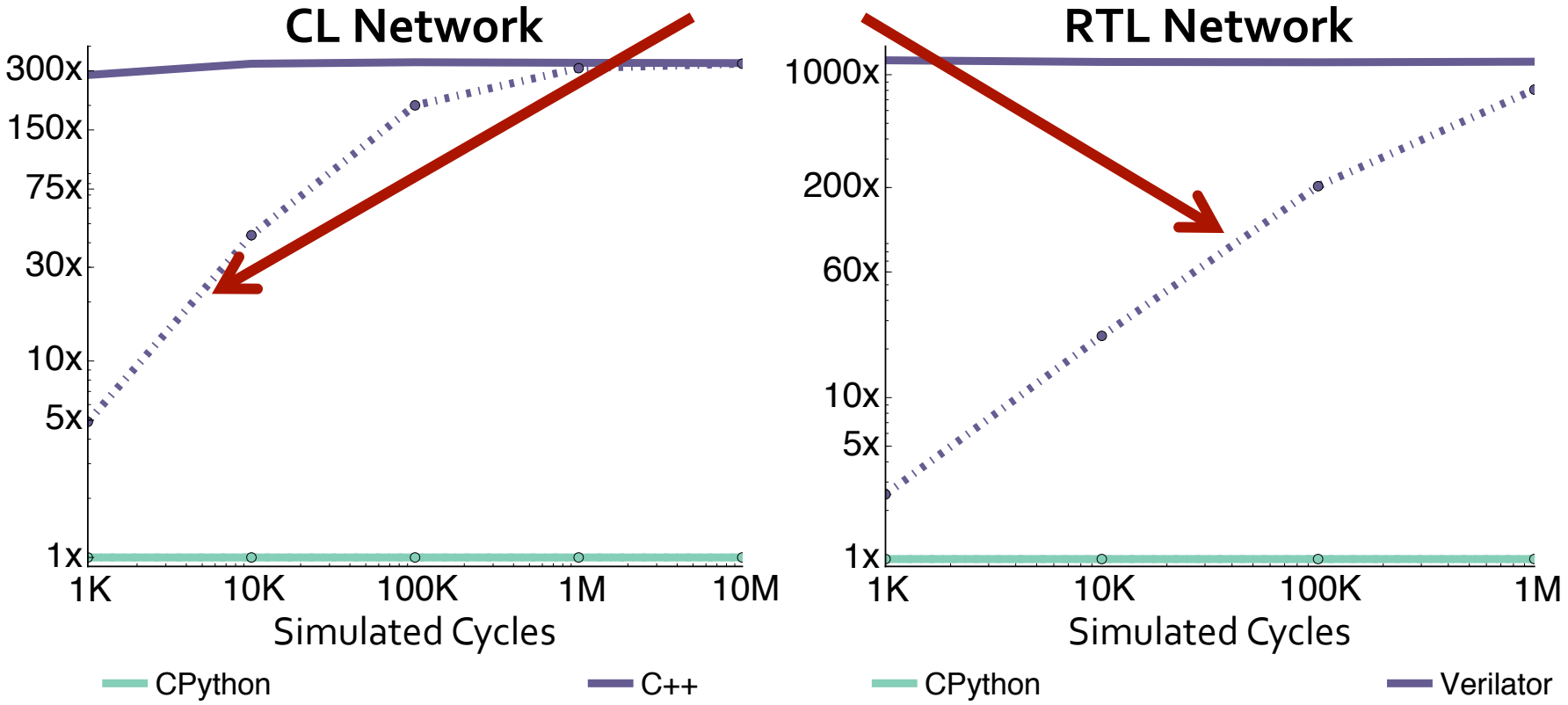
Performance-Productivity Gap





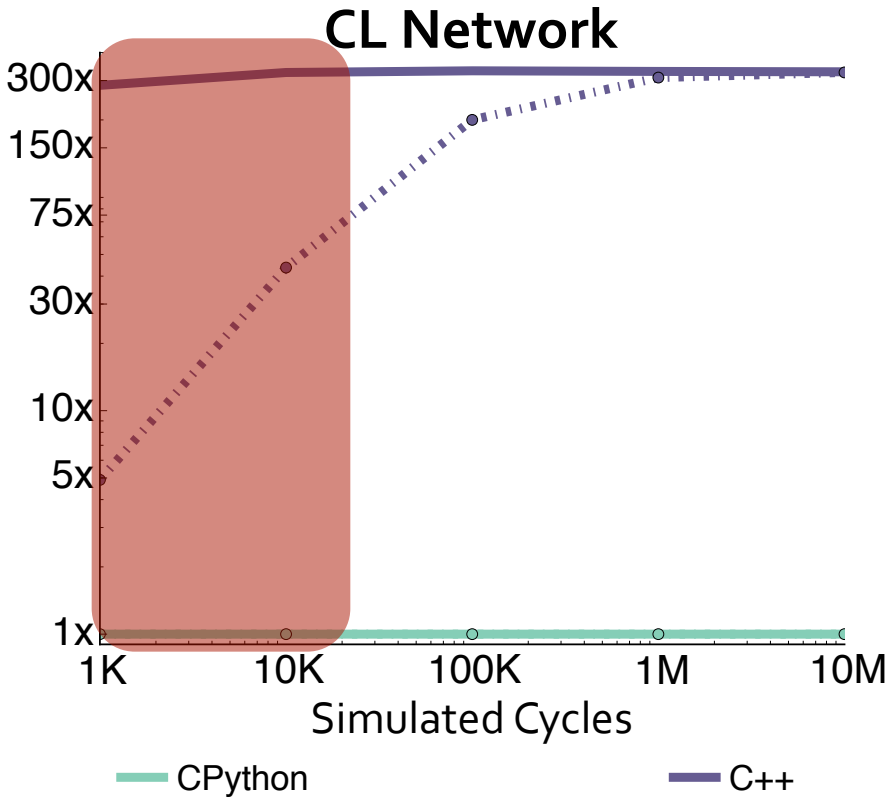
Performance-Productivity Gap

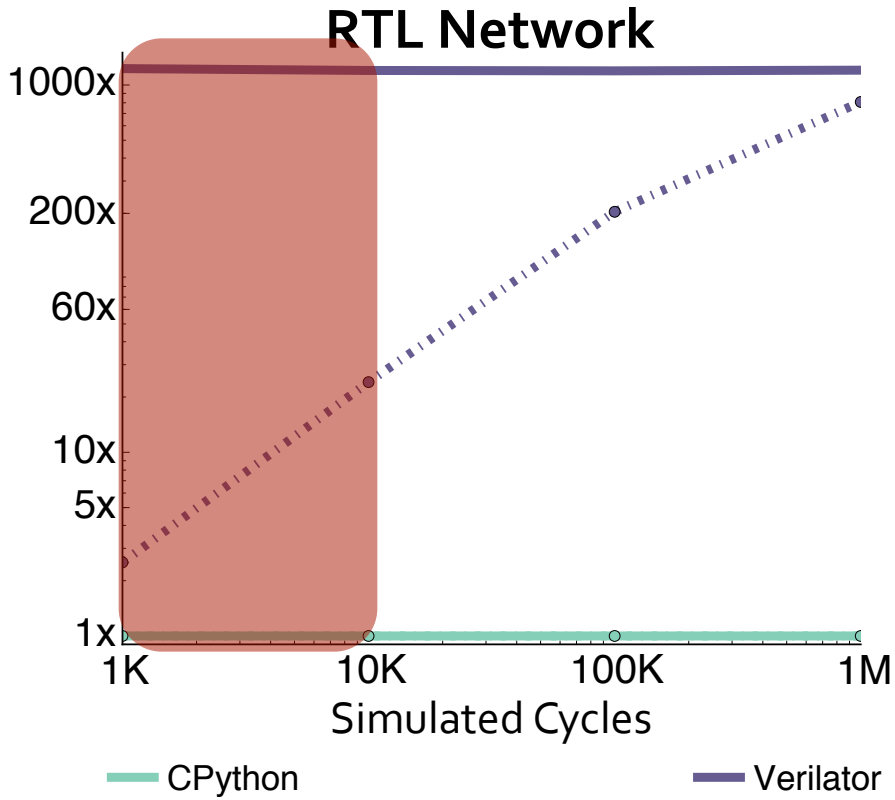
Performance degradation due to Compilation



Performance-Productivity Gap

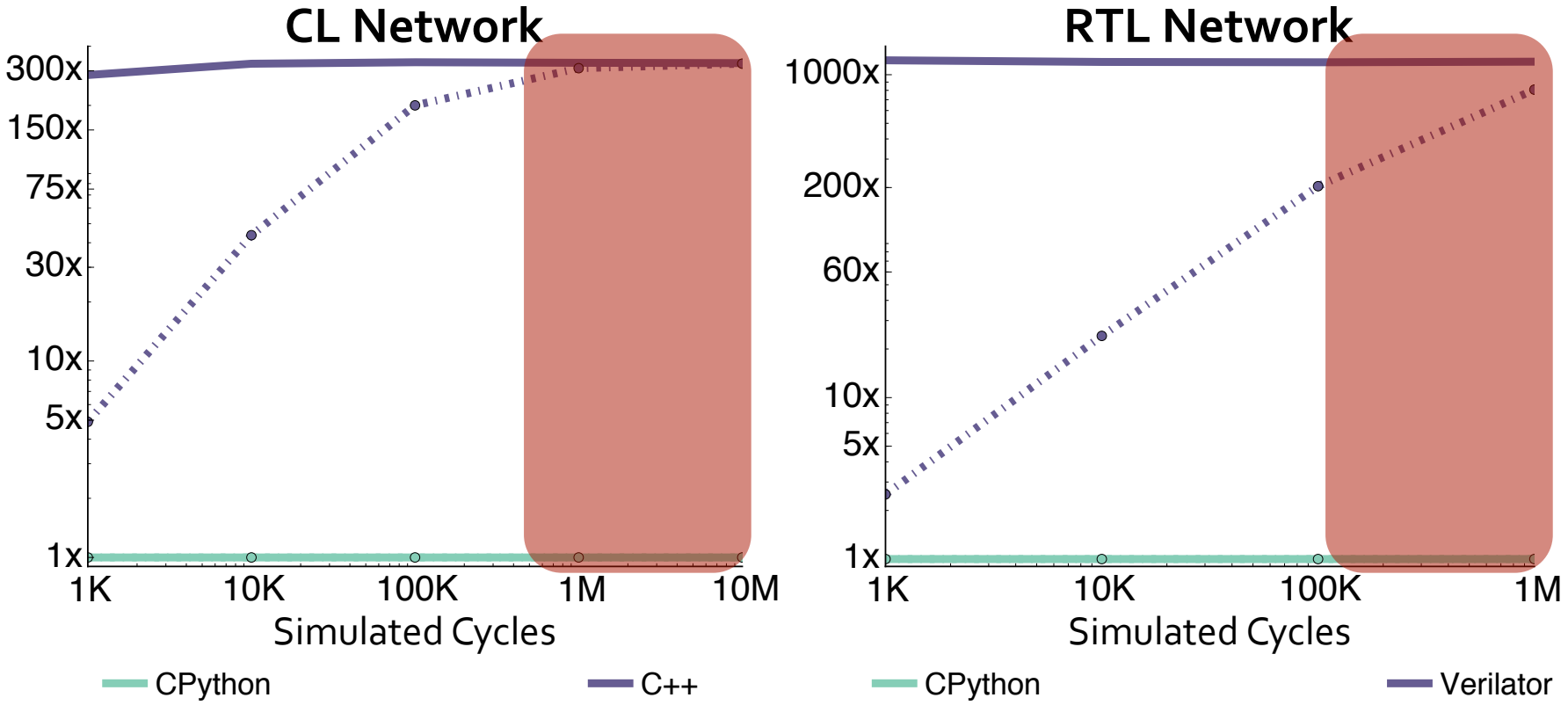
Short Simulations: Large-Compilation Overhead



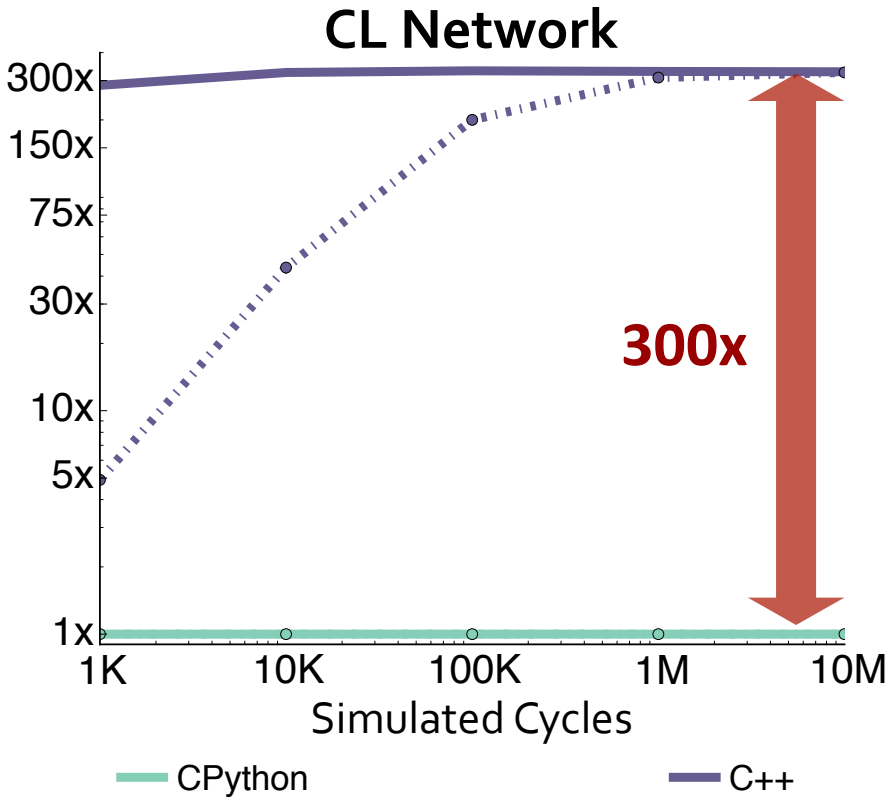


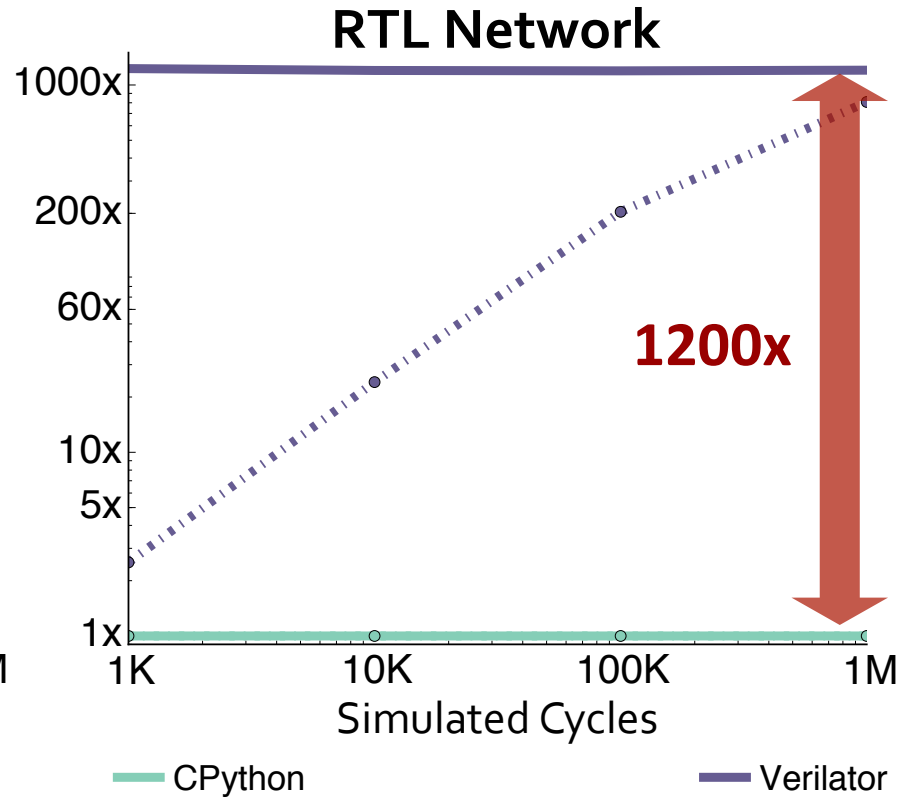
Performance-Productivity Gap

Long Simulations: Compilation Overhead Amortized



Performance-Productivity Gap





Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**

Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**

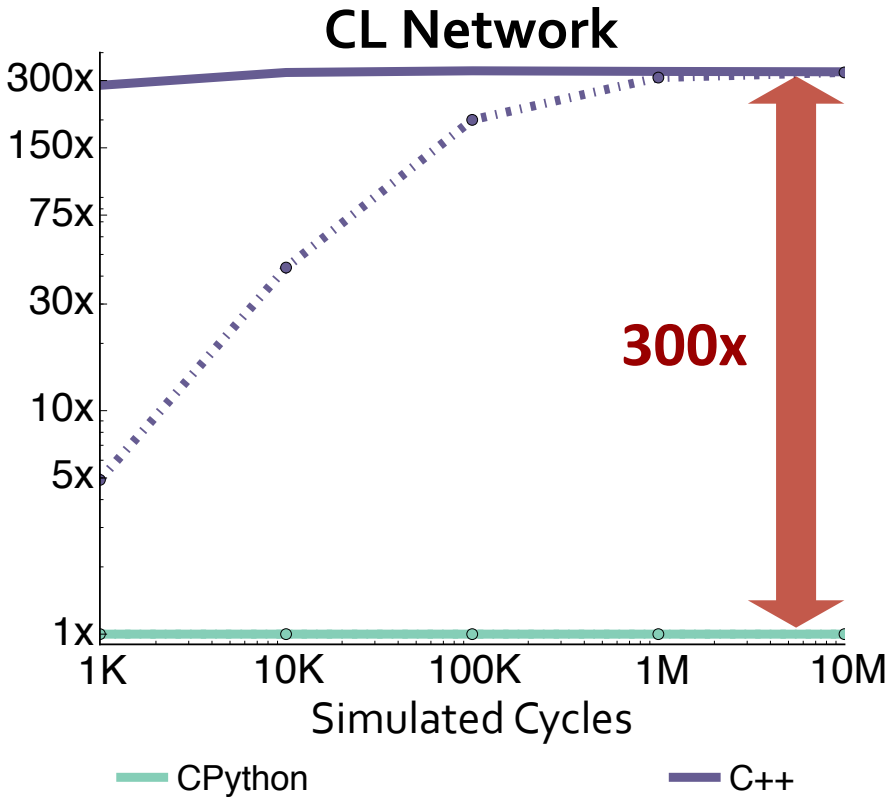
- **Python-Wrapped C/C++ Libraries**
  (NumPy, CVXOPT, NLPy, pythonOCC, GEM5)
- **Numerical Just-In-Time Compilers**
  (Numba, Parakeet)
- **Just-In-Time Compiled Interpreters**
  (PyPy, Pyston)
- **Selective Embedded Just-In-Time Specialization**
  (SEJITS)

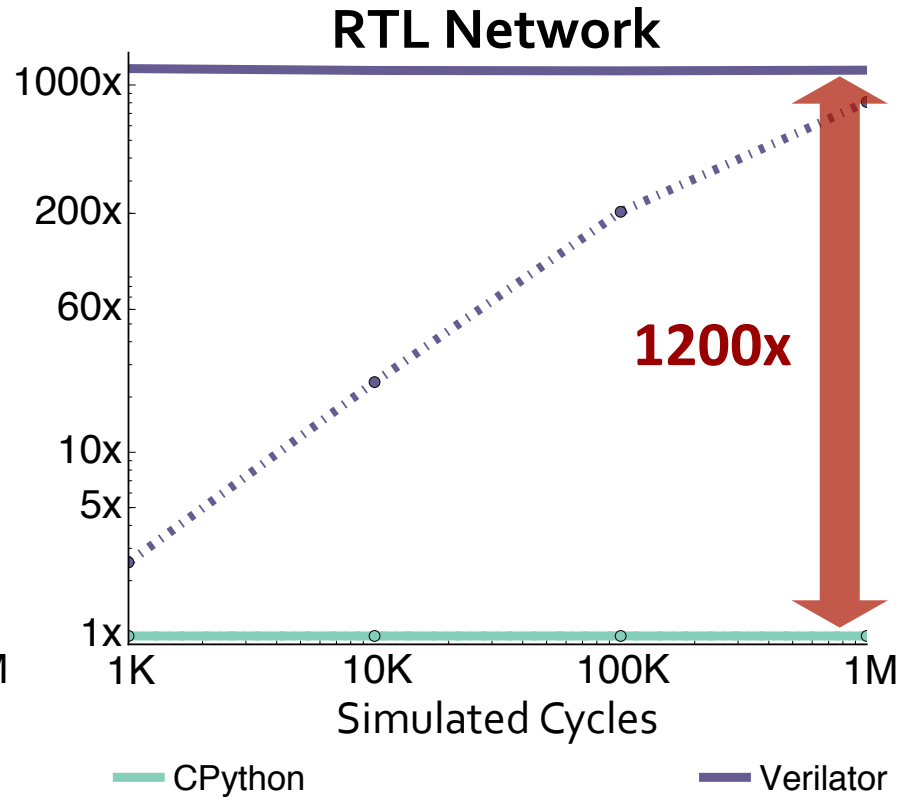
Performance-Productivity Gap

Python is growing in popularity in many domains of scientific and high-performance computing. **How do they close this gap?**

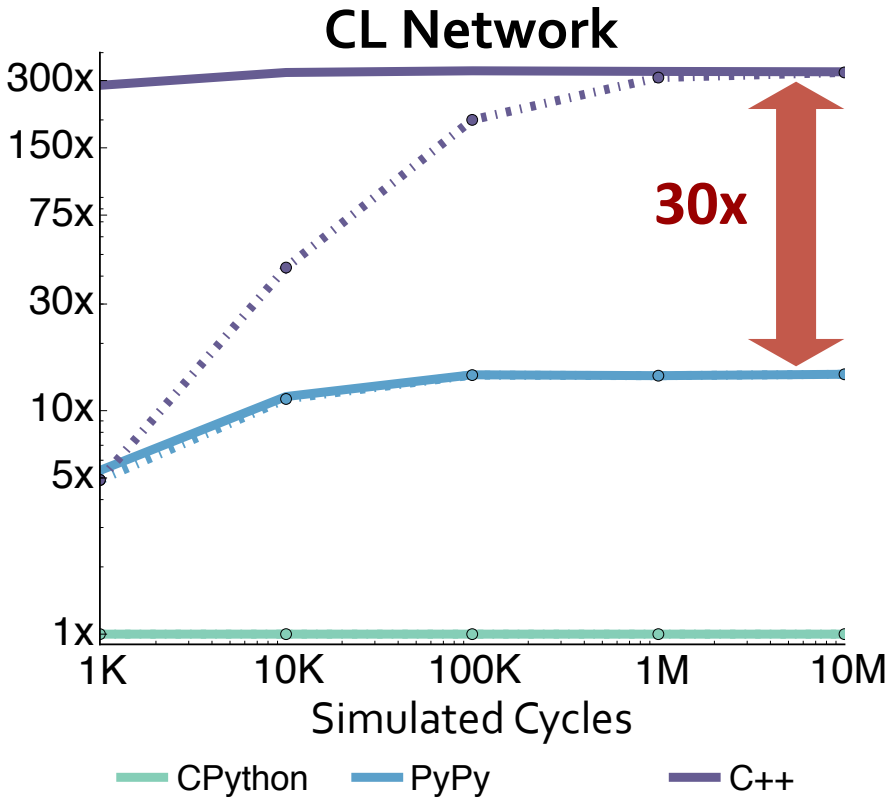
- **Python-Wrapped C/C++ Libraries**
  (NumPy, CVXOPT, NLPy, pythonOCC, GEM5)
- **Numerical Just-In-Time Compilers**
  (Numba, Parakeet)
- **Just-In-Time Compiled Interpreters**
  (PyPy, Pyston)
- **Selective Embedded Just-In-Time Specialization**
  (SEJITS)

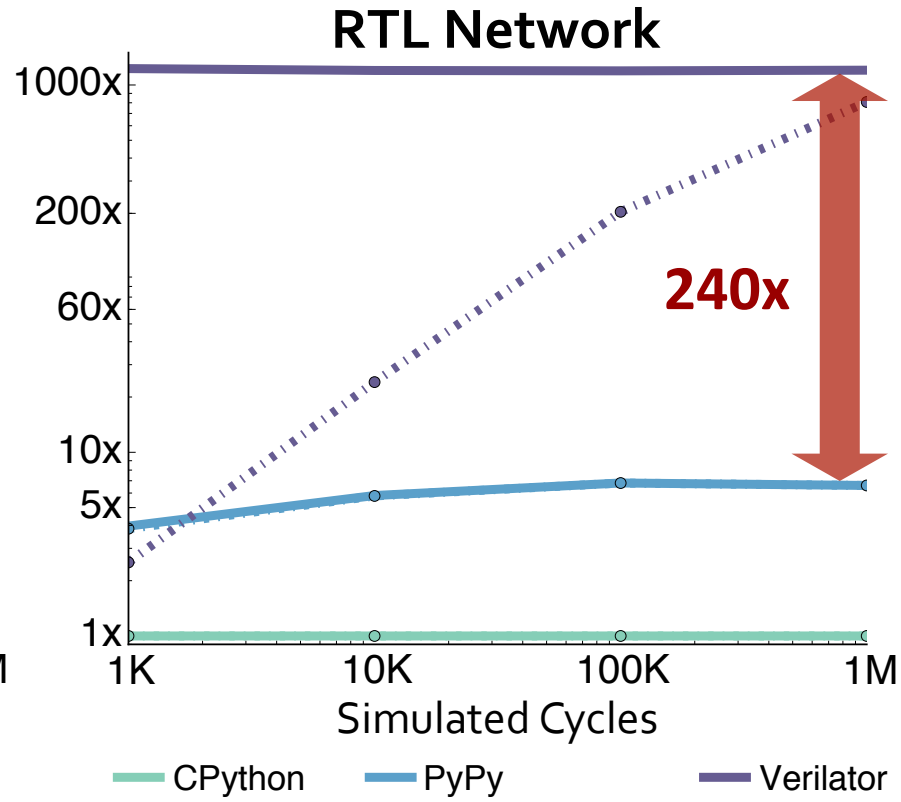
Performance-Productivity Gap





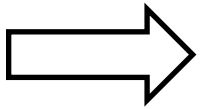
Performance-Productivity Gap





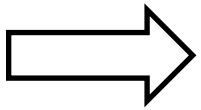
Outline

The Computer Architecture Research Methodology Gap



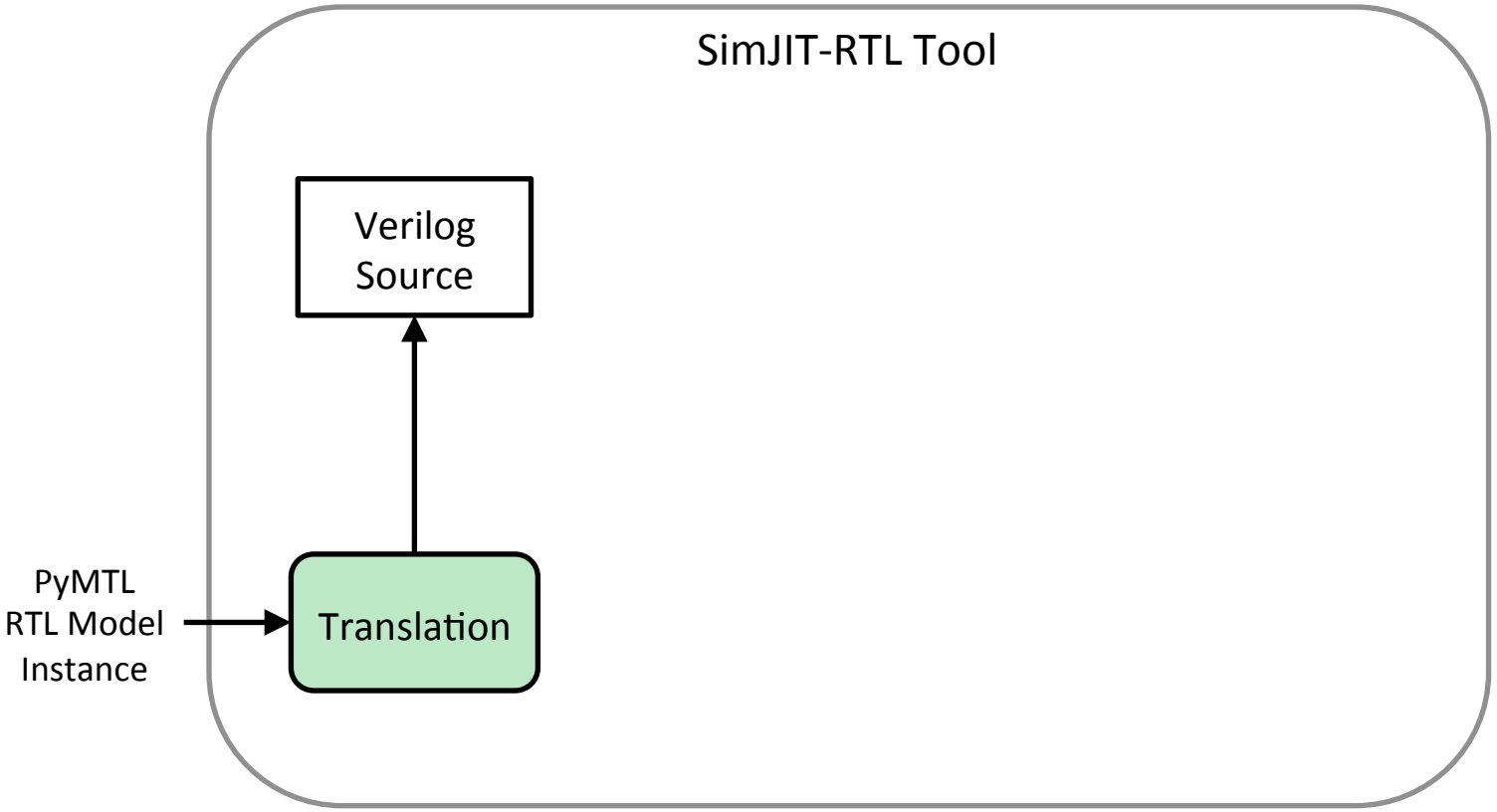
PyMTL

The Performance-Productivity Gap

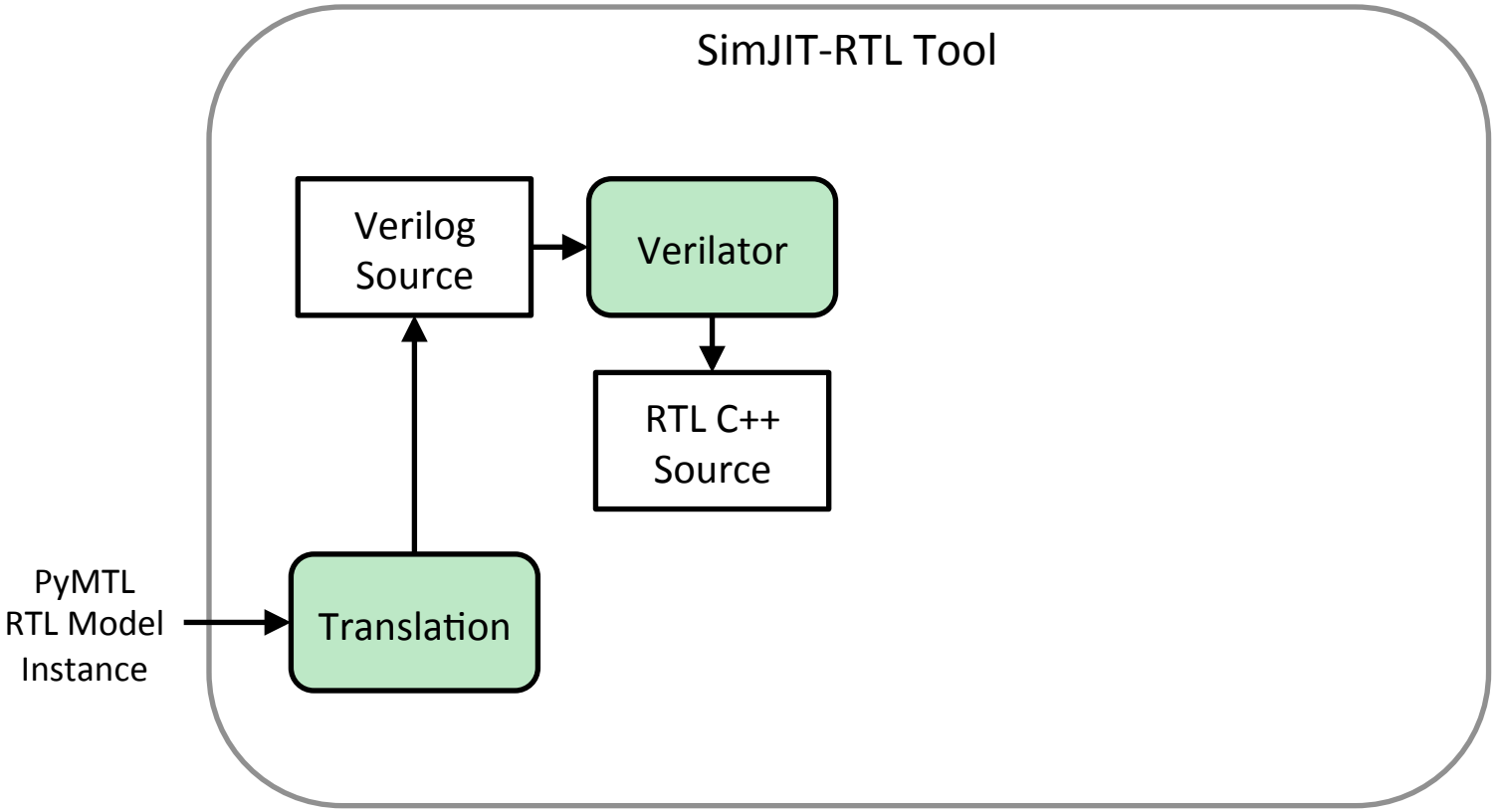


SimJIT

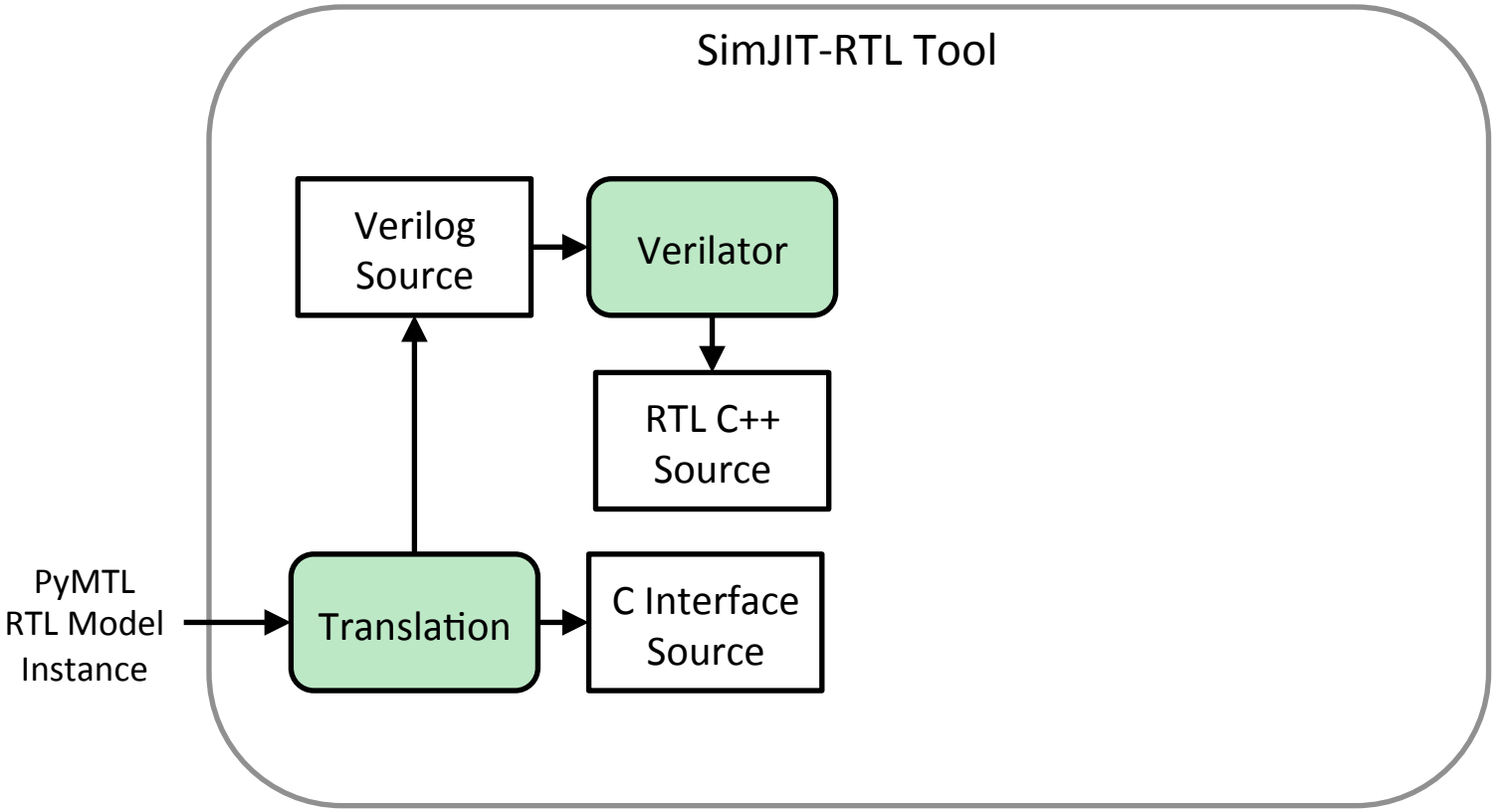
PyMTL SimJIT Architecture



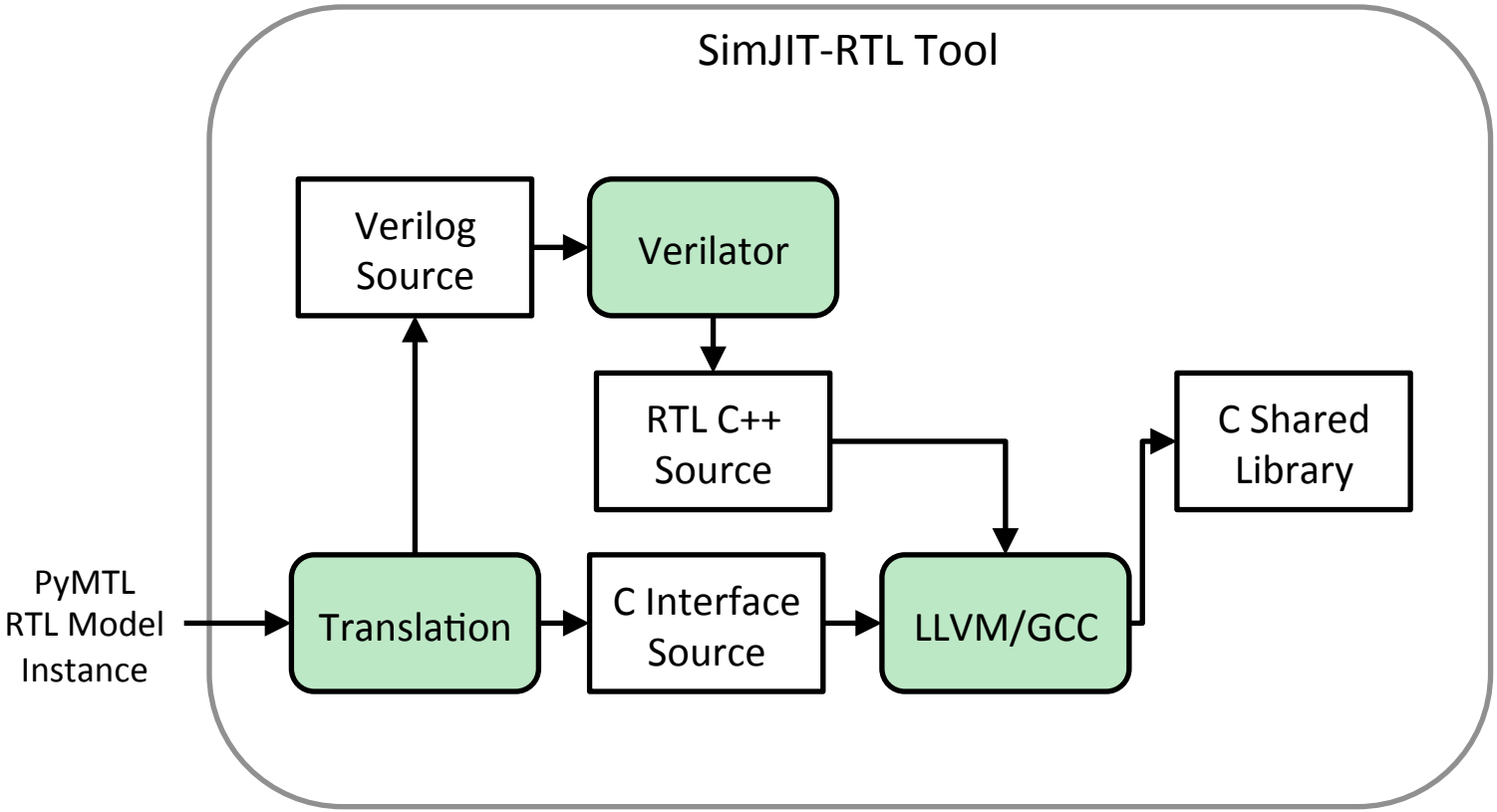
PyMTL SimJIT Architecture



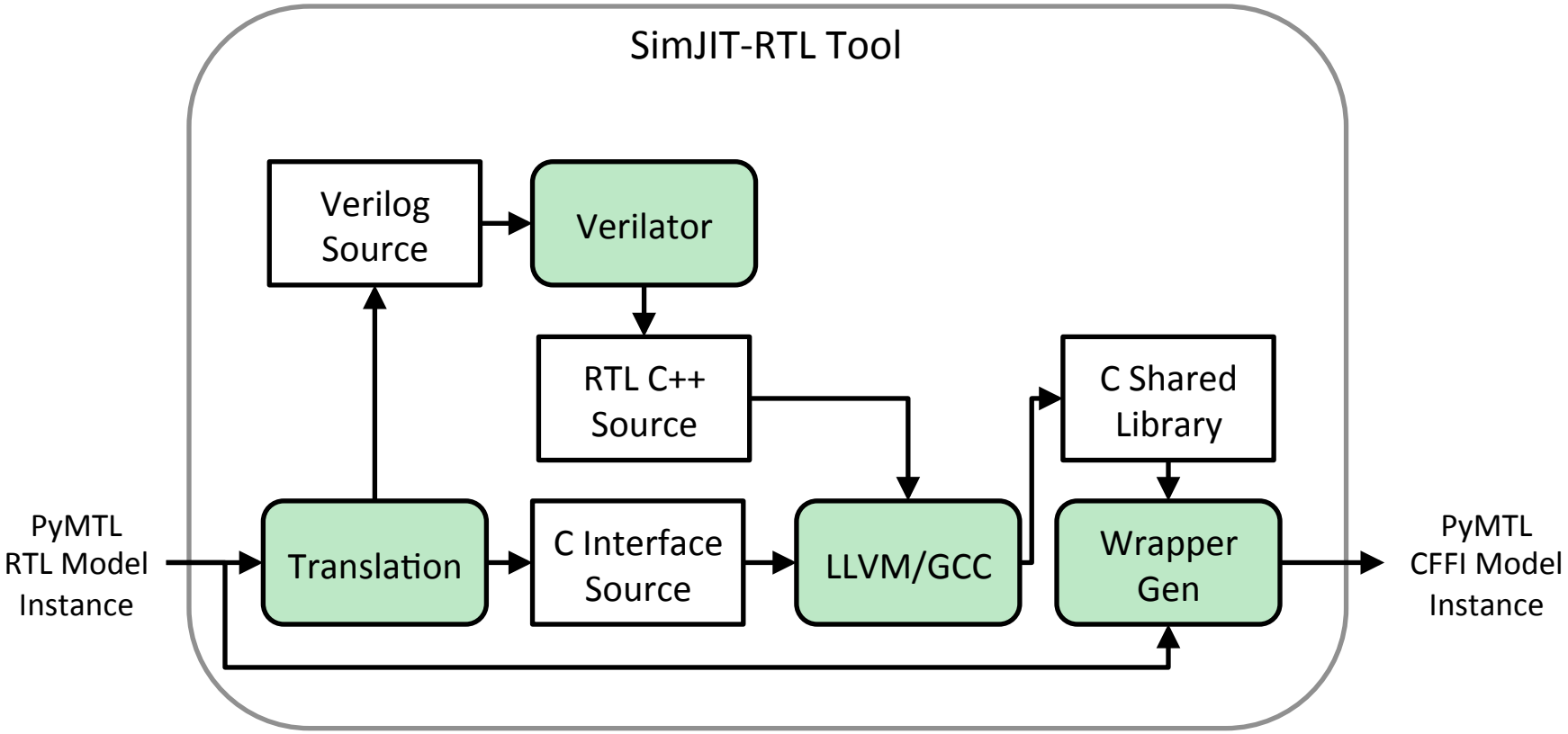
PyMTL SimJIT Architecture



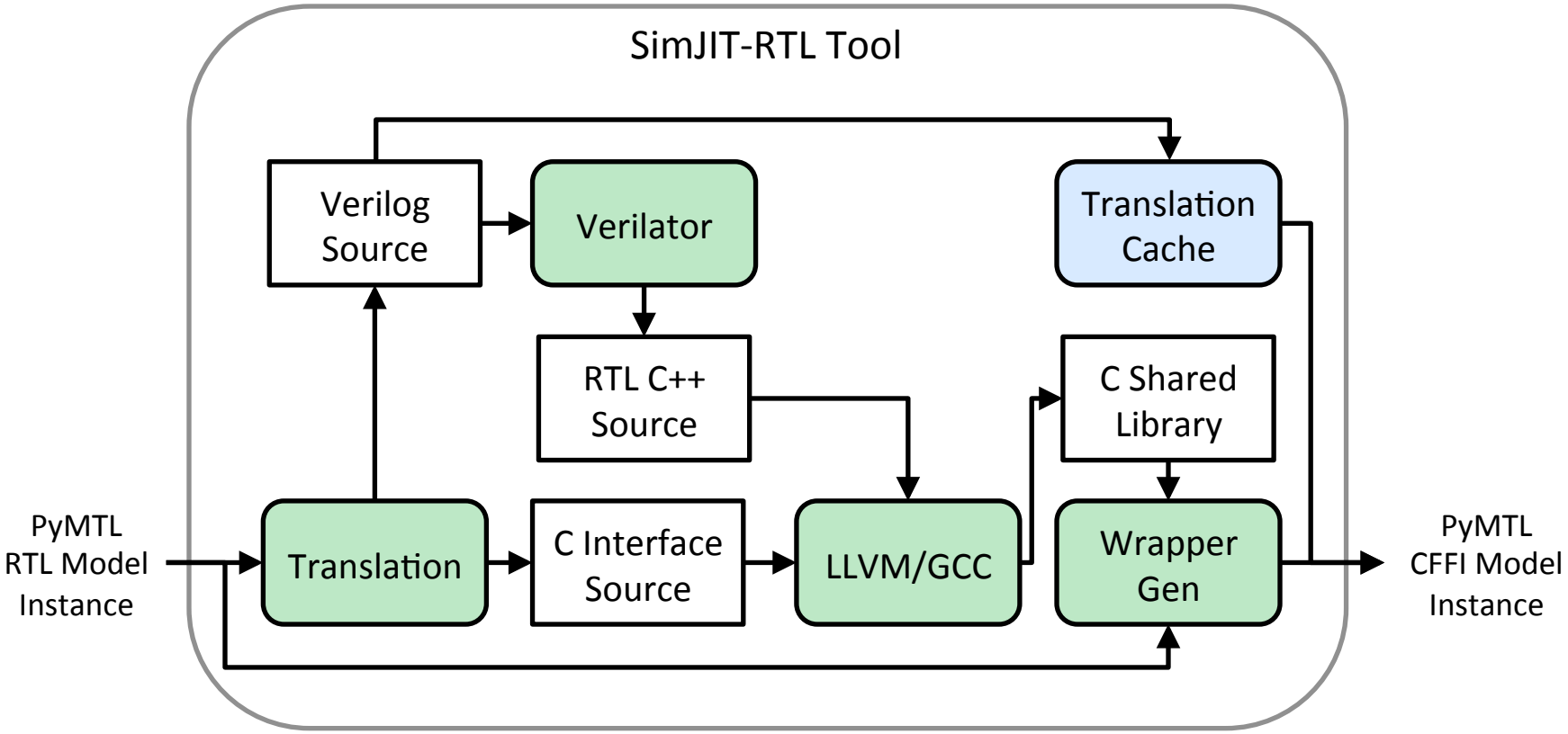
PyMTL SimJIT Architecture



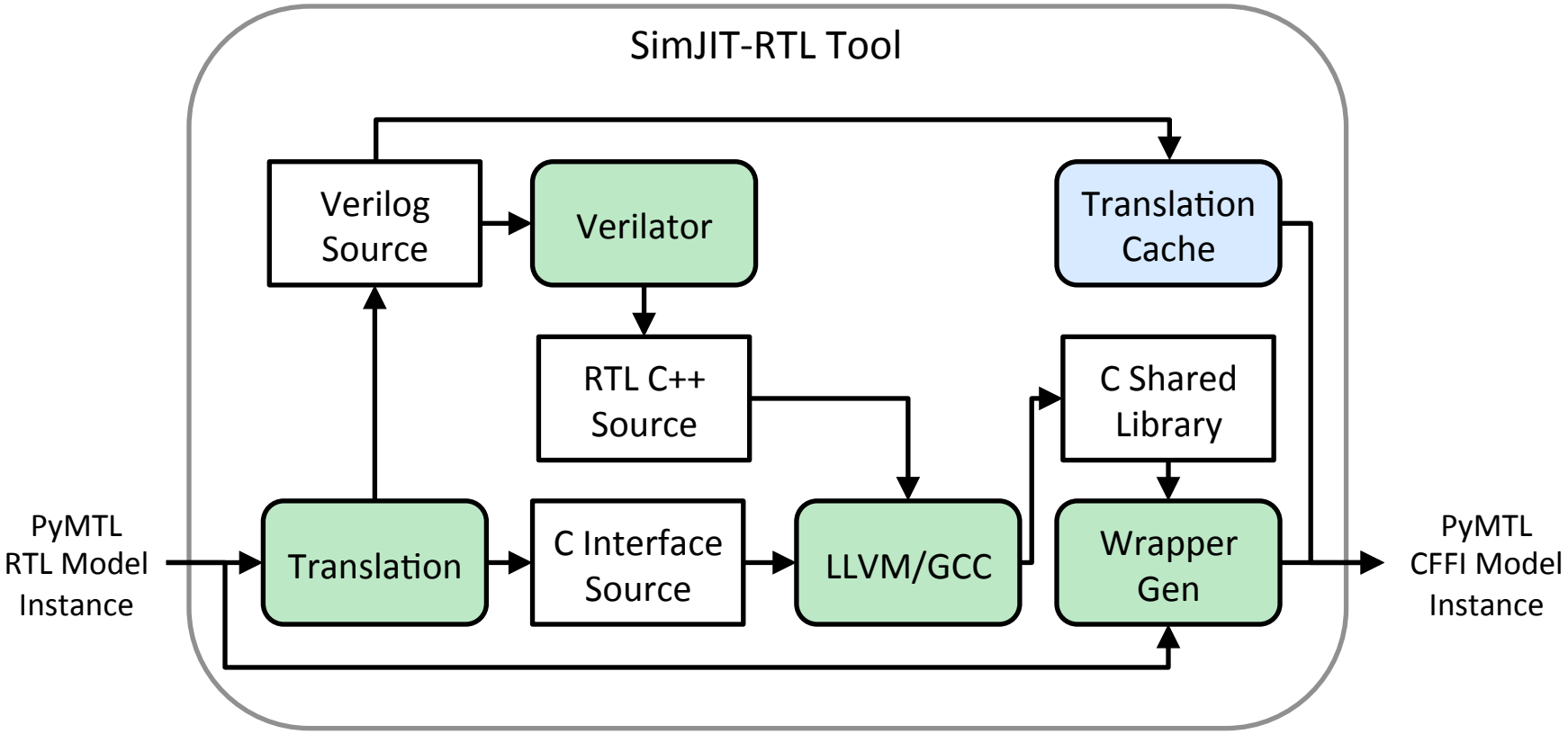
PyMTL SimJIT Architecture



PyMTL SimJIT Architecture

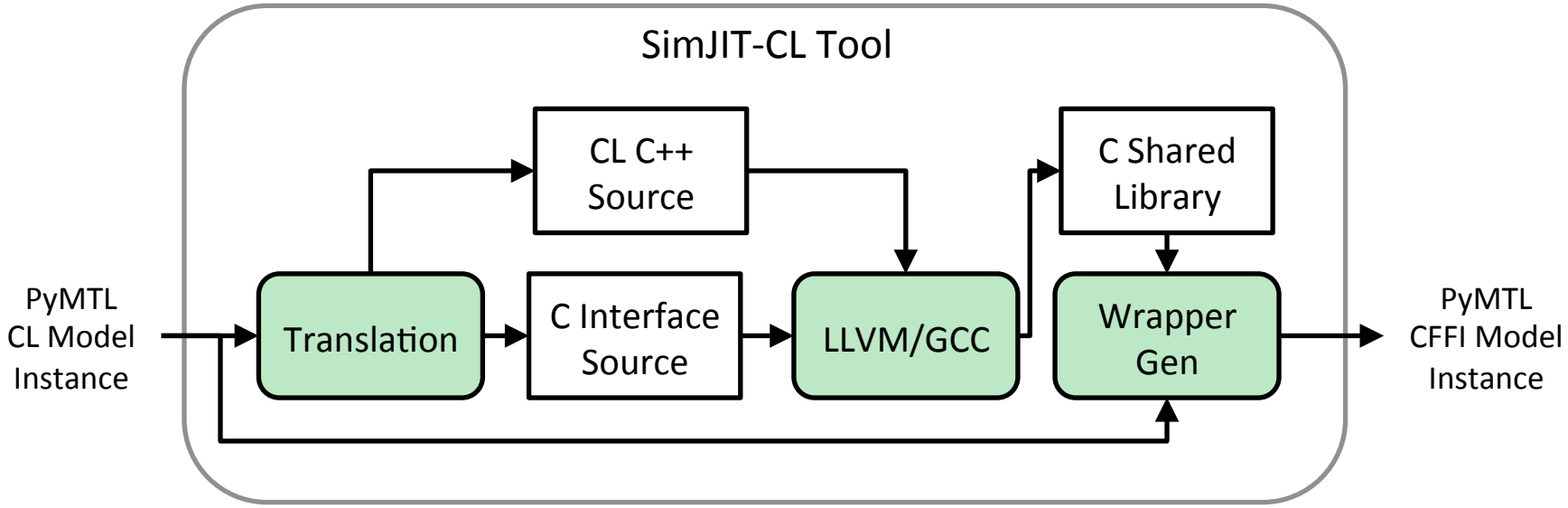


PyMTL SimJIT Architecture



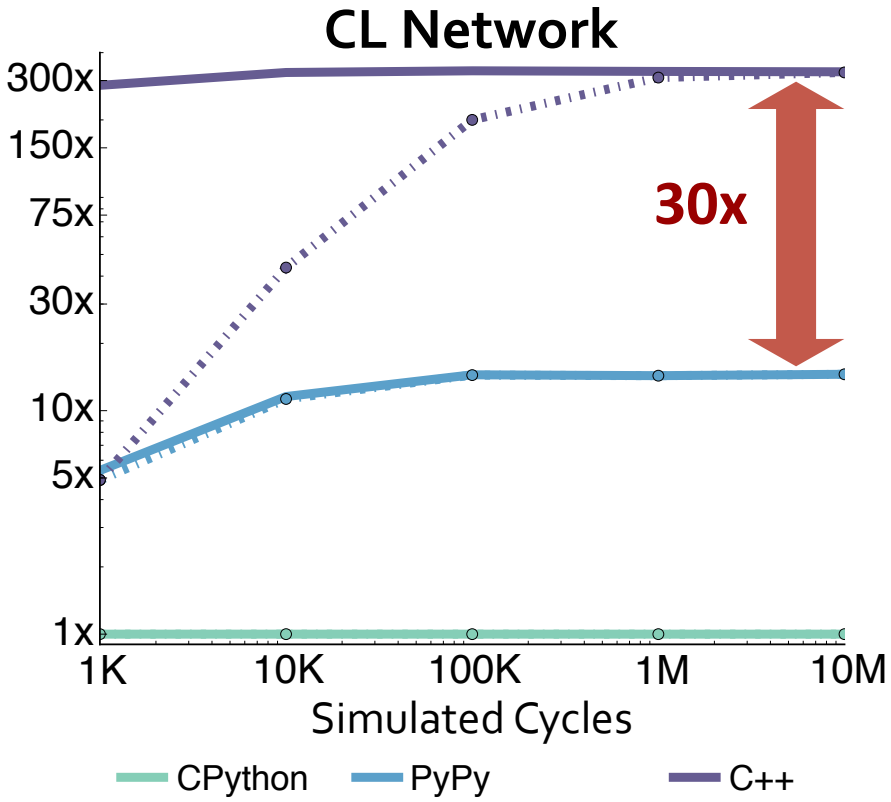
Fairly robust, ready for use in research!

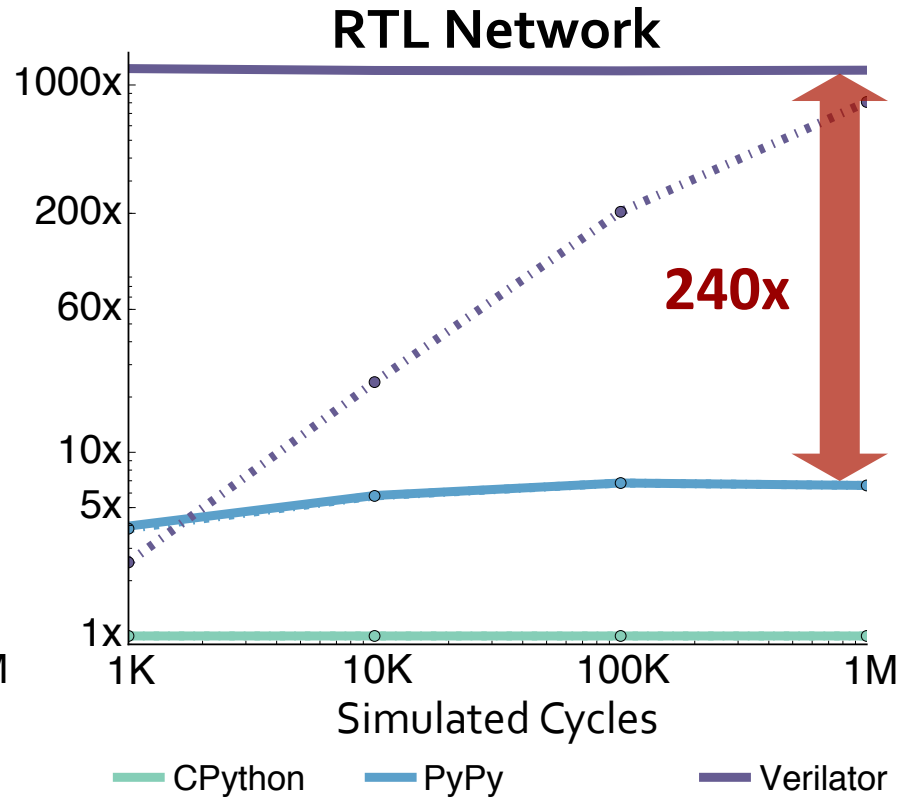
PyMTL SimJIT Architecture



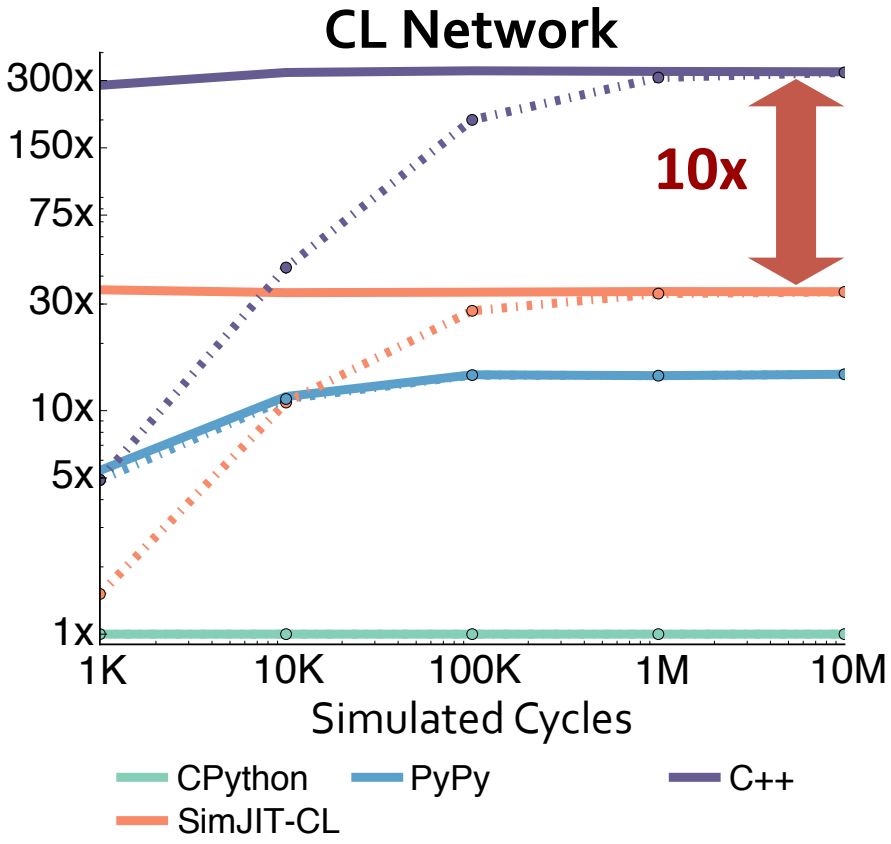
Just a prototype!

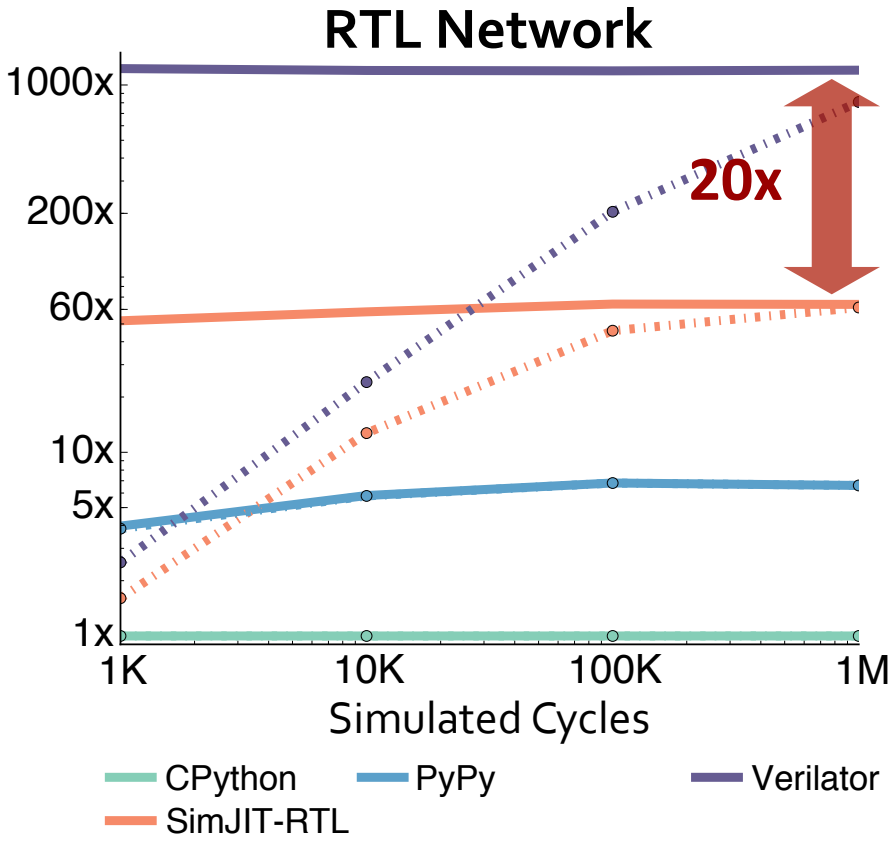
Performance-Productivity Gap



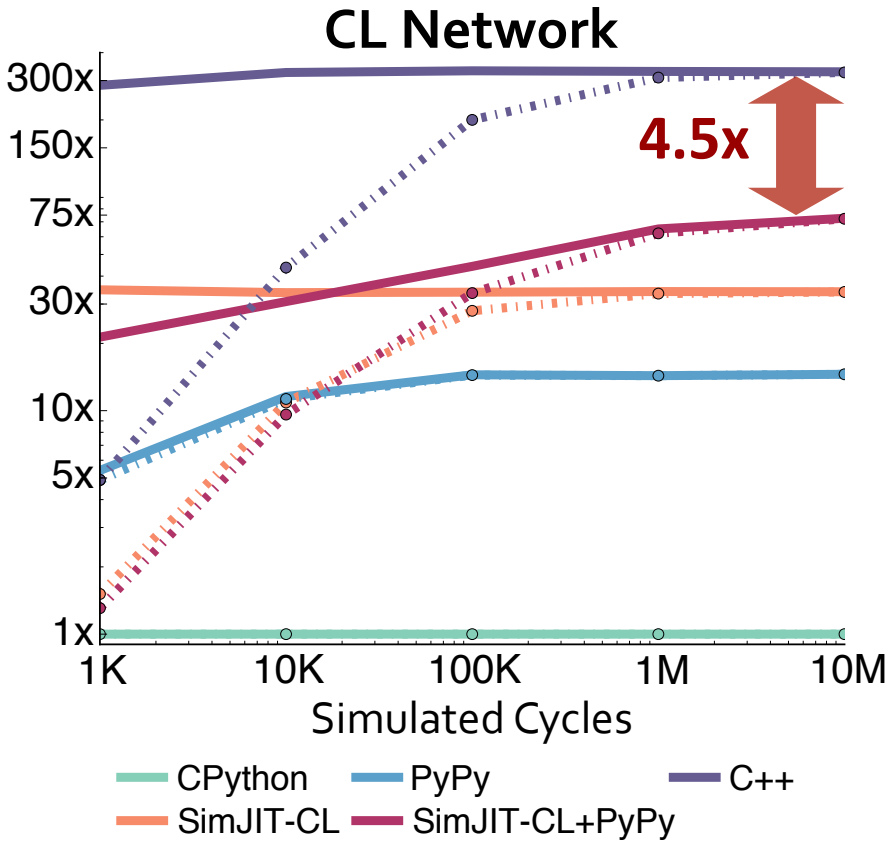


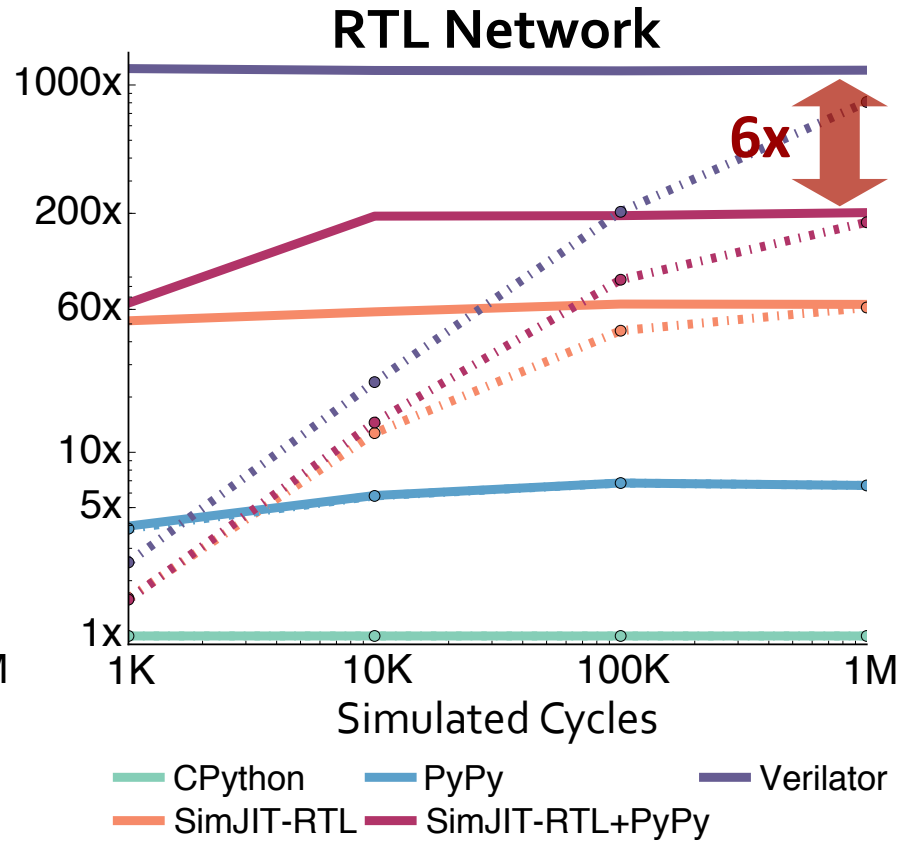
PyMTL SimJIT Performance





PyMTL SimJIT Performance





PyMTL SimJIT Performance

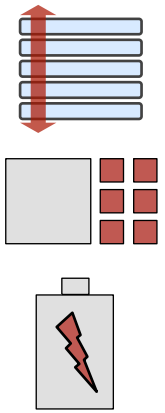
Opportunities to further reduce the performance gap:

- **Reduce overhead of Python-to-C++ interfaces**
- Optimized (non-Python) event queue
- Better code generation
- Better event queue scheduling
- Removal of unnecessary double-buffering
- Parallel simulation

Contributions

PyMTL is a productive Python framework for FL, CL, and RTL modeling, enabling:

- Vertically Integrated Computer Architecture Research
- Accelerator Design Space Exploration
- Construction of Flexible RTL Chip Generators



SimJIT considerably closes the performance-productivity gap between Python and C++ simulations.

- 72x Speedup over CPython for SimJIT-CL (within 4.5x of C++)
- 200x Speedup over CPython for SimJIT-RTL (within 6x of Verilator)

Conclusion

PyMTL is a productive, **open-source** Python framework for FL/CL/RTL modeling and hardware design.

https://github.com/cornell-brg/pymtl

Thank you to our sponsors for their support:
NSF, DARPA, and donations from Intel Corporation and Synopsys, Inc.