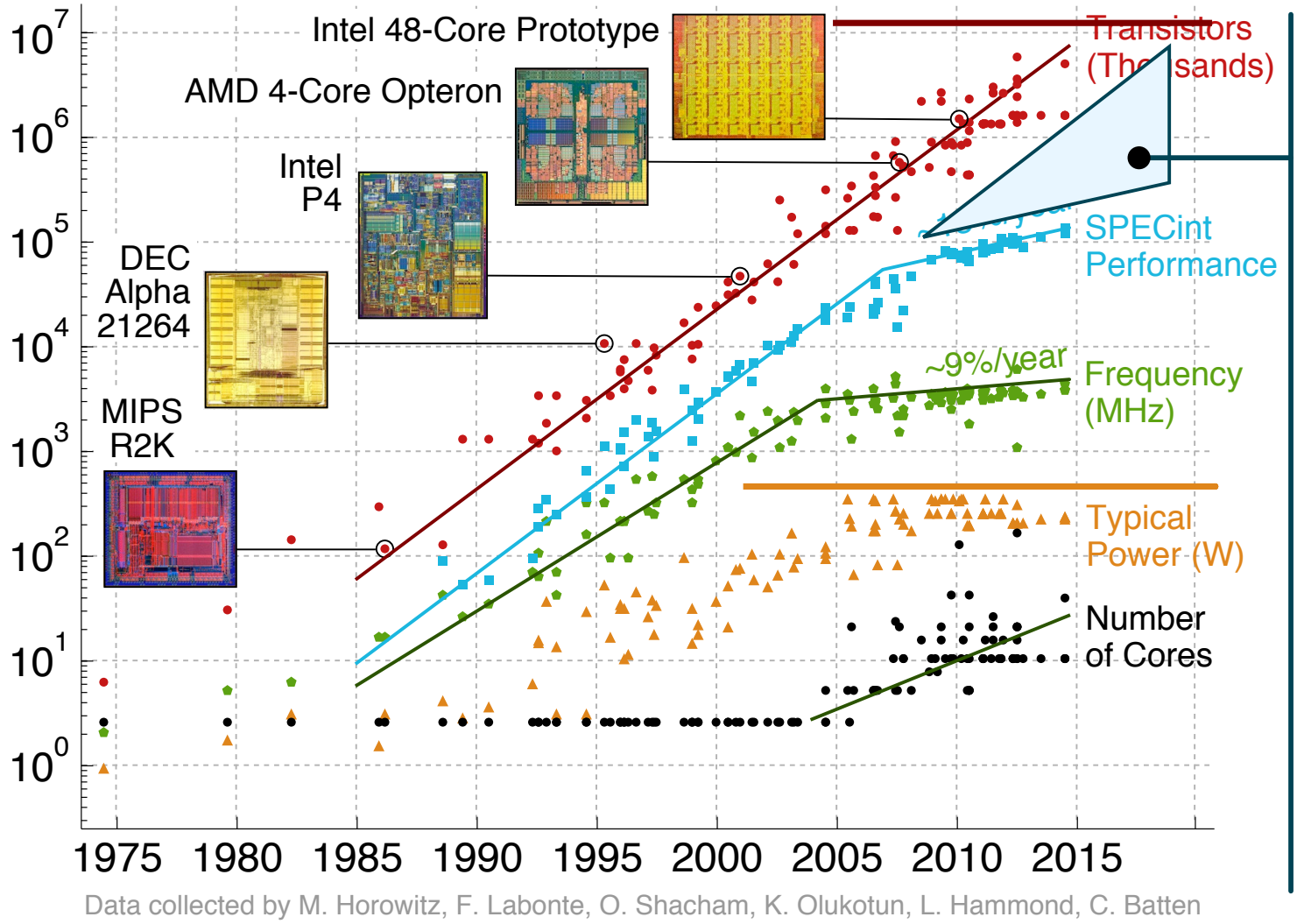
Intra-Core Loop-Task Accelerators for Task-Based Parallel Programs

Christopher Batten

Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University

Spring 2018

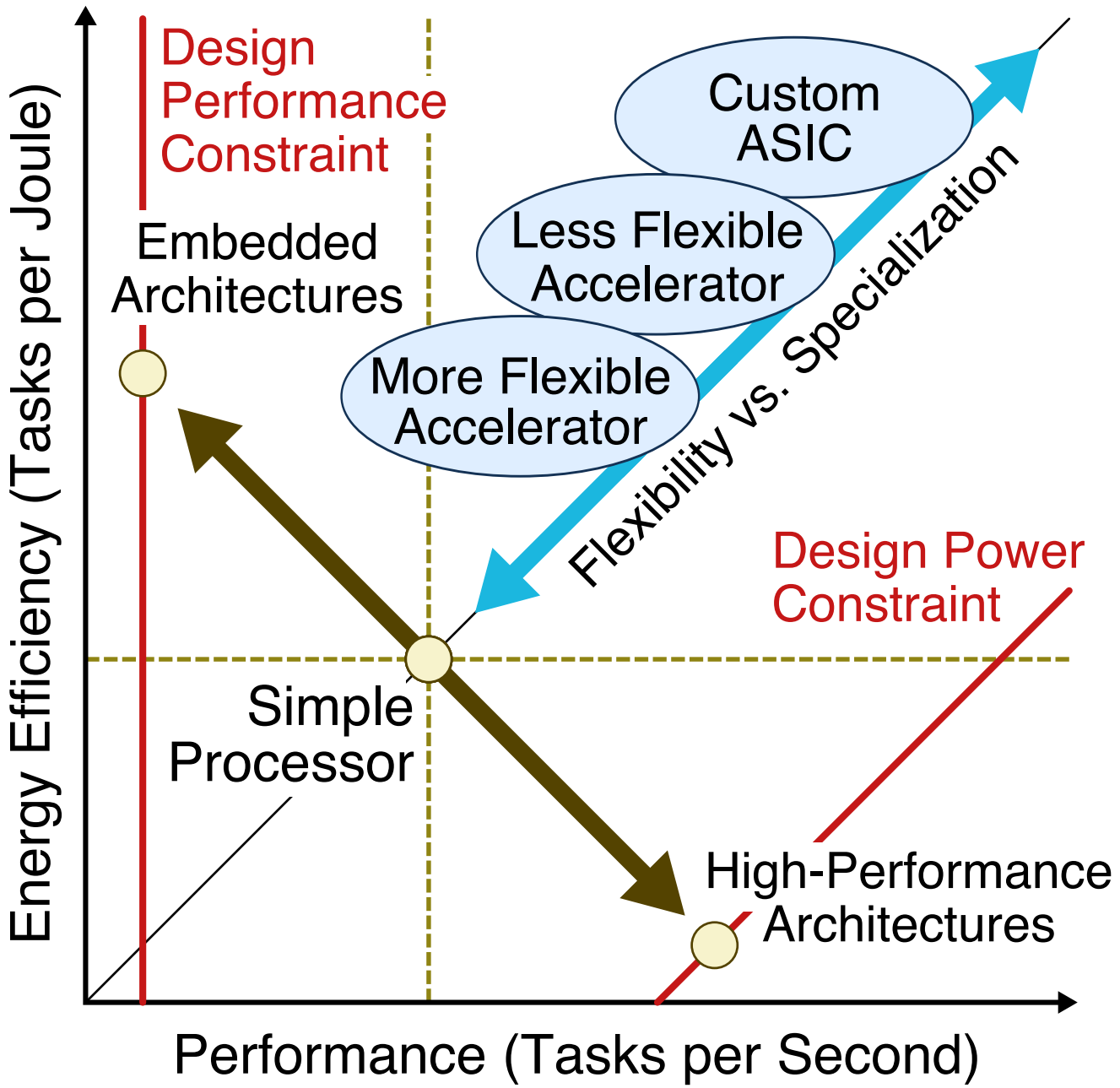
Motivating Trends in Computer Architecture



Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten

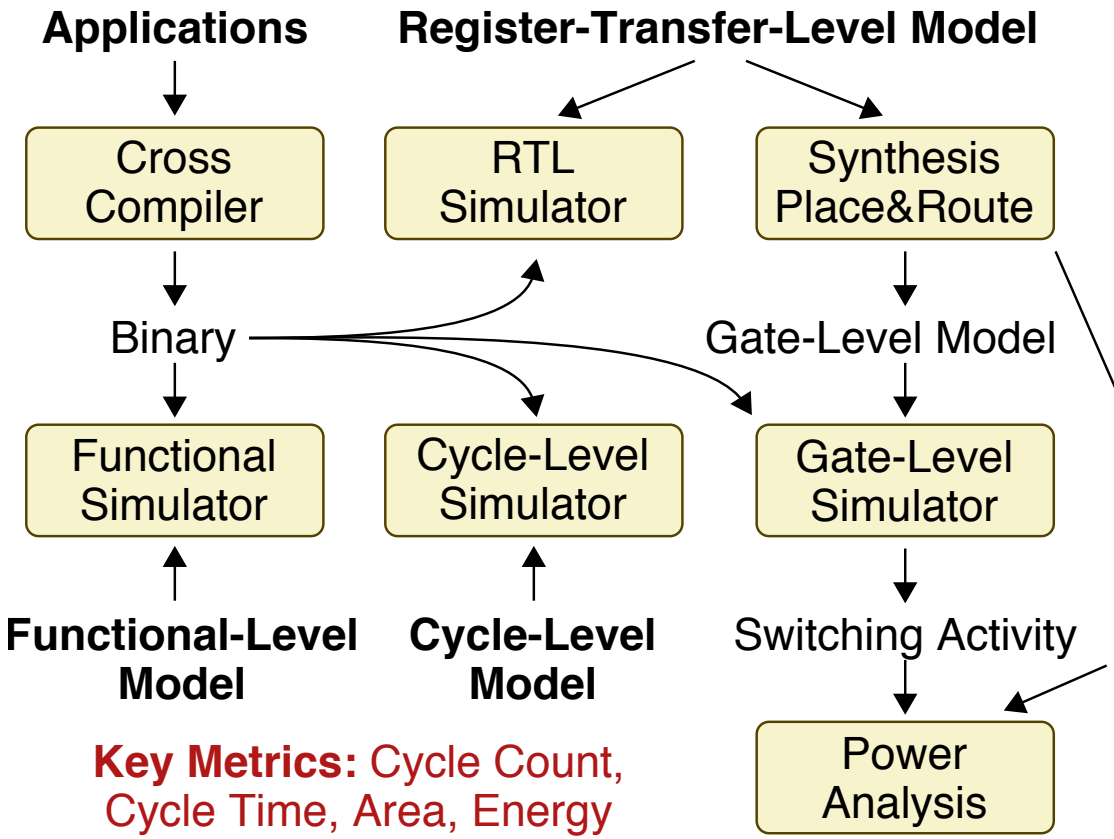
Hardware Specialization

- Data-Parallelism via GPGPUs & Vector
- Fine-Grain Task-Level Parallelism
- Instruction Set Specialization
- Subgraph Specialization
- Application-Specific Accelerators
- Domain-Specific Accelerators
- Coarse-Grain Reconfig Arrays
- Field-Programmable Gate Arrays


Design
Performance
Constraint
Custom
ASIC
Less Flexible
Accelerator
More Flexible
Accelerator
Flexibility vs. Specialization
Embedded
Architectures
Design Power
Constraint
Simple
Processor
High-Performance
Architectures
Energy Efficiency (Tasks per Joule)
Performance (Tasks per Second)

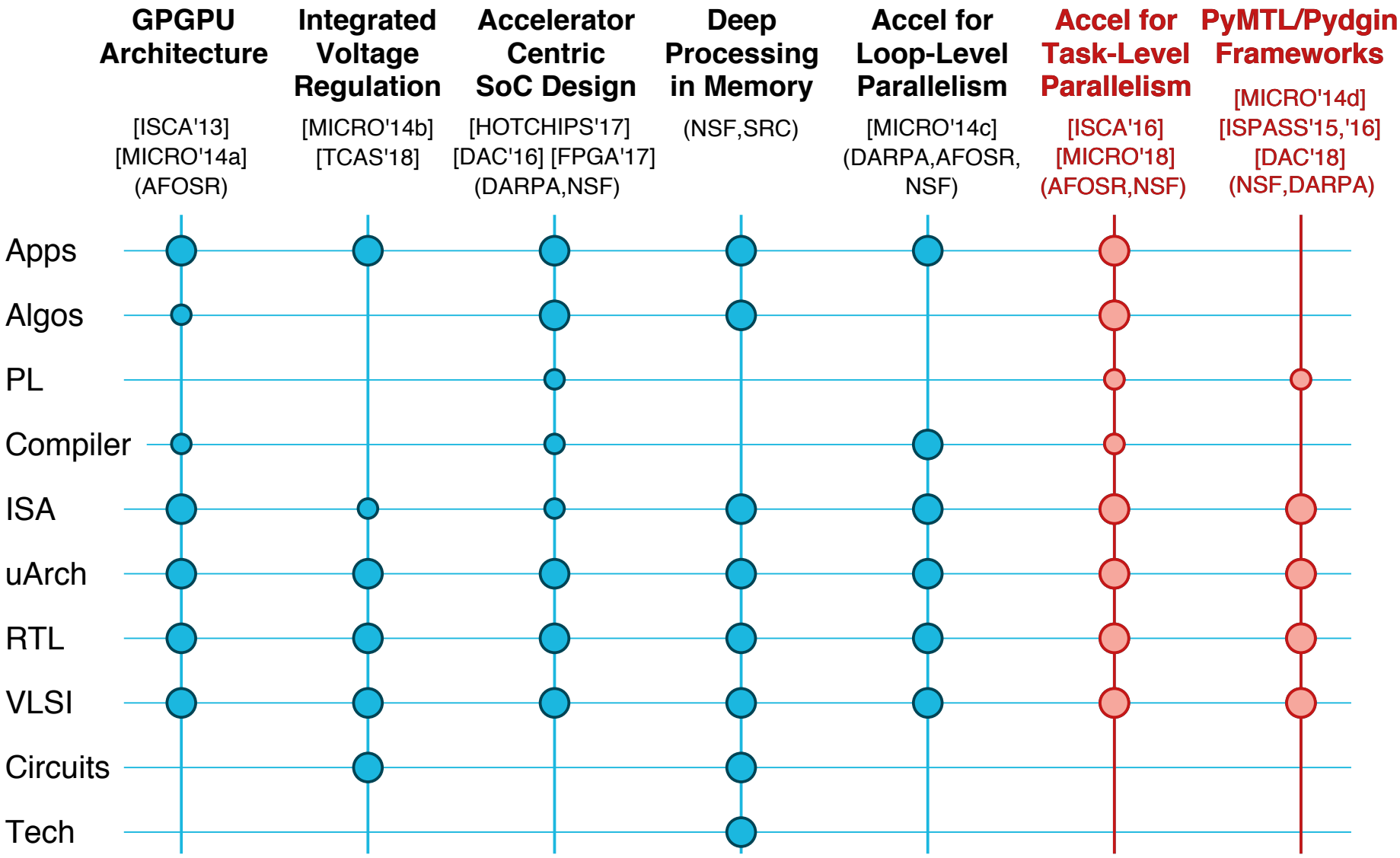
Vertically Integrated Research Methodology

Our research involves reconsidering all aspects of the computing stack including applications, programming frameworks, compiler optimizations, runtime systems, instruction set design, microarchitecture design, VLSI implementation, and hardware design methodologies



Experimenting with full-chip layout, FPGA prototypes, and test chips is a key part of our research methodology

Projects Within the Batten Research Group



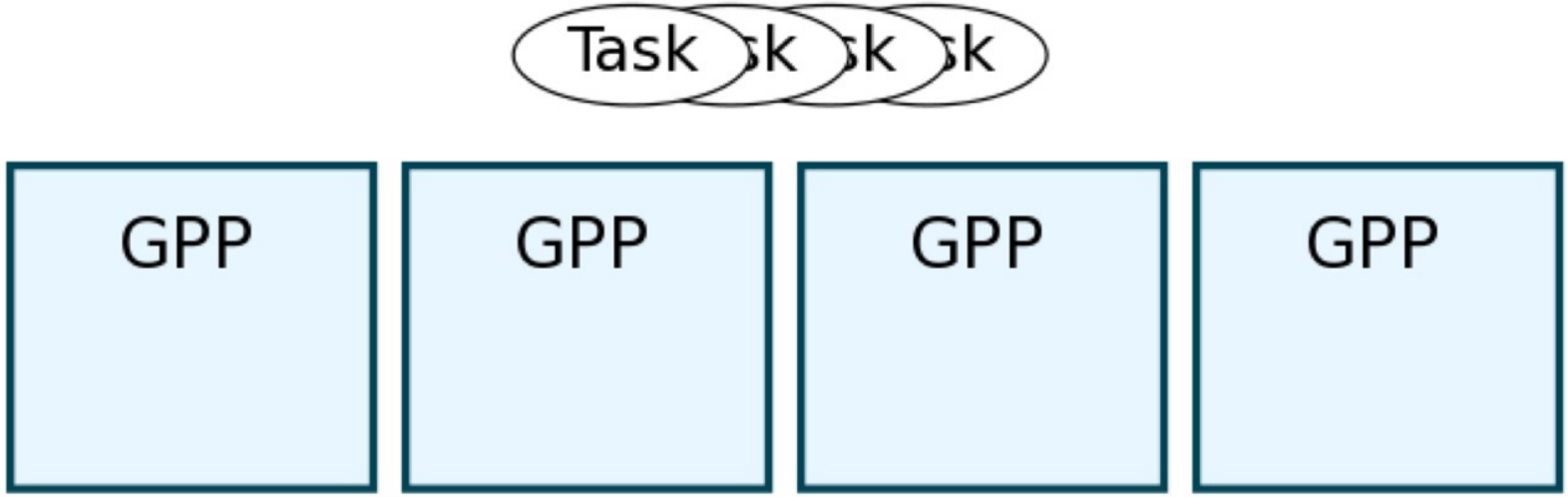
Using Intra-Core Loop-Task Accelerators to Improve the Productivity and Performance of Task-Based Parallel Programs

Ji Kim, Shunning Jiang, Christopher Torng, Moyang Wang
Shreesha Srinath, Berkin Ilbeyi, Khalid Al-Hawaj
Christopher Batten

50th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)
Cambridge, MA, Oct. 2017

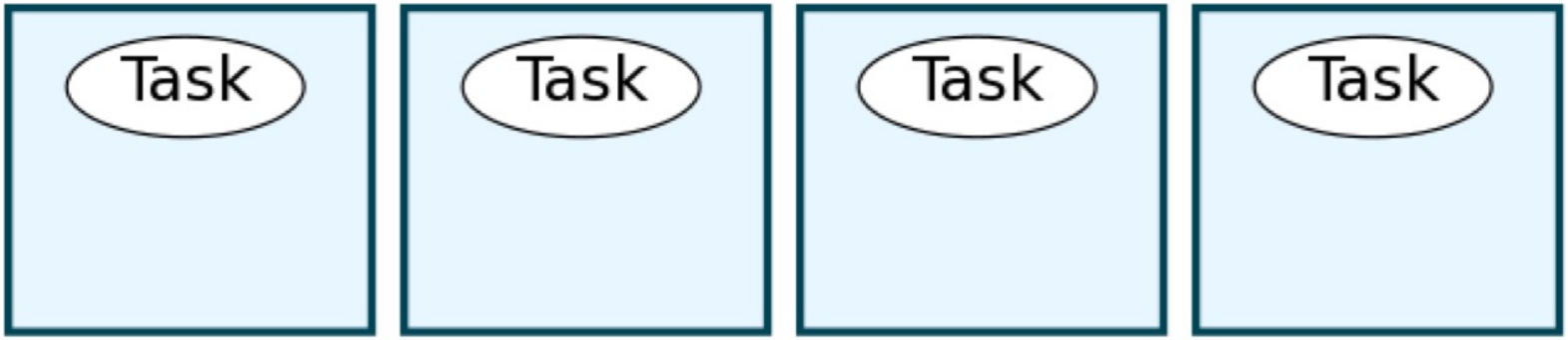
Inter-Core

- Task-Based Parallel Programming Frameworks
  - Intel TBB, Cilk



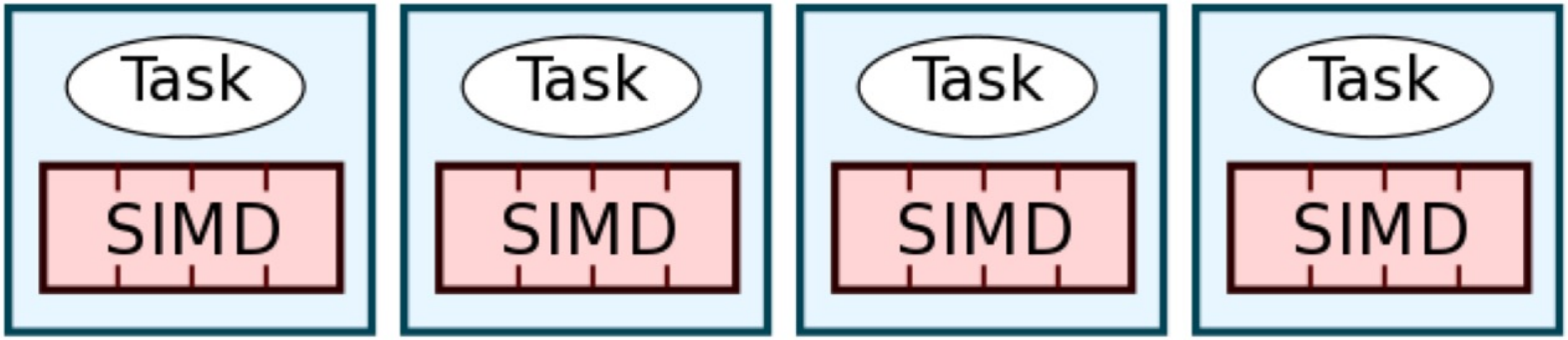
Inter-Core

- Task-Based Parallel Programming Frameworks
  - Intel TBB, Cilk



Inter-Core

- Task-Based Parallel Programming Frameworks
  - Intel TBB, Cilk



Intra-Core

- Packed-SIMD Vectorization
  - Intel AVX, Arm NEON

Challenges of Combining Tasks and Vectors

```
void app_kernel_tbb_avx(int N, float* src, float* dst) {
  // Pack data into padded aligned chunks
  //   src -> src_chunks[NUM_CHUNKS * SIMD_WIDTH]
  //   dst -> dst_chunks[NUM_CHUNKS * SIMD_WIDTH]
  ...

  // Use TBB across cores
  parallel_for (range(0, NUM_CHUNKS, TASK_SIZE), [&] (range r) {
    for (int i = r.begin(); i < r.end(); i++) {
      // Use packed-SIMD within a core
      #pragma simd vlen(SIMD_WIDTH)
      for (int j = 0; j < SIMD_WIDTH; j++) {
        if (src_chunks[i][j] > THRESHOLD)
          aligned_dst[i] = DoLightCompute aligned_src[i]);
        else
          aligned_dst[i] = DoHeavyCompute aligned_src[i]);
  ...
  ...
```

Challenge #1: Intra-Core Parallel Abstraction Gap

Challenges of Combining Tasks and Vectors

```
void app_kernel_tbb_avx(int N, float* src, float* dst) {
  // Pack data into padded aligned chunks
  //   src -> src_chunks[NUM_CHUNKS * SIMD_WIDTH]
  //   dst -> dst_chunks[NUM_CHUNKS * SIMD_WIDTH]
  ...

  // Use TBB across cores
  parallel_for (range(0, NUM_CHUNKS, TASK_SIZE), [&] (range r) {
    for (int i = r.begin(); i < r.end(); i++) {
      // Use packed-SIMD within a core
      #pragma simd vlen(SIMD_WIDTH)
      for (int j = 0; j < SIMD_WIDTH; j++) {
        if (src_chunks[i][j] > THRESHOLD)
          aligned_dst[i] = DoLightCompute(aligned_src[i]);
        else
          aligned_dst[i] = DoHeavyCompute(aligned_src[i]);
  ...
  ...
```

Challenge #1: Intra-Core Parallel Abstraction Gap

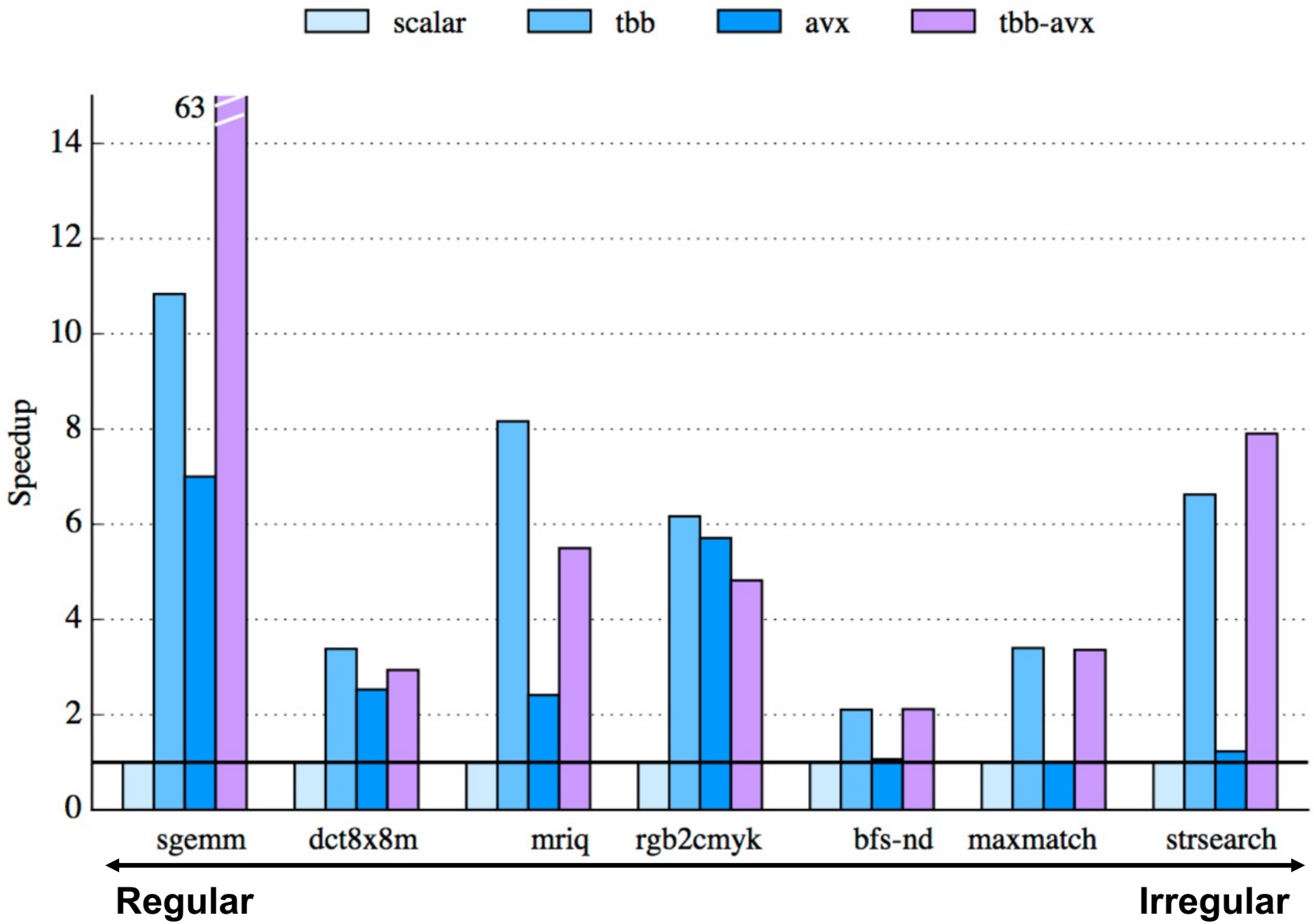
Challenges of Combining Tasks and Vectors

```
void app_kernel_tbb_avx(int N, float* src, float* dst) {
  // Pack data into padded aligned chunks
  //   src -> src_chunks[NUM_CHUNKS * SIMD_WIDTH]
  //   dst -> dst_chunks[NUM_CHUNKS * SIMD_WIDTH]
  ...

  // Use TBB across cores
  parallel_for (range(0, NUM_CHUNKS, TASK_SIZE), [&] (range r) {
    for (int i = r.begin(); i < r.end(); i++) {
      // Use packed-SIMD within a core
      #pragma simd vlen(SIMD_WIDTH)
      for (int j = 0; j < SIMD_WIDTH; j++) {
        if (src_chunks[i][j] > THRESHOLD)
          aligned_dst[i] = DoLightCompute(aligned_src[i]);
        else
          aligned_dst[i] = DoHeavyCompute(aligned_src[i]);
  ...
  ...
```

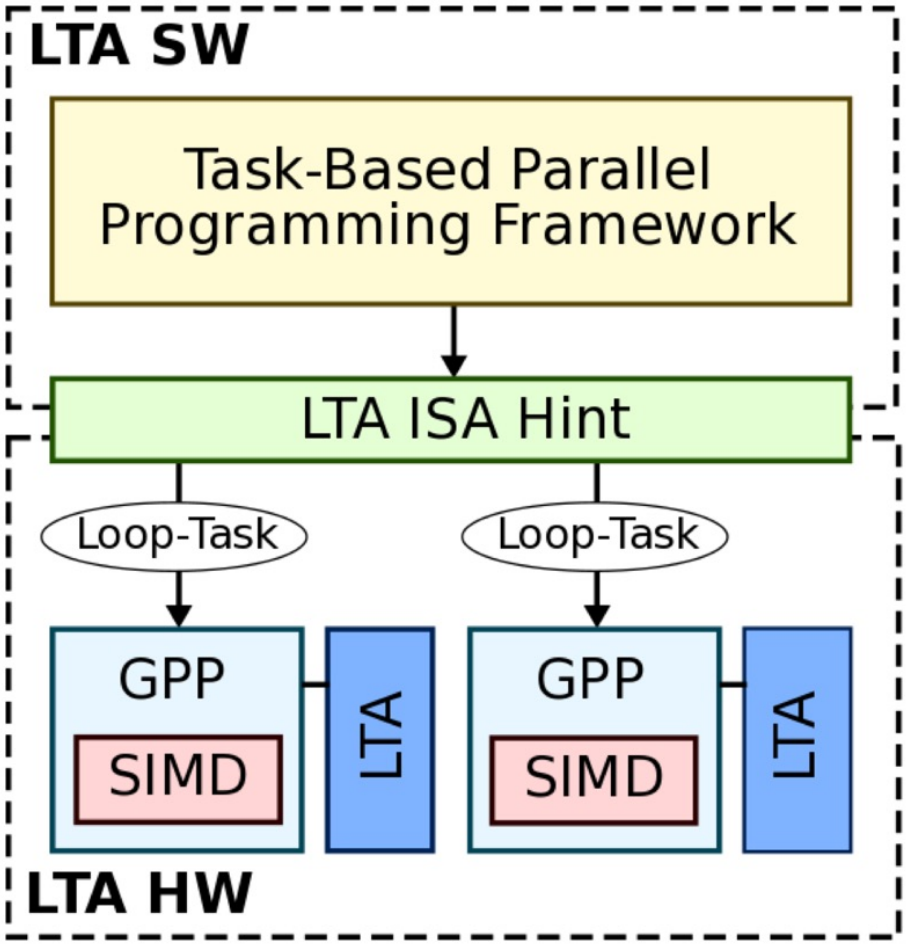
Challenge #1: Intra-Core Parallel Abstraction Gap

Challenge #2: Inefficient Execution of Irregular Tasks

Native Performance Results



Loop-Task Accelerator (LTA) Vision

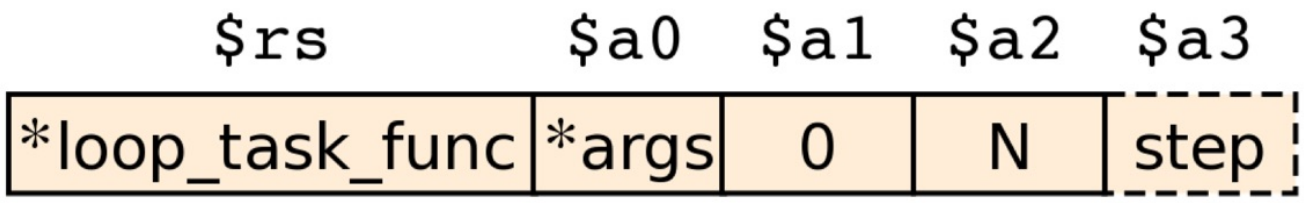


- Motivation
- **Challenge #1: LTA SW**
- Challenge #2: LTA HW
- Evaluation

LTA SW: API and ISA Hint

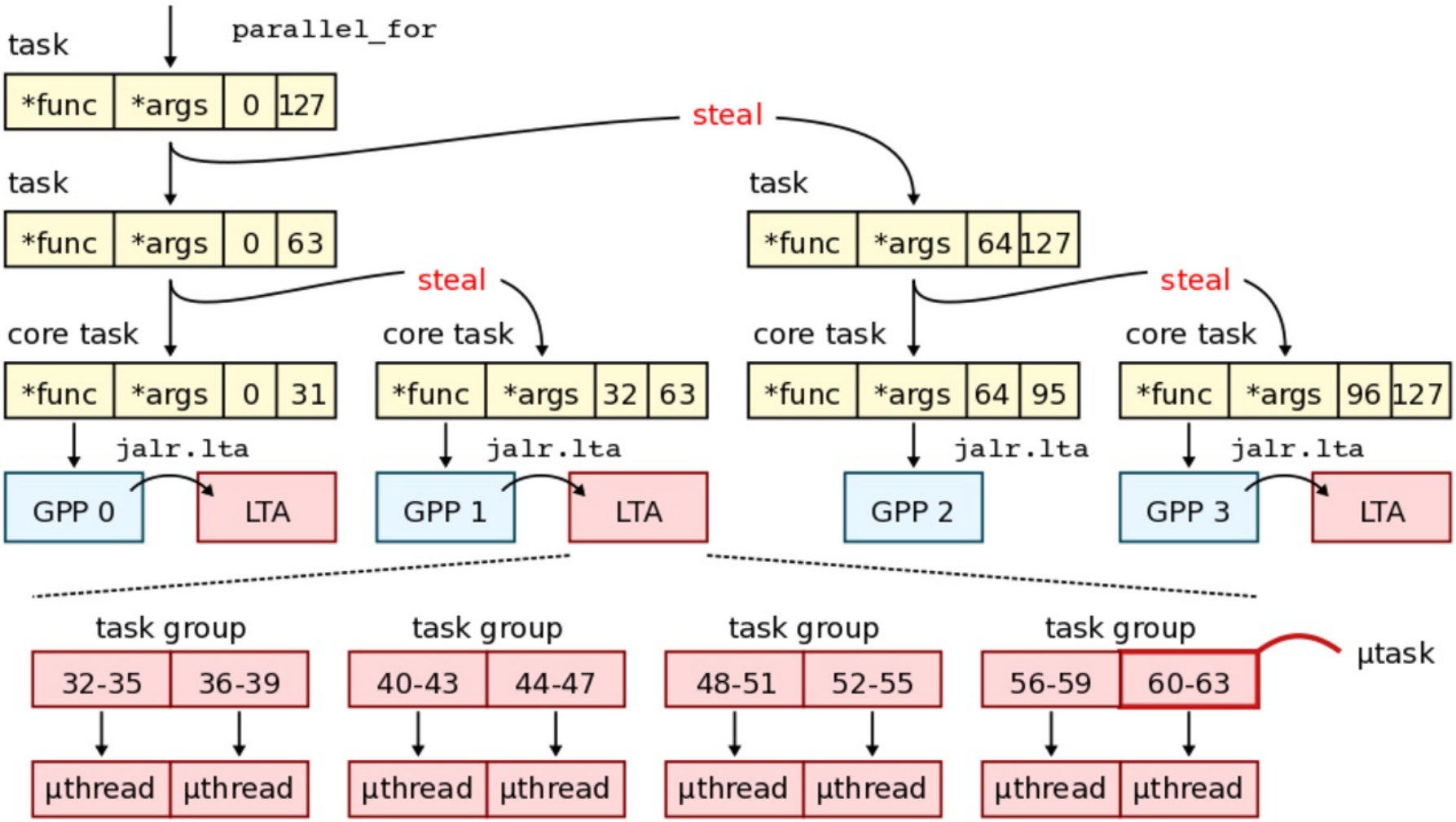
```
void app_kernel_lta(int N, float* src, float* dst) {
  LTA_PARALLEL_FOR(0, N, (dst, src), ({
    if (src[i] > THRESHOLD)
      dst[i] = DoComputeLight(src[i]);
    else
      dst[i] = DoComputeHeavy(src[i]);
  }));
}

void loop_task_func(void* a, int start, int end, int step=1);
```

```
jalr.lta $rd, $rs
```

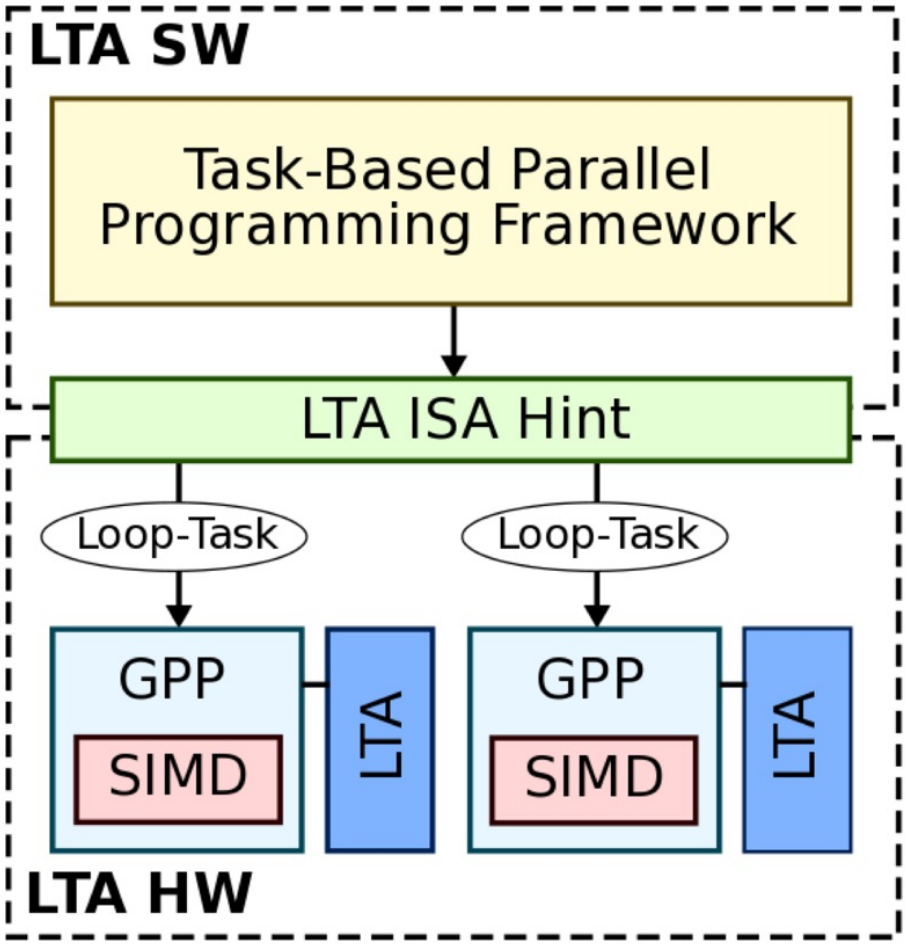


Hint that hardware can potentially accelerate task execution

LTA SW: Task-Based Runtime

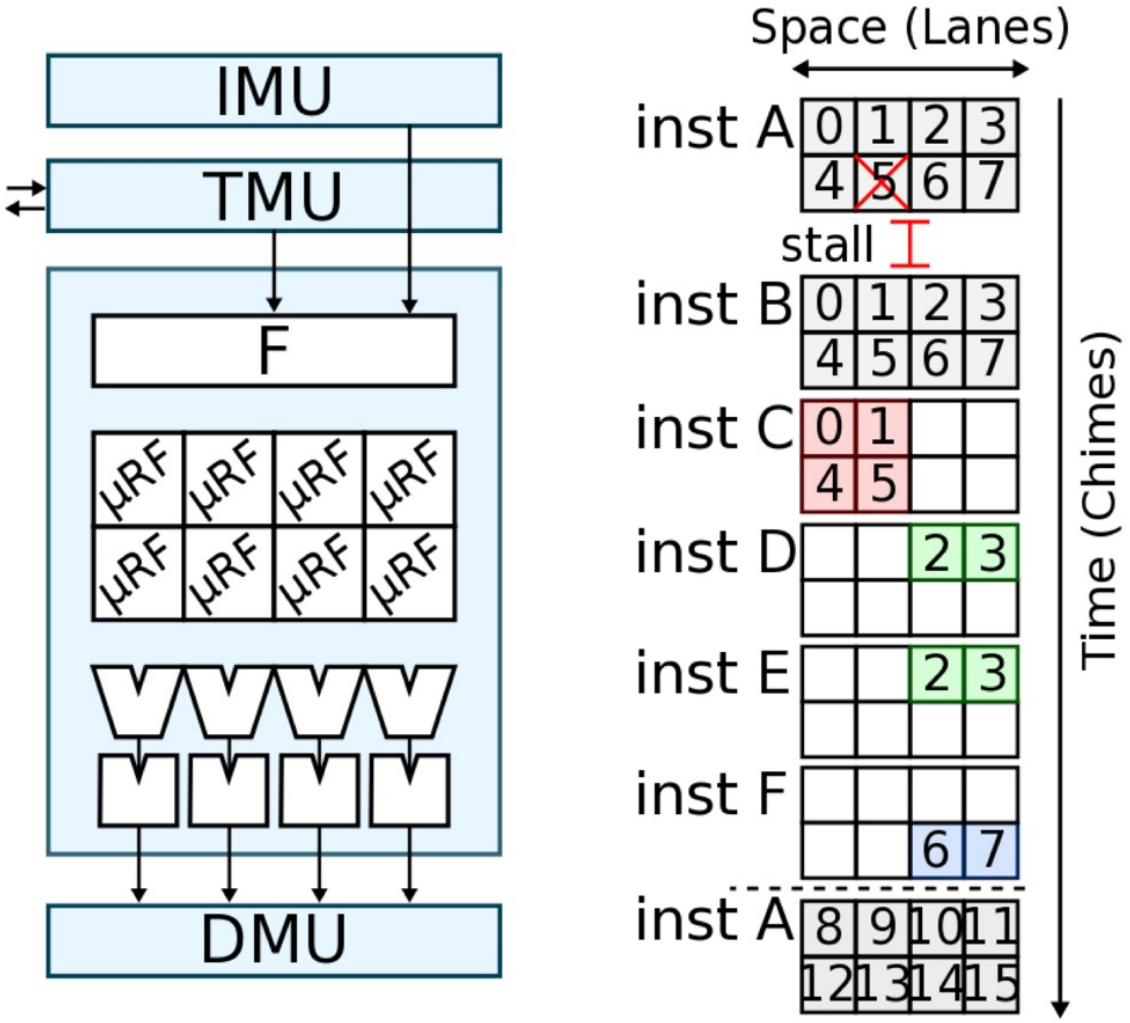


Loop-Task Accelerator (LTA) Vision



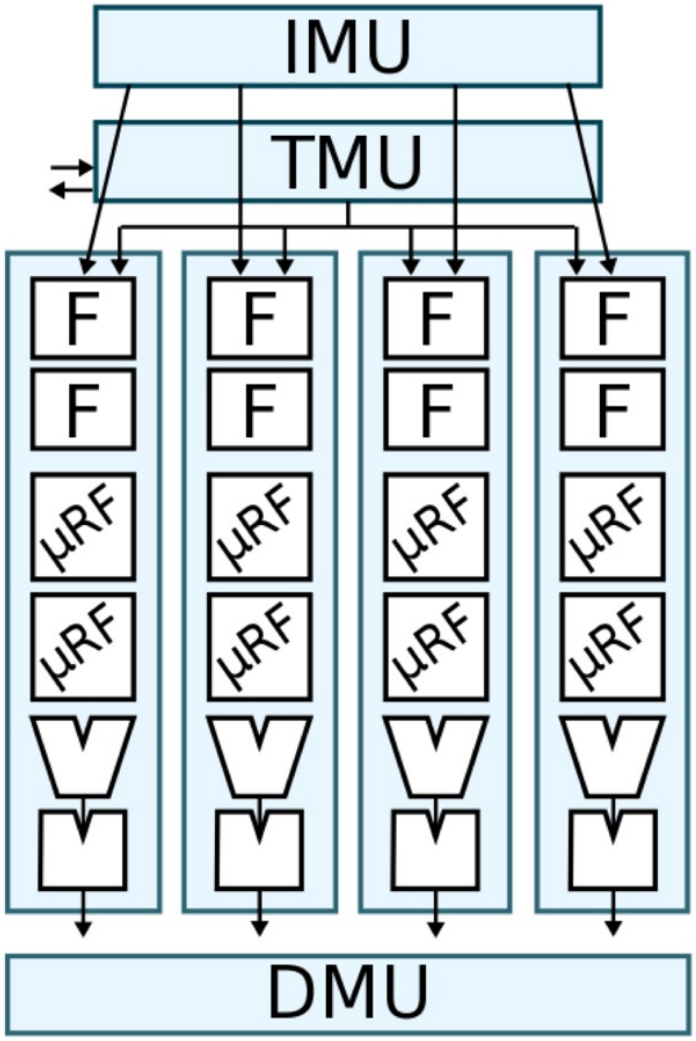
- Motivation
- Challenge #1: LTA SW
- **Challenge #2: LTA HW**
- Evaluation

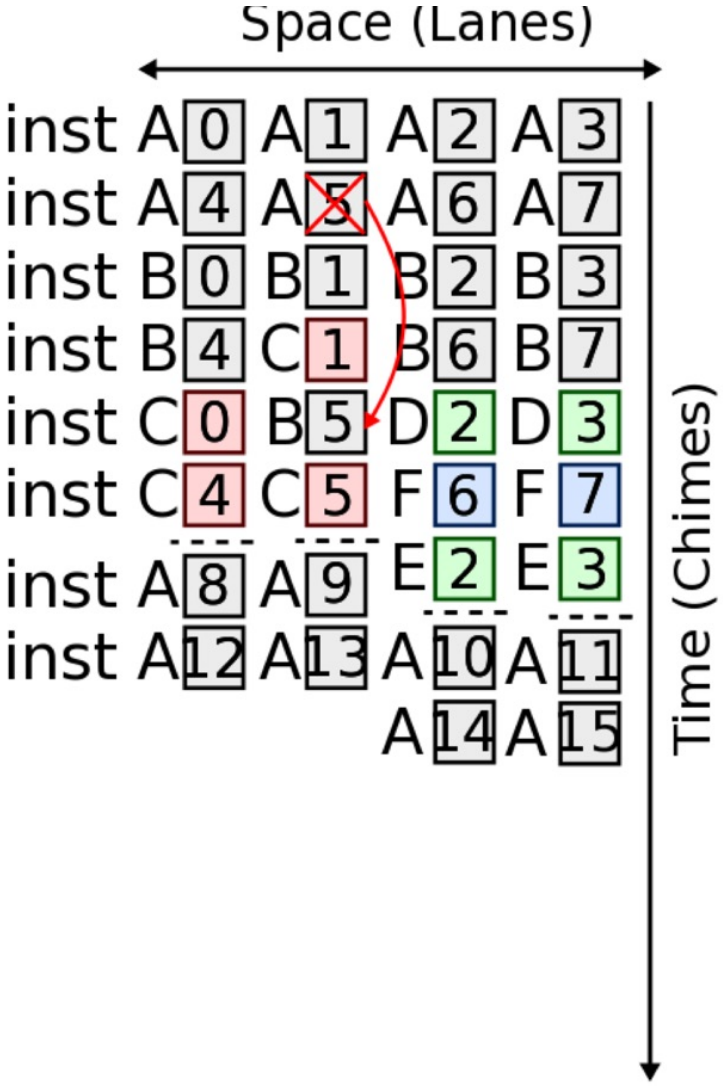
LTA HW: Fully Coupled LTA



Coupling better for regular workloads (amortize frontend/memory)

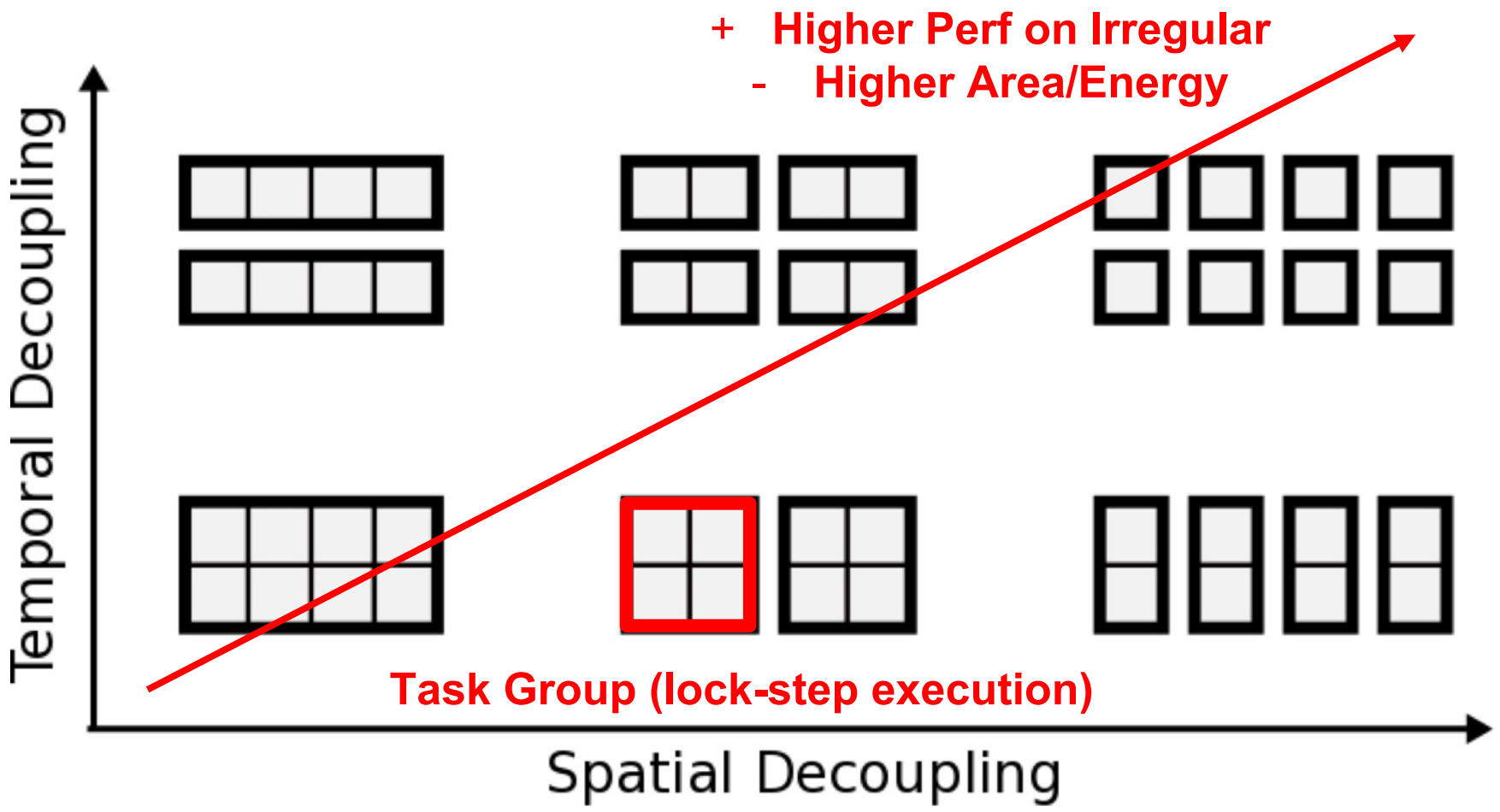
LTA HW: Fully Decoupled LTA





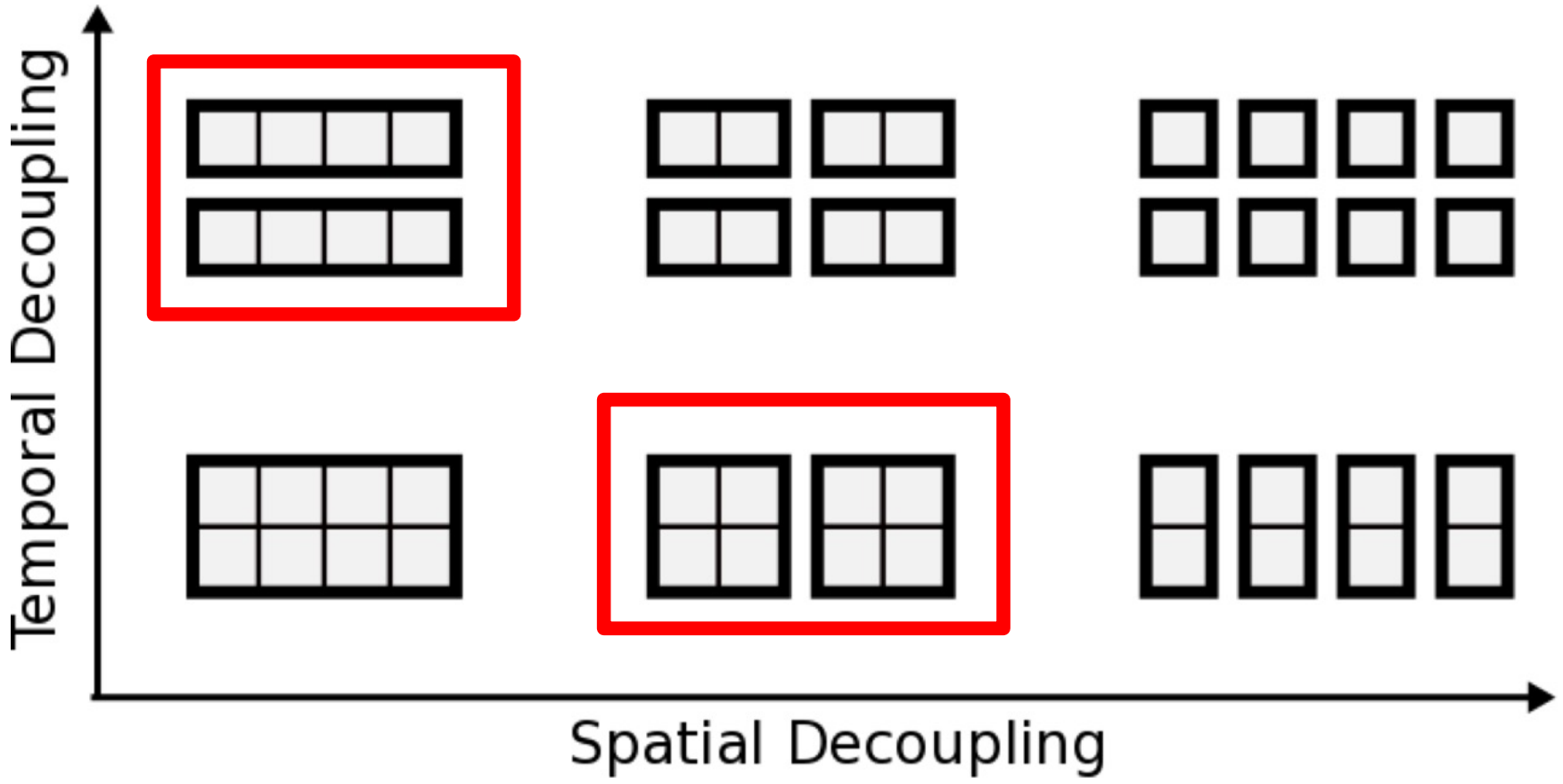
Decoupling better for irregular workloads (hide latencies)

LTA HW: Task-Coupling Taxonomy

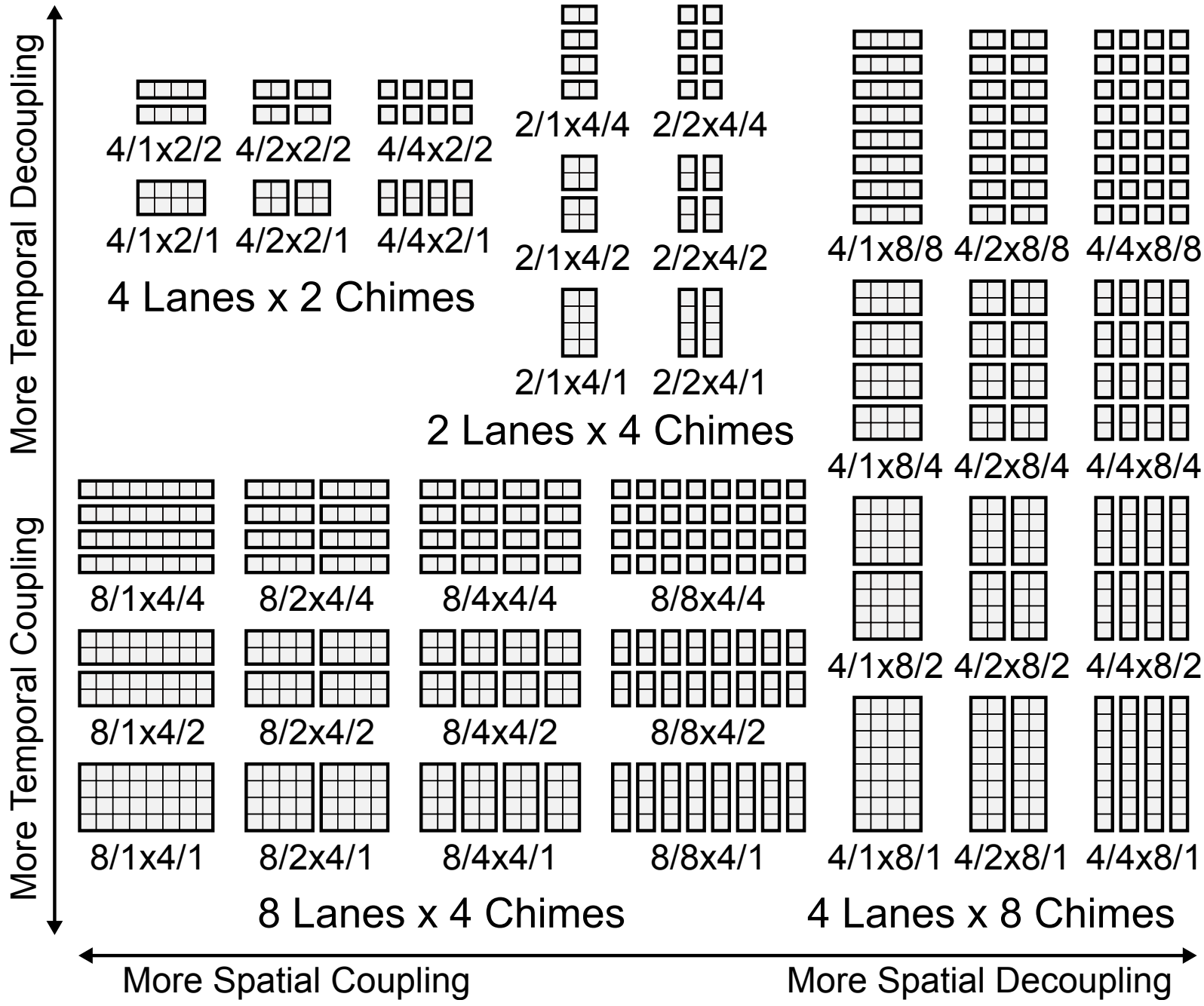


More decoupling (more task groups) in either space or time improves performance on irregular workloads at the cost of area/energy

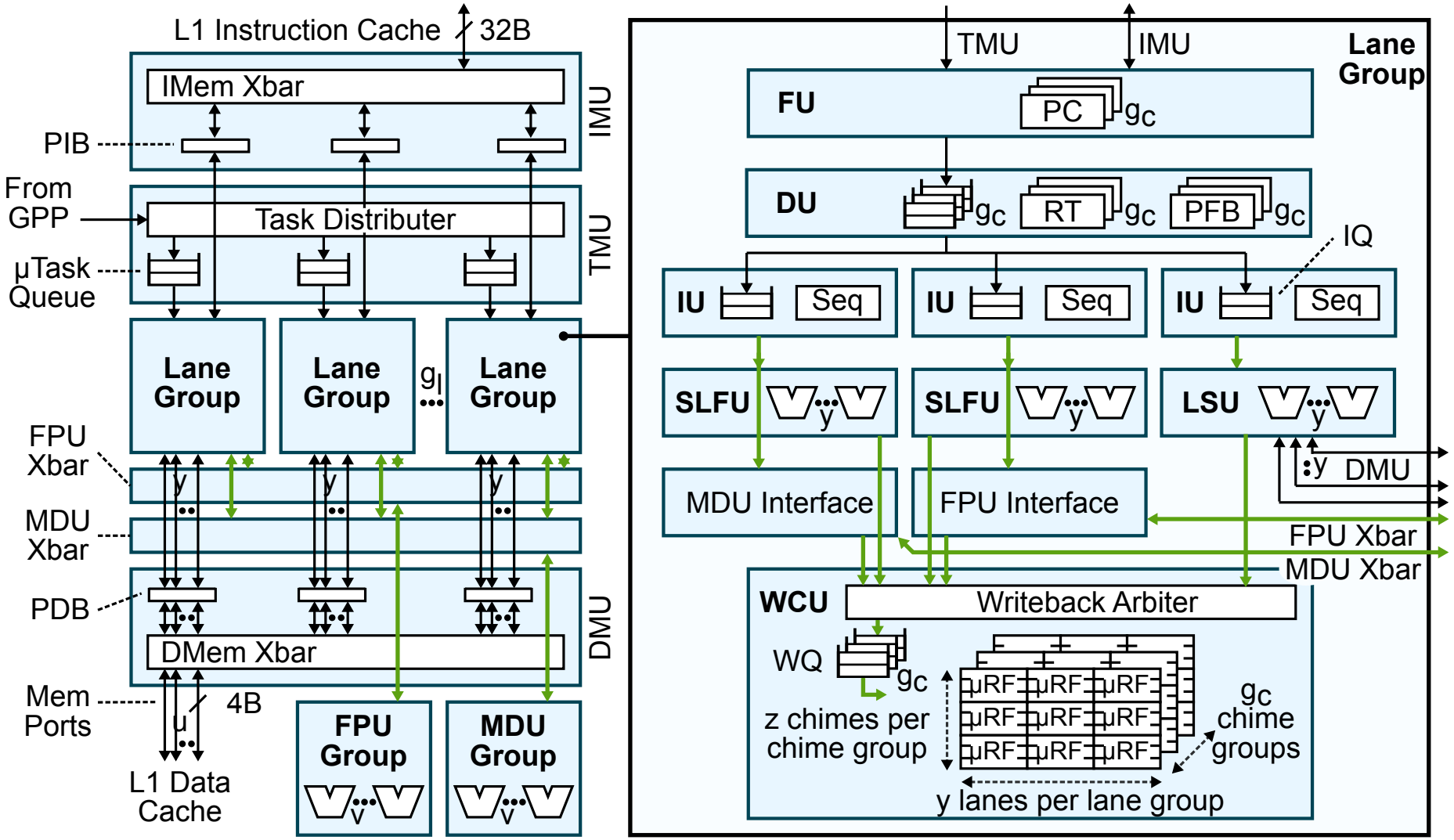
LTA HW: Task-Coupling Taxonomy



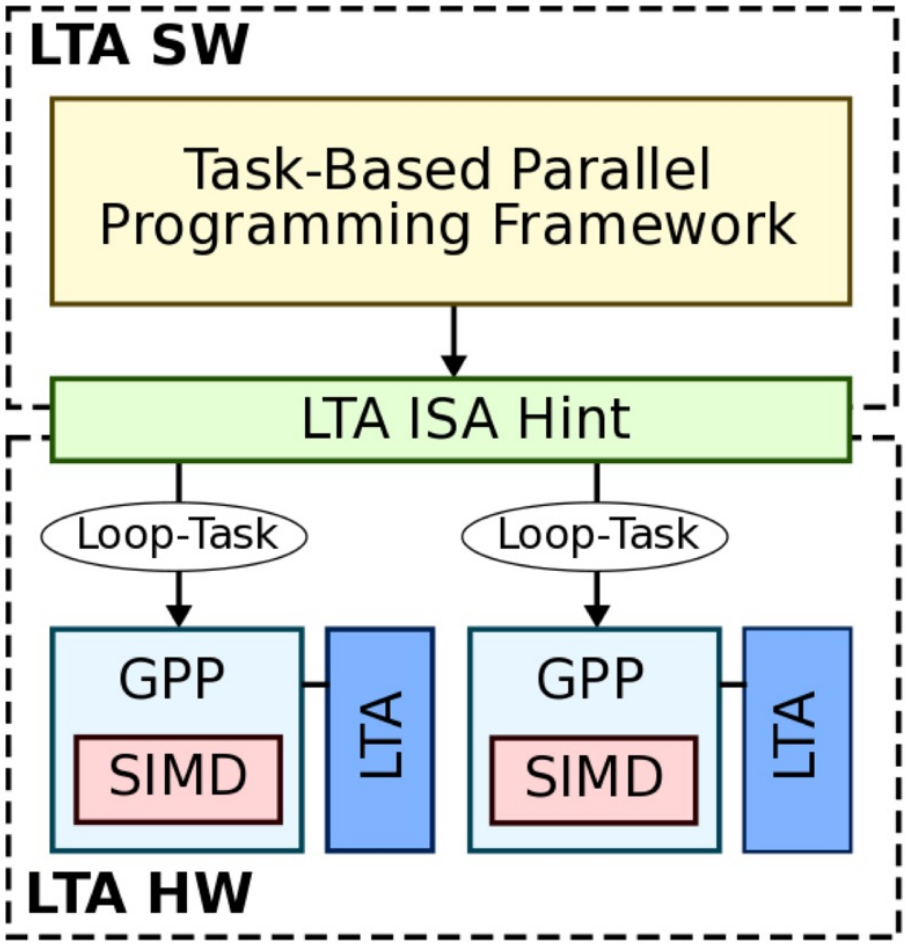
Does it matter whether we decouple in space or in time?


More Temporal Decoupling
More Temporal Coupling
4/1x2/2
4/2x2/2
4/4x2/2
4/1x2/1
4/2x2/1
4/4x2/1
4 Lanes x 2 Chimes
2/1x4/4
2/2x4/4
2/1x4/2
2/2x4/2
2/1x4/1
2/2x4/1
2 Lanes x 4 Chimes
4/1x8/8
4/2x8/8
4/4x8/8
4/1x8/4
4/2x8/4
4/4x8/4
4/1x8/2
4/2x8/2
4/4x8/2
4/1x8/1
4/2x8/1
4/4x8/1
4 Lanes x 8 Chimes
8/1x4/4
8/2x4/4
8/4x4/4
8/8x4/4
8/1x4/2
8/2x4/2
8/4x4/2
8/8x4/2
8/1x4/1
8/2x4/1
8/4x4/1
8/8x4/1
8 Lanes x 4 Chimes
More Spatial Coupling
More Spatial Decoupling

LTA HW: Microarchitectural Template



Loop-Task Accelerator (LTA) Vision

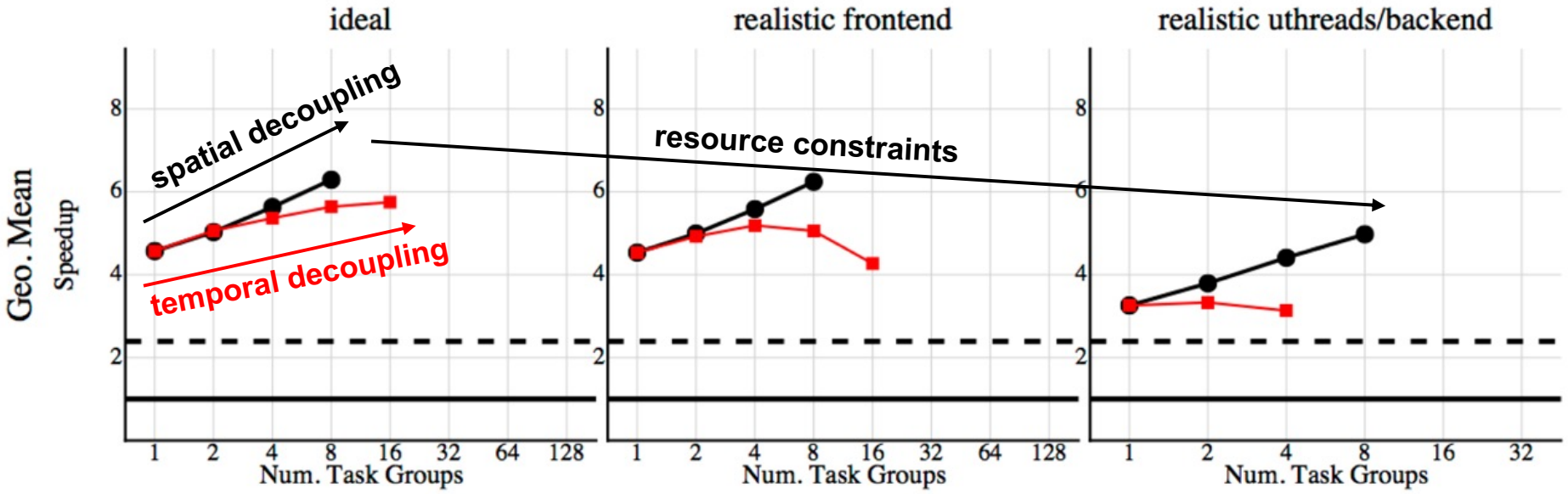


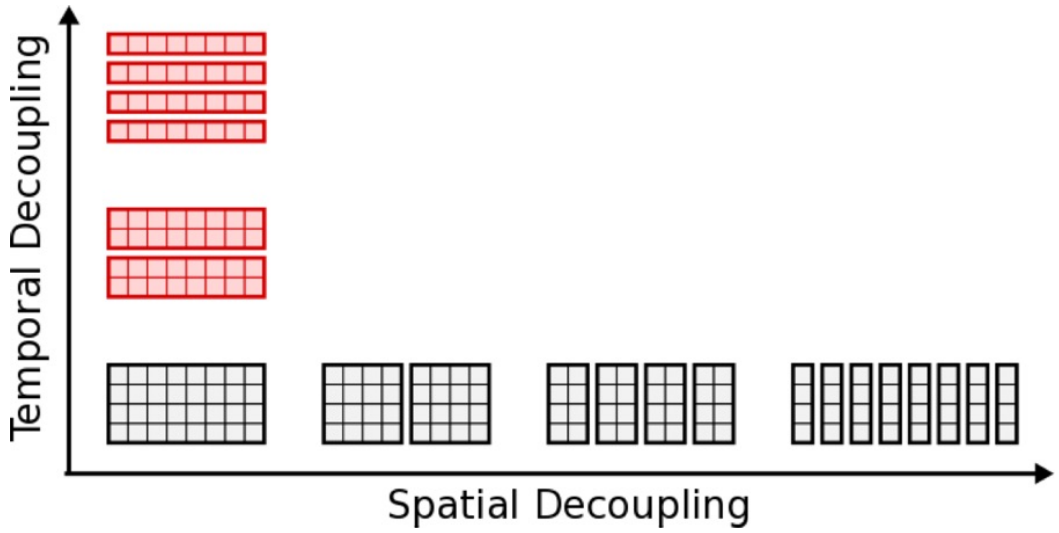
- Motivation
- Challenge #1: LTA SW
- Challenge #2: LTA HW
- **Evaluation**

Evaluation: Methodology

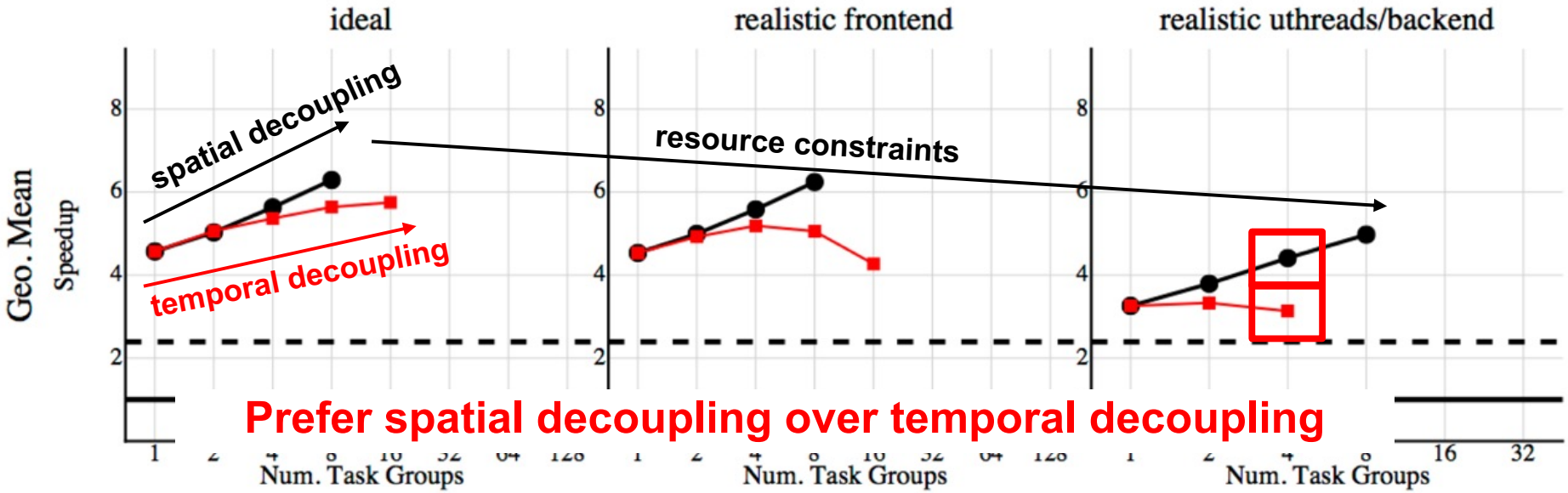
- Ported 16 application kernels from PBBS and in-house benchmark suites with diverse loop-task parallelism
  - Scientific computing: N-body simulation, MRI-Q, SGEMM
  - Image processing: bilateral filter, RGB-to-CMYK, DCT
  - Graph algorithms: breadth-first search, maximal matching
  - Search/Sort algorithms: radix sort, substring matching
- gem5 + PyMTL co-simulation for cycle-level performance
- Component/event-based area/energy modeling
  - Uses area/energy dictionary backed by VLSI results and McPAT

Evaluation: Design-Space Exploration

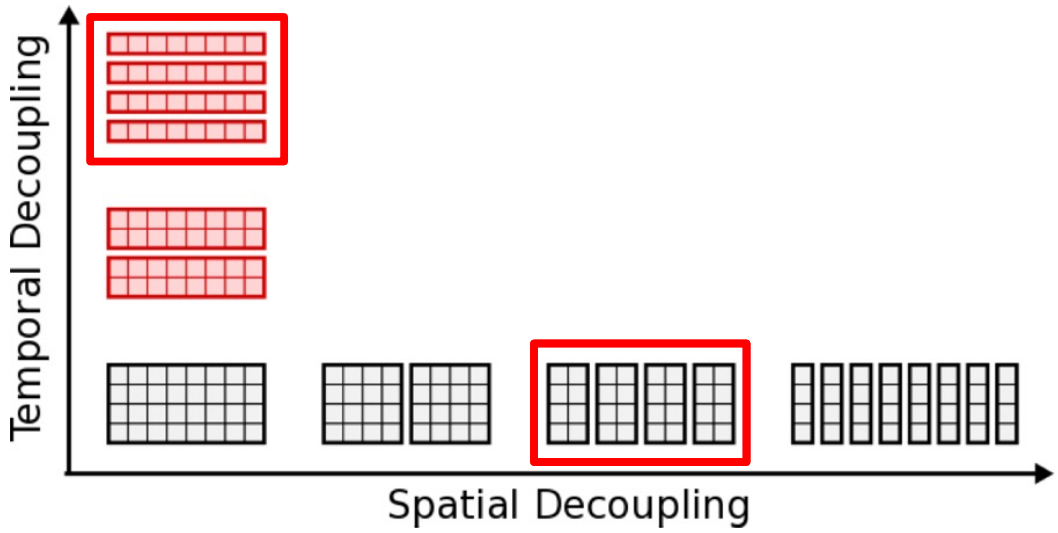




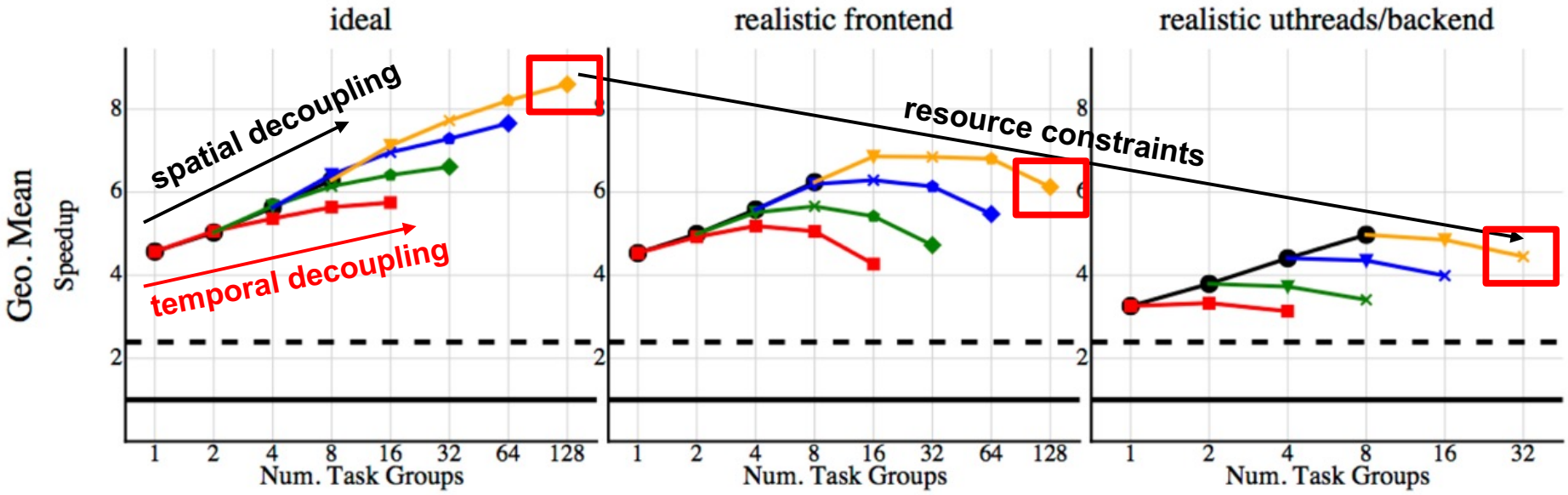
Evaluation: Design-Space Exploration

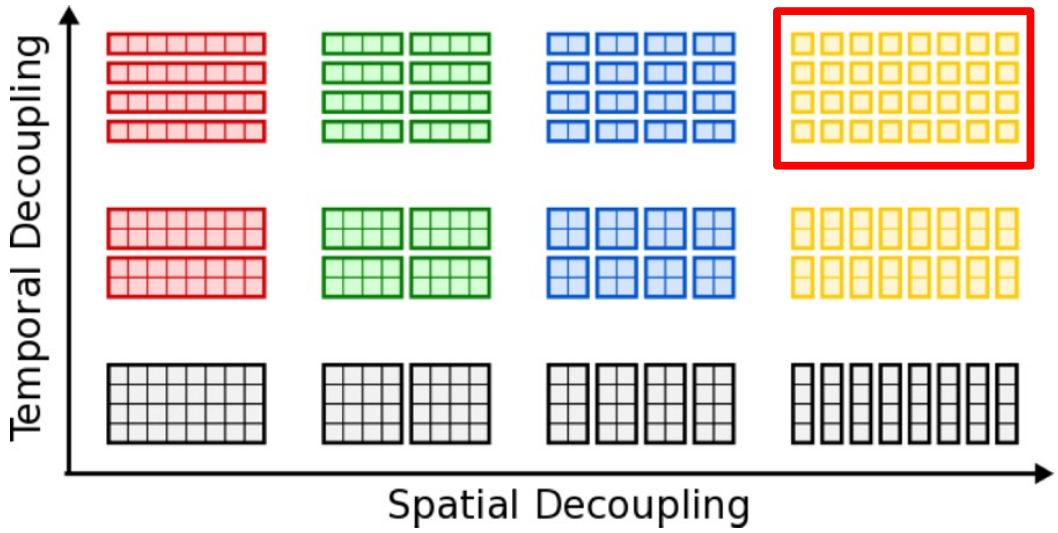


Prefer spatial decoupling over temporal decoupling

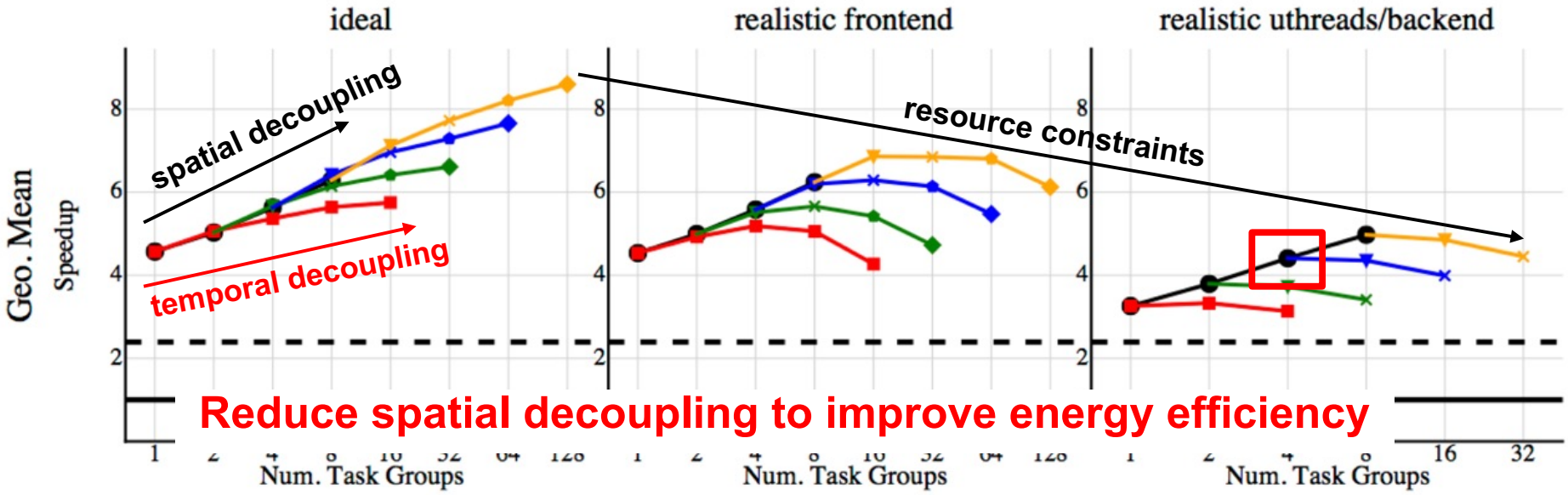


Evaluation: Design-Space Exploration

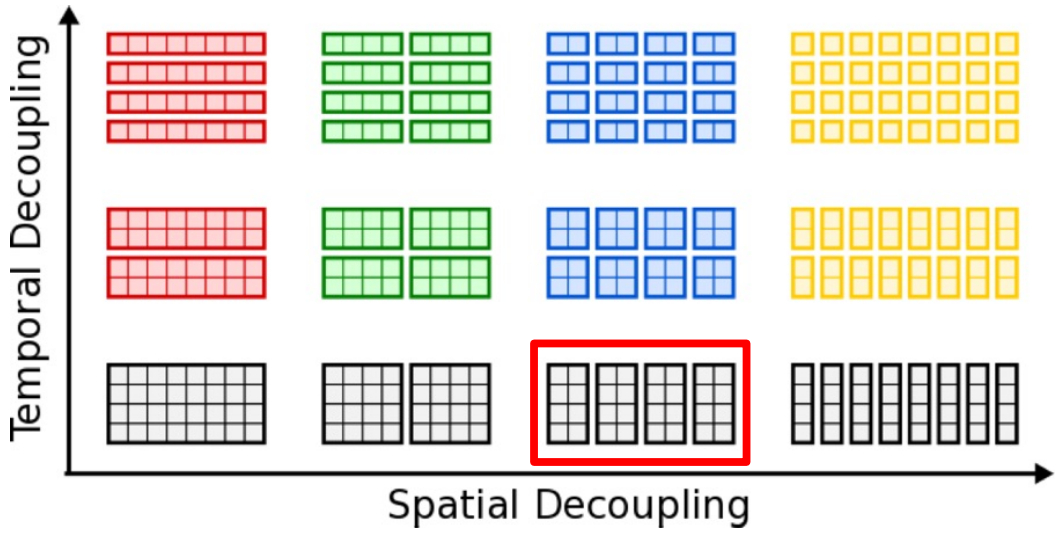




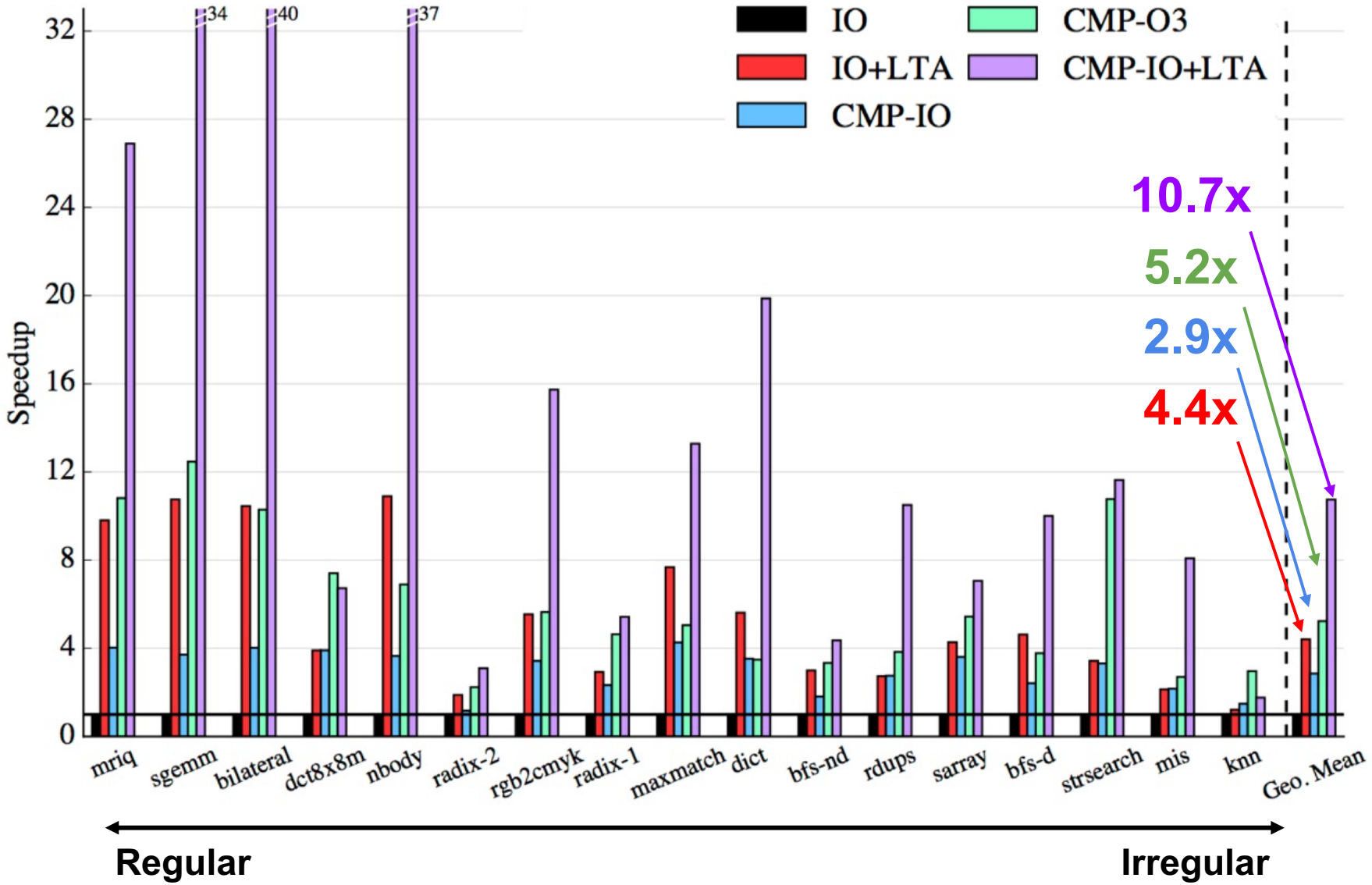
Evaluation: Design-Space Exploration



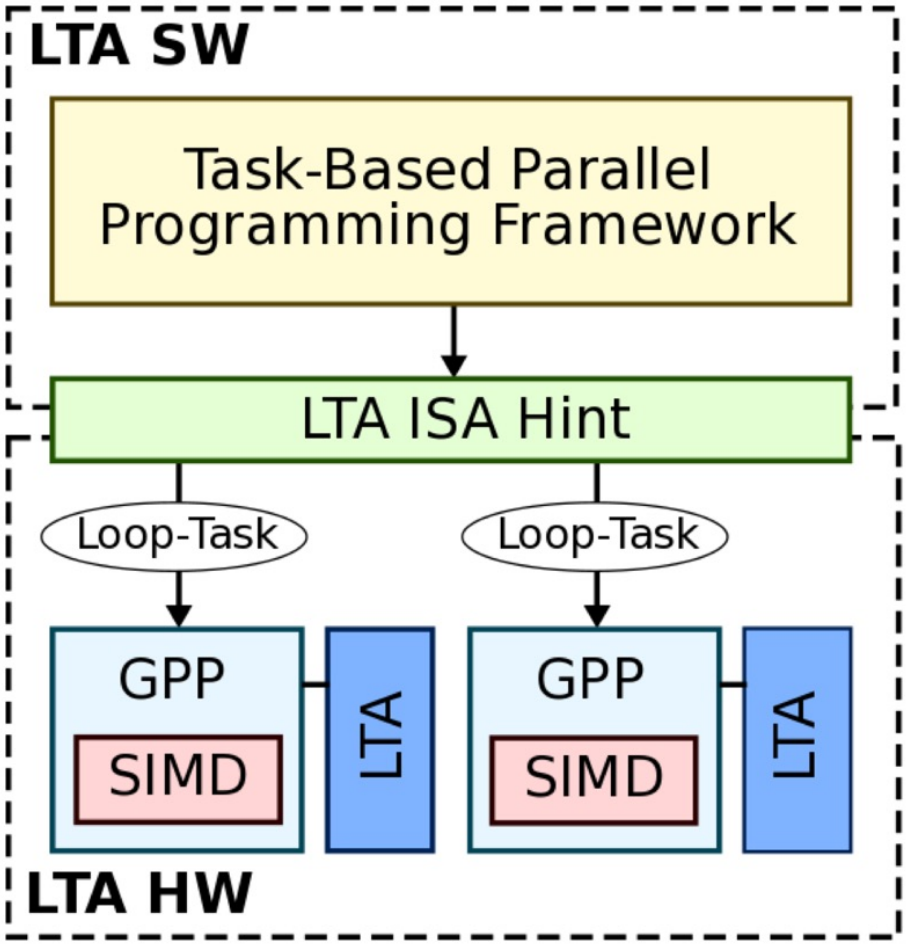
Reduce spatial decoupling to improve energy efficiency

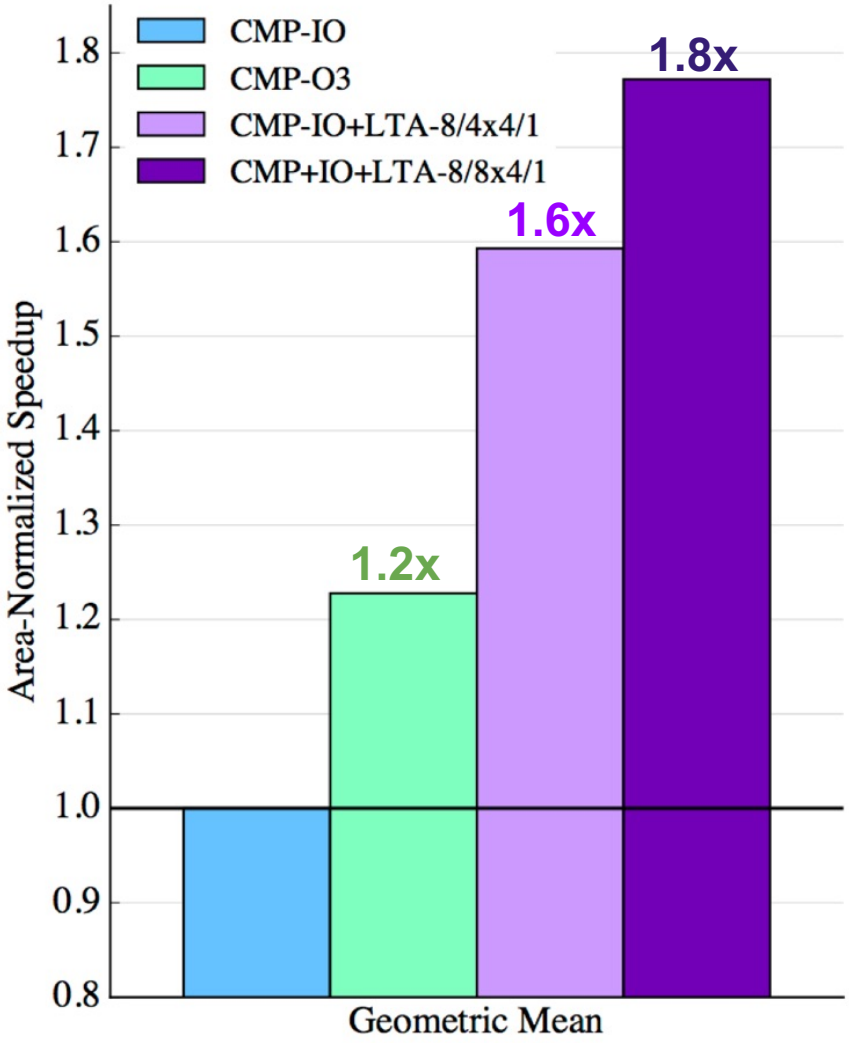


Evaluation: Multicore LTA Performance



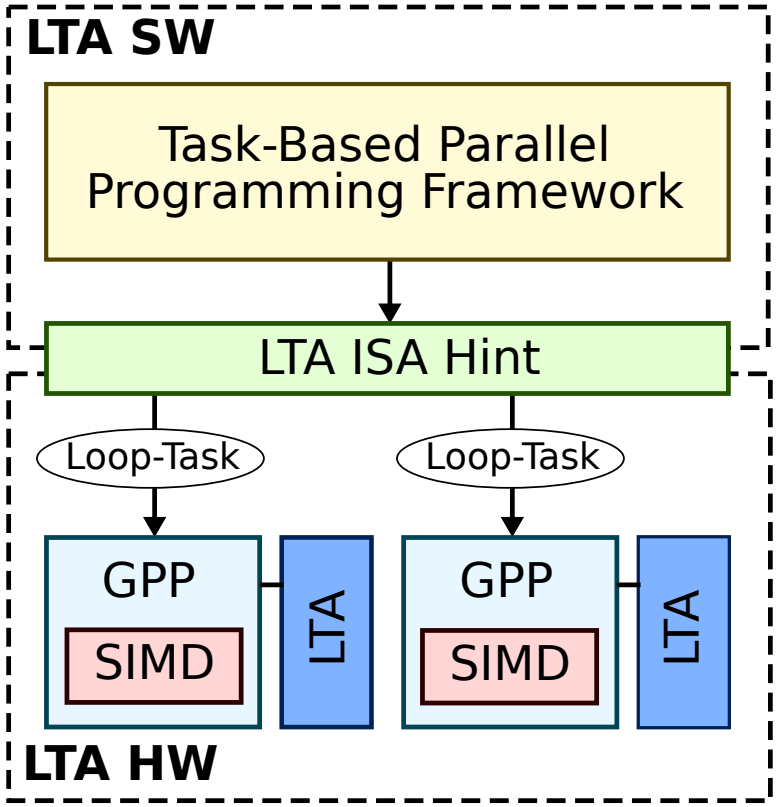
Evaluation: Area-Normalized Performance



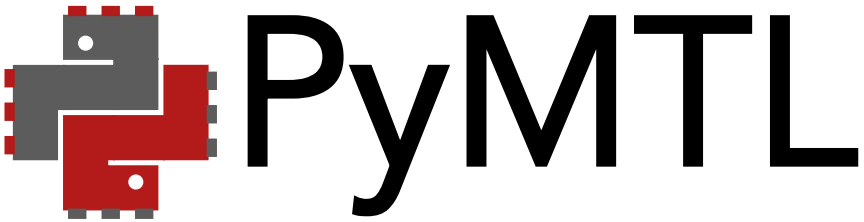


LTA Take-Away Points

- Intra-core parallel abstraction gap and inefficient execution of irregular tasks are fundamental challenges for CMPs
- LTAs address both challenges with a lightweight ISA hint and a flexible microarchitectural template



- Results suggest in a resource-constrained environment, architects should favor spatial decoupling over temporal decoupling



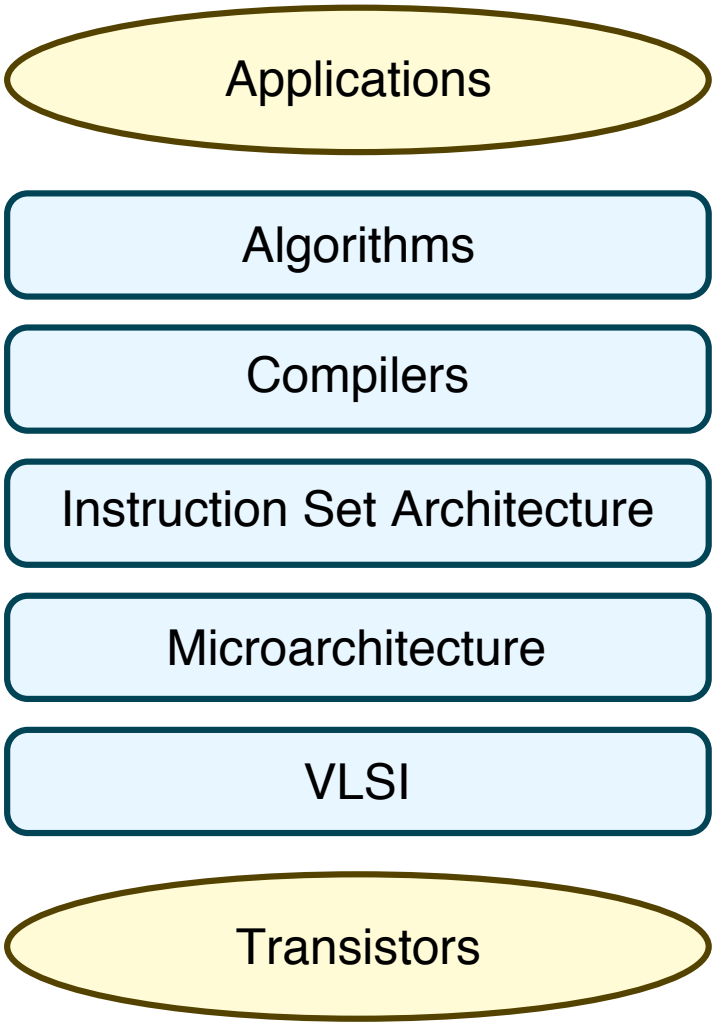
PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research

Derek Lockhart, Gary Zibrat, Christopher Batten
47th ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)
Cambridge, UK, Dec. 2014

Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks

Shunning Jiang, Berkin Ilbeyi, Christopher Batten
55thth ACM/IEEE Design Automation Conf. (DAC)
San Francisco, CA, June 2018

Multi-Level Modeling Methodologies



Functional-Level Modeling

– Behavior

Cycle-Level Modeling

– Behavior
– Cycle-Approximate
– Analytical Area, Energy, Timing

Register-Transfer-Level Modeling

– Behavior
– Cycle-Accurate Timing
– Gate-Level Area, Energy, Timing

Multi-Level Modeling Methodologies

Multi-Level Modeling Challenge

FL, CL, RTL modeling use very different languages, patterns, tools, and methodologies

SystemC is a good example of a unified multi-level modeling framework

Is SystemC the best we can do in terms of **productive** multi-level modeling?

Functional-Level Modeling

- Algorithm/ISA Development
- MATLAB/Python, C++ ISA Sim

Cycle-Level Modeling

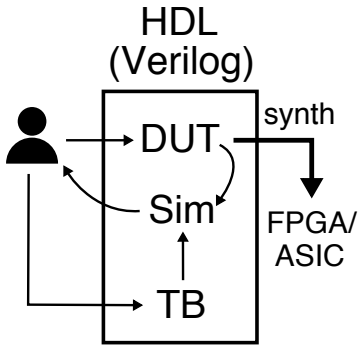
- Design-Space Exploration
- C++ Simulation Framework
- SW-Focused Object-Oriented
- gem5, SESC, McPAT

Register-Transfer-Level Modeling

- Prototyping & AET Validation
- Verilog, VHDL Languages
- HW-Focused Concurrent Structural
- EDA Toolflow

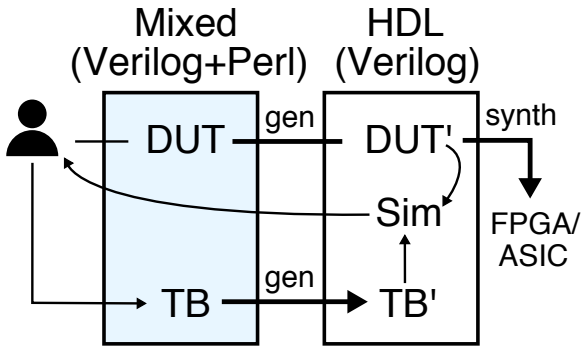
VLSI Design Methodologies

HDL
Hardware Description Language



- ✓ Fast edit-sim-debug loop
- ✓ Single language for structural, behavioral, + TB
- ✗ Difficult to create highly parameterized generators

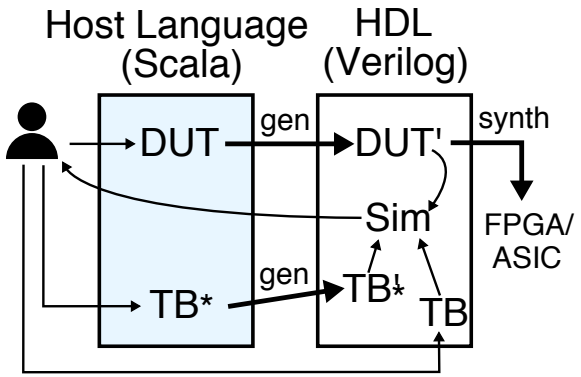
HPF
Hardware Preprocessing Framework



Example: Genesis2

- ✗ Slower edit-sim-debug loop
- ✗ Multiple languages create "semantic gap"
- ✓ Easier to create highly parameterized generators

HGF
Hardware Generation Framework



Example: Chisel

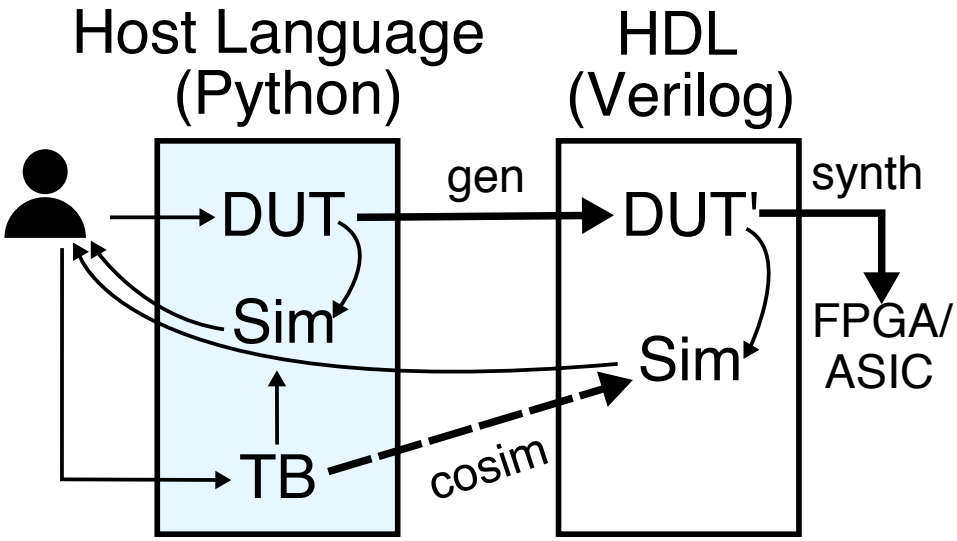
- ✗ Slower edit-sim-debug loop
- ✓ Single language for structural + behavioral
- ✓ Easier to create highly parameterized generators
- ✗ Cannot use power of host language for verification

Productive Multi-Level Modeling *and* VLSI Design

Multi-Level Modeling
SystemC

VLSI Design
Chisel

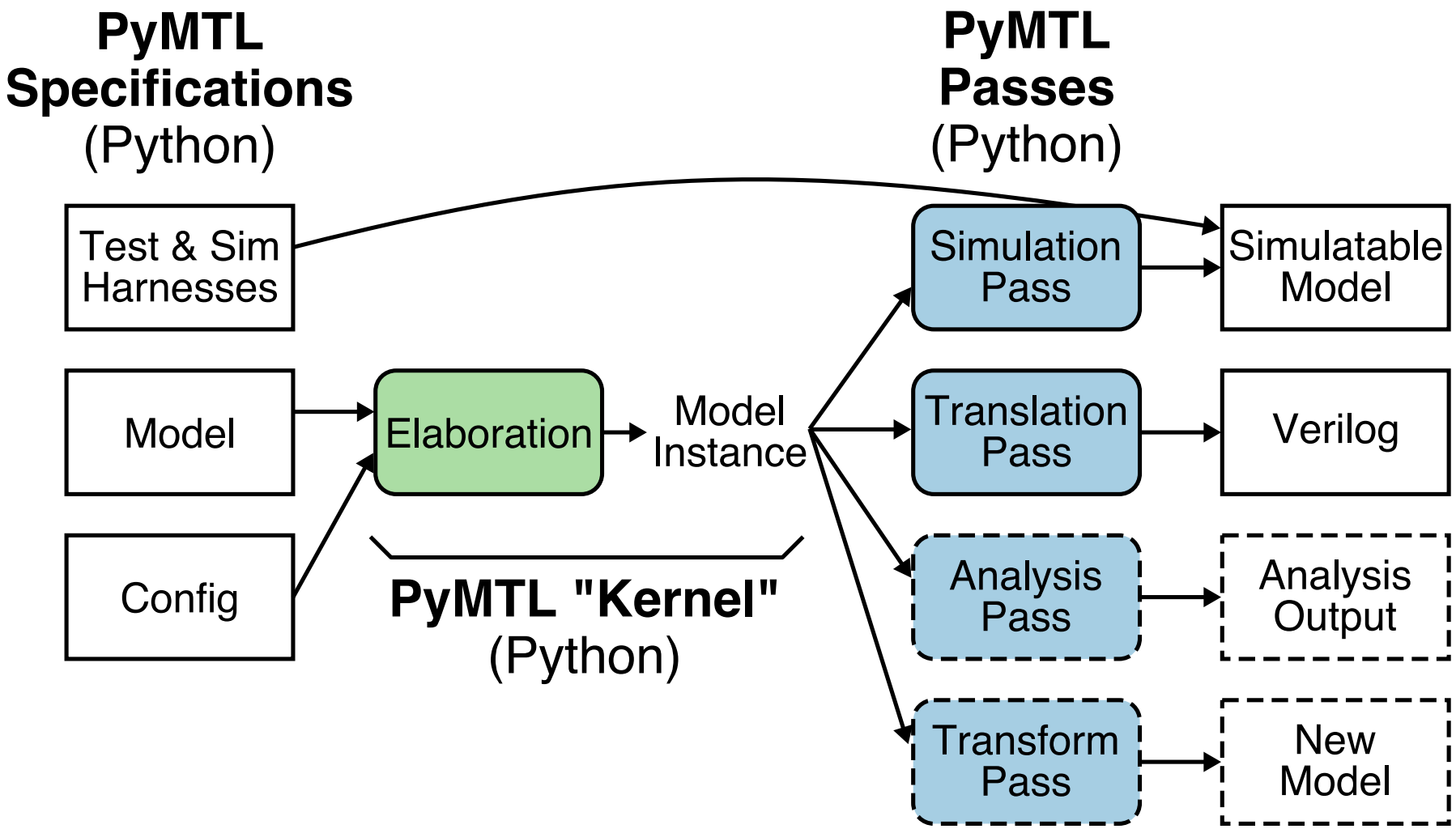
HGSF
Hardware Generation and Simulation Framework



- Single framework for ML modeling & VLSI design
- Fast edit-sim-debug loop
- Single language for structural, behavioral, + TB
- Easy to create highly parameterized generators
- Use power of host language for verification

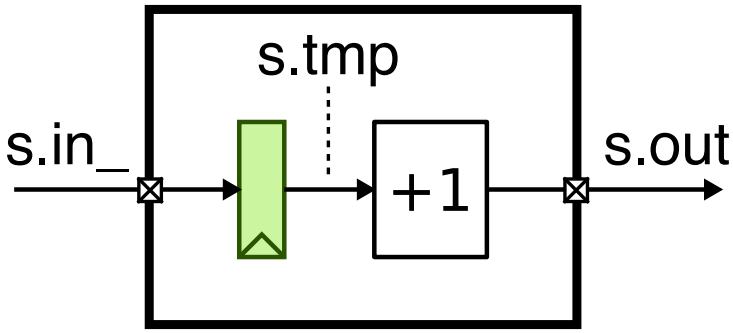
PyMTL is a Python-based hardware generation and simulation framework for computer architecture research which enables productive multi-level modeling and VLSI design

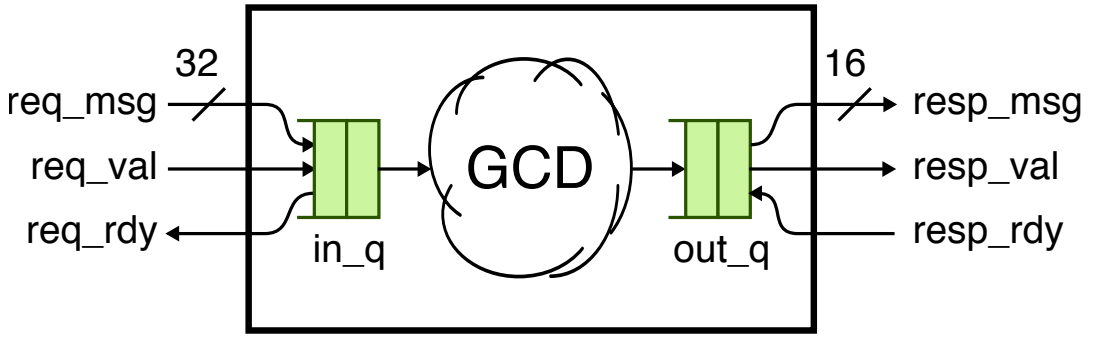
The PyMTL Framework



PyMTL v2 Syntax and Semantics

```
1  from pymtl import *
2
3  class RegIncrRTL( Model ):
4
5    def __init__( s, dtype ):
6      s.in_  = InPort ( dtype )
7      s.out  = OutPort( dtype )
8      s.tmp  = Wire   ( dtype )
9
10     @s.tick_rtl
11     def seq_logic():
12       s.tmp.next = s.in_
13
14     @s.combinational
15     def comb_logic():
16       s.out.value = s.tmp + 1
```

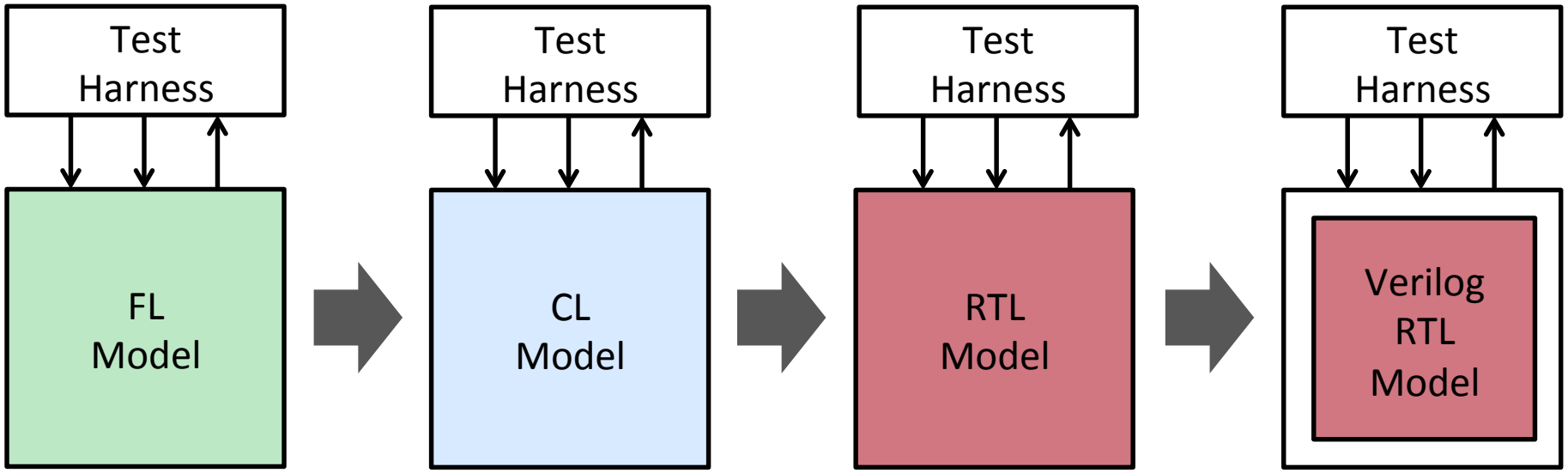





```
1  class GcdUnitFL( Model ):
2   def __init__( s ):
3
4    # Interface
5    s.req    = InValRdyBundle  ( GcdUnitReqMsg() )
6    s.resp   = OutValRdyBundle ( Bits(16)        )
7
8    # Adapters (e.g., TLM Transactors)
9    s.req_q  = InValRdyQueueAdapter  ( s.req  )
10   s.resp_q = OutValRdyQueueAdapter ( s.resp )
11
12   # Concurrent block
13   @s.tick_fl
14   def block():
15    req_msg = s.req_q.popleft()
16    result = gcd( req_msg.a, req_msg.b )
17    s.resp_q.append( result )
```

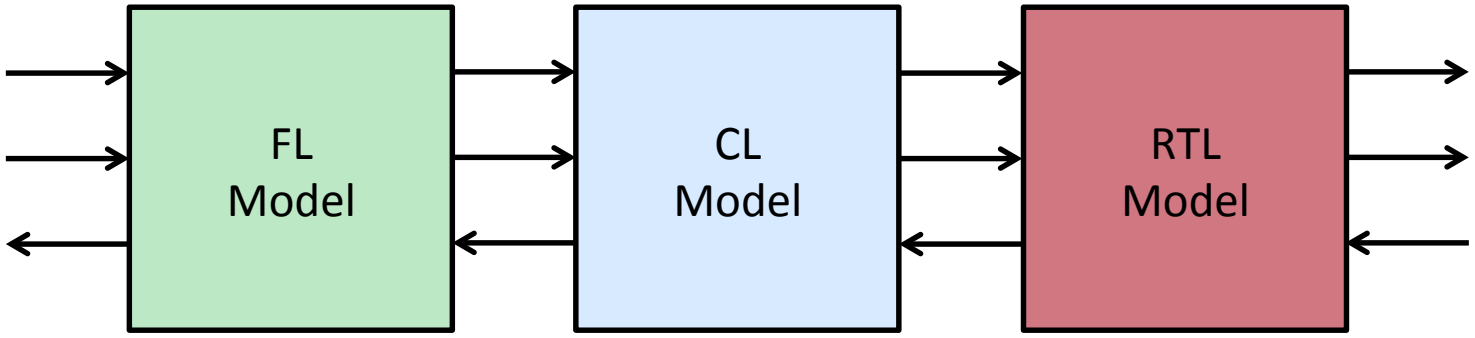
What Does PyMTL Enable?

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



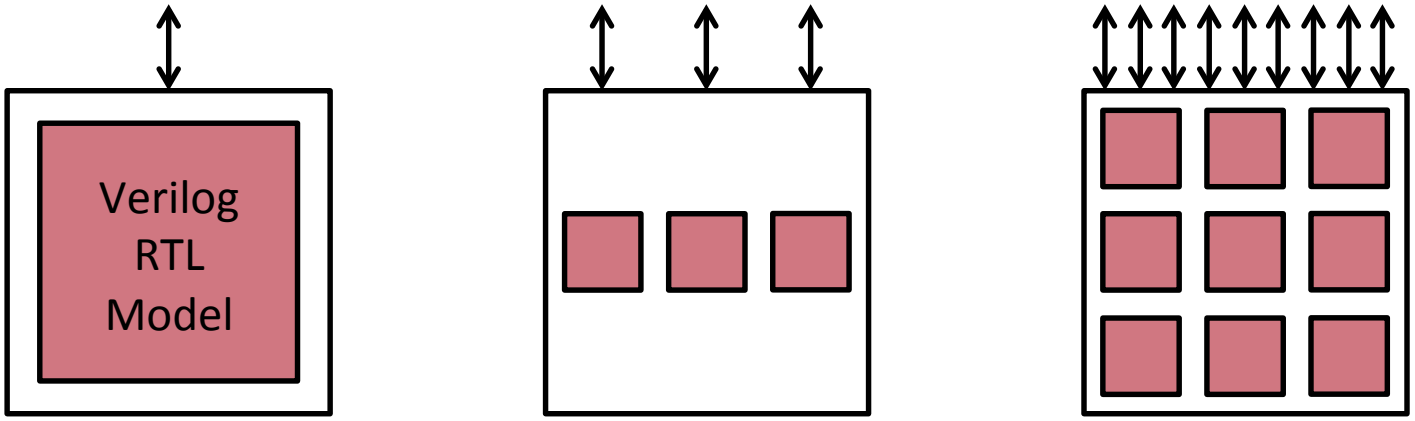
What Does PyMTL Enable?

- 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?

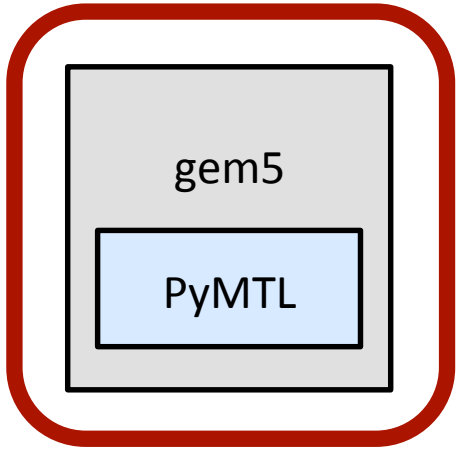
- Incremental refinement from algorithm to accelerator implementation
- Automated testing and integration of PyMTL-generated Verilog
- Multi-level co-simulation of FL, CL, and RTL models
- Construction of highly-parameterized RTL chip generators

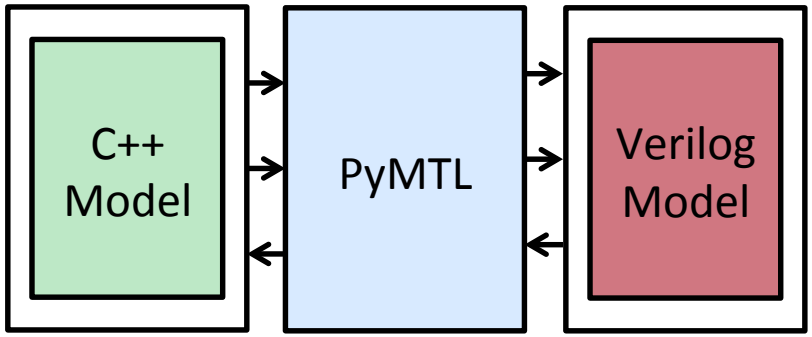


What Does PyMTL Enable?

- 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

(Used to implement CL model for LTA)



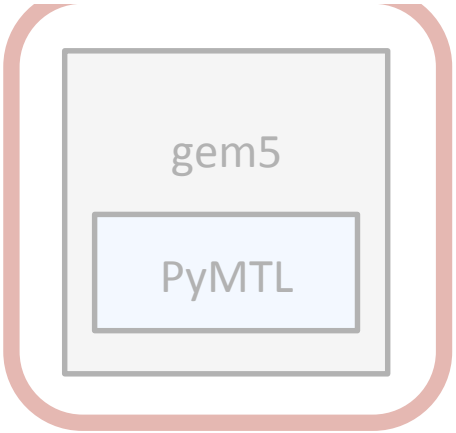


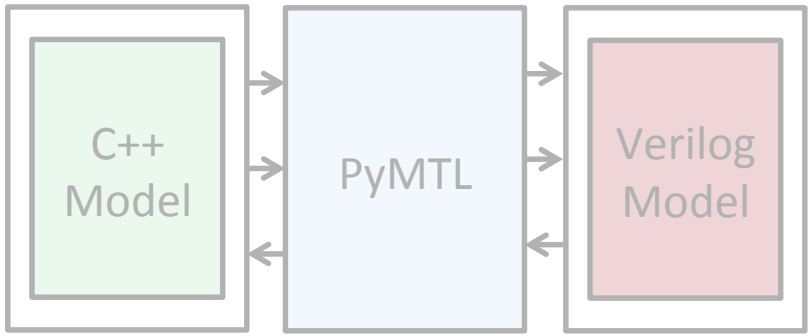
What Does PyMTL Enable?

- 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

(Use

But isn't Python too slow?



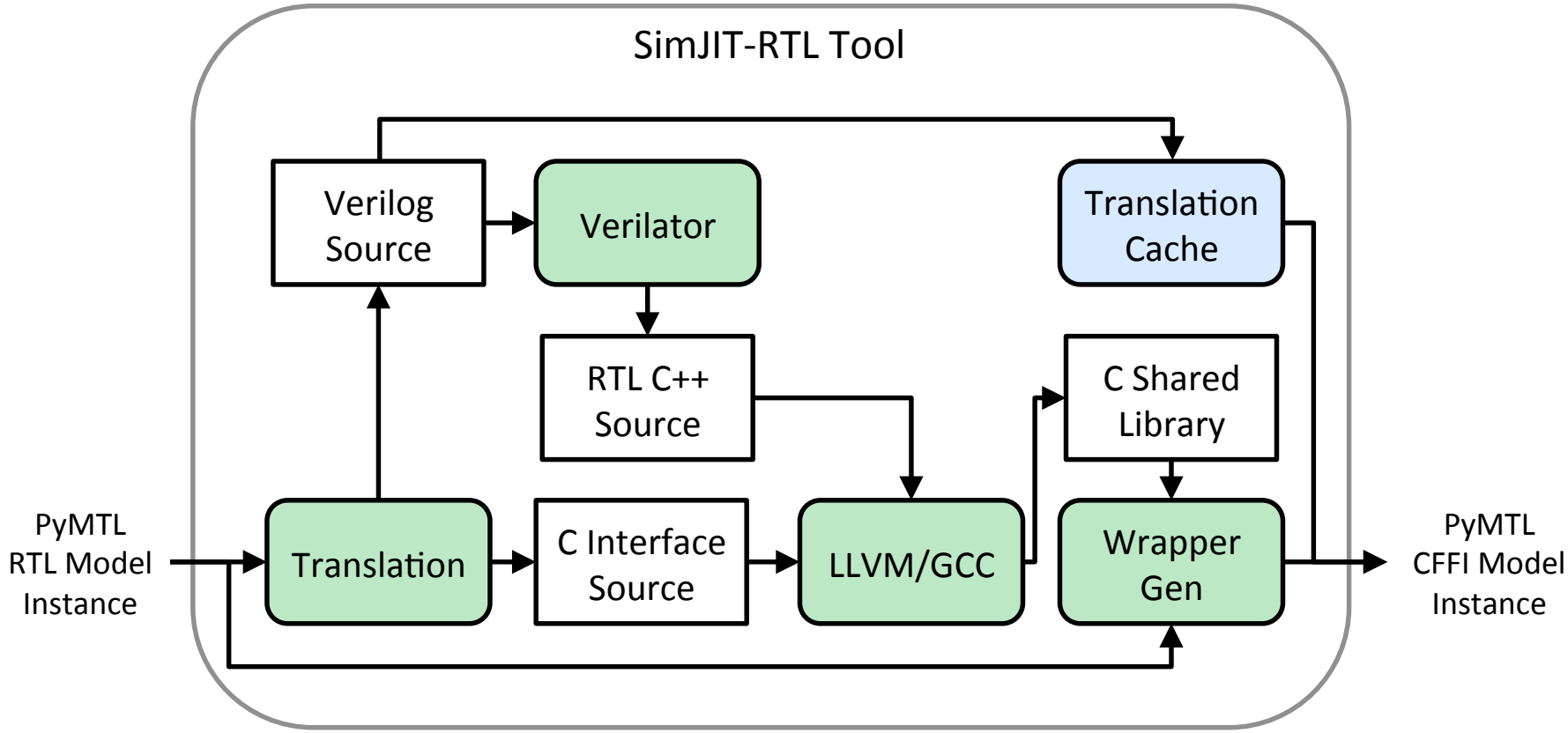


Performance/Productivity Gap

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)

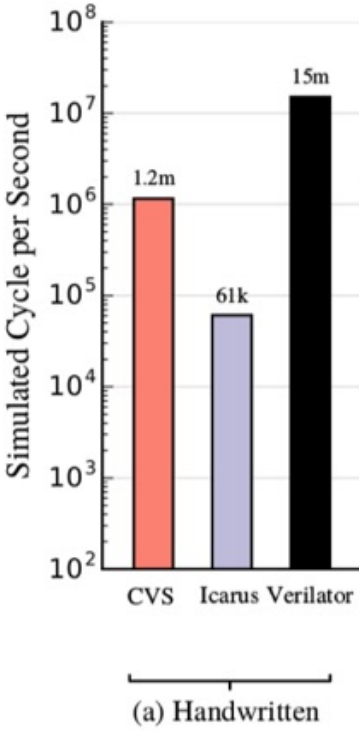
PyMTL Hybrid Python/C++ Co-Simulation



Evaluating HDLs, HGFs, and HGSFs

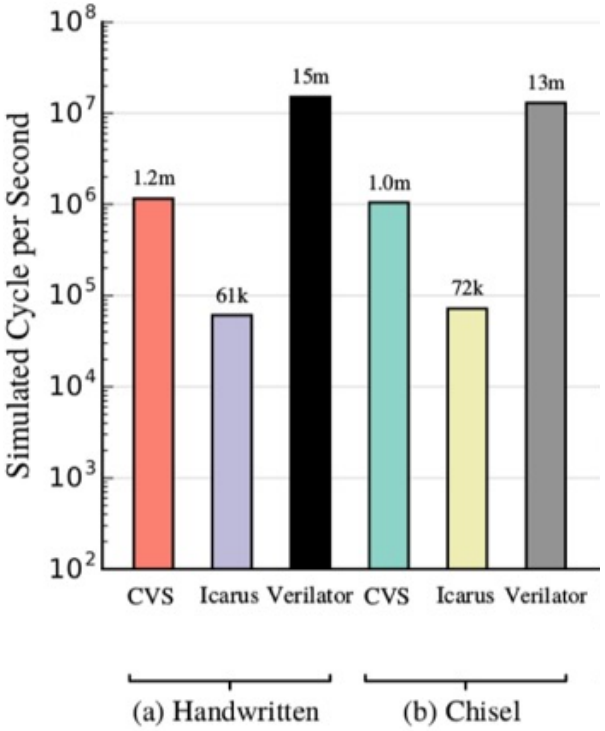
- Apple-to-apple comparison of simulator performance
- 64-bit radix-four integer iterative divider
- All implementations use same control/datapath split with the same level of detail
- Modeling and simulation frameworks:
  - Verilog: Commercial verilog simulator, Icarus, Verilator
  - HGF: Chisel
  - HGSFs: PyMTL, MyHDL, PyRTL, Migen

Productivity/Performance Gap



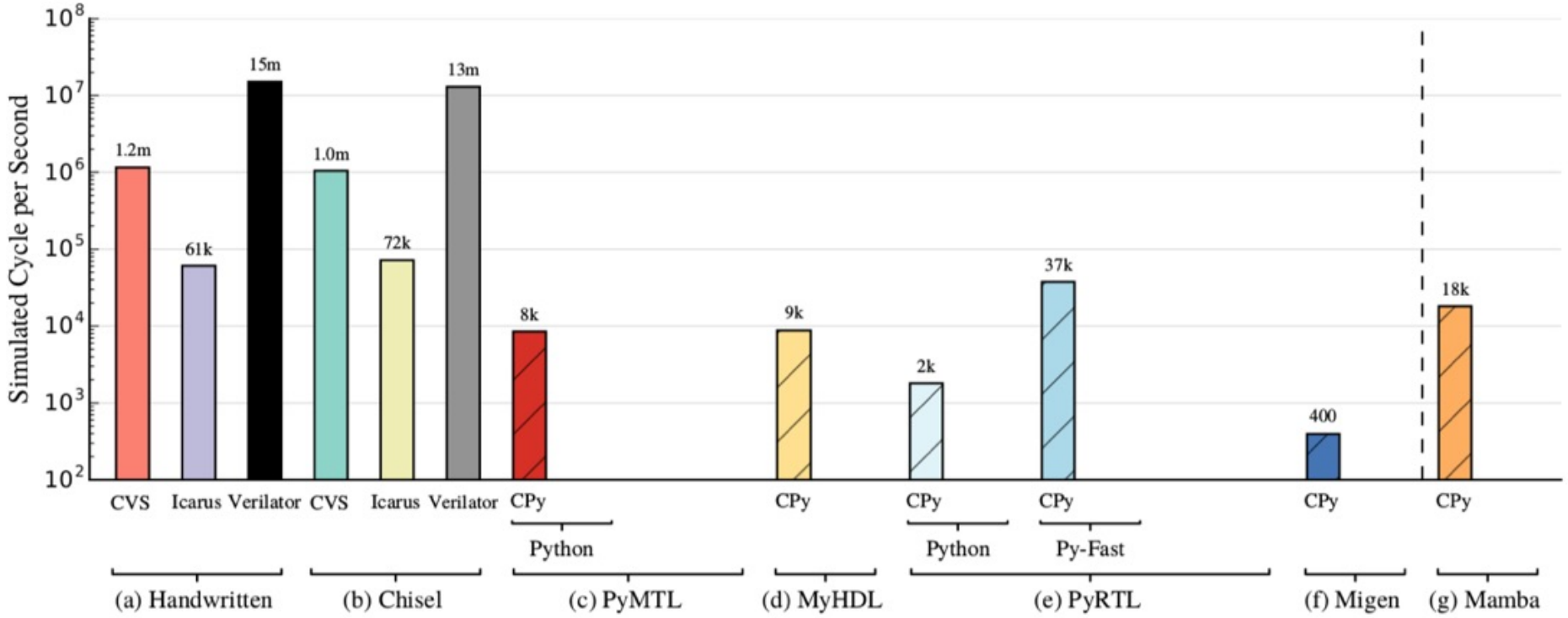
- Higher is better
- Log scale (gap is larger than it seems)
- Commercial Verilog simulator is 20× faster than Icarus
- Verilator requires C++ testbench, only works with synthesizable code, takes significant time to compile, but is 200× faster than Icarus

Productivity/Performance Gap



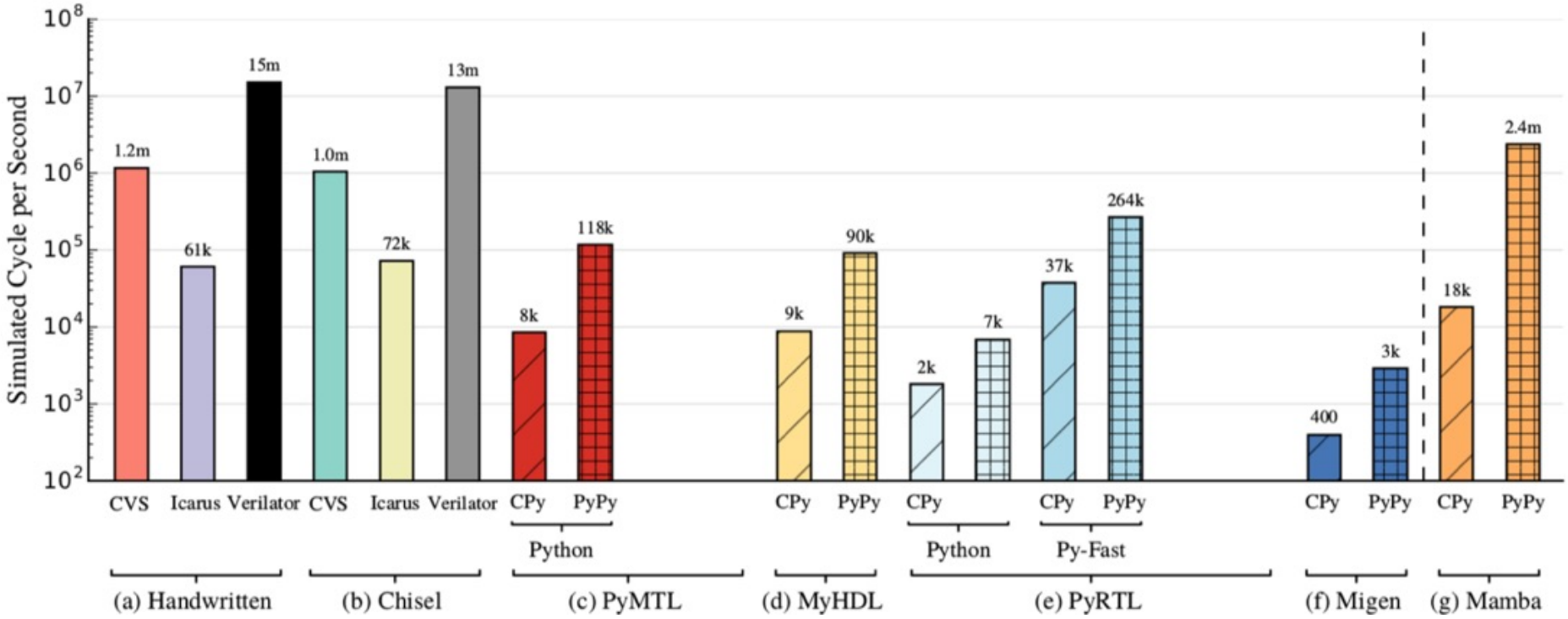
- Chisel (HGF) generates Verilog and uses Verilog simulator

Productivity/Performance Gap



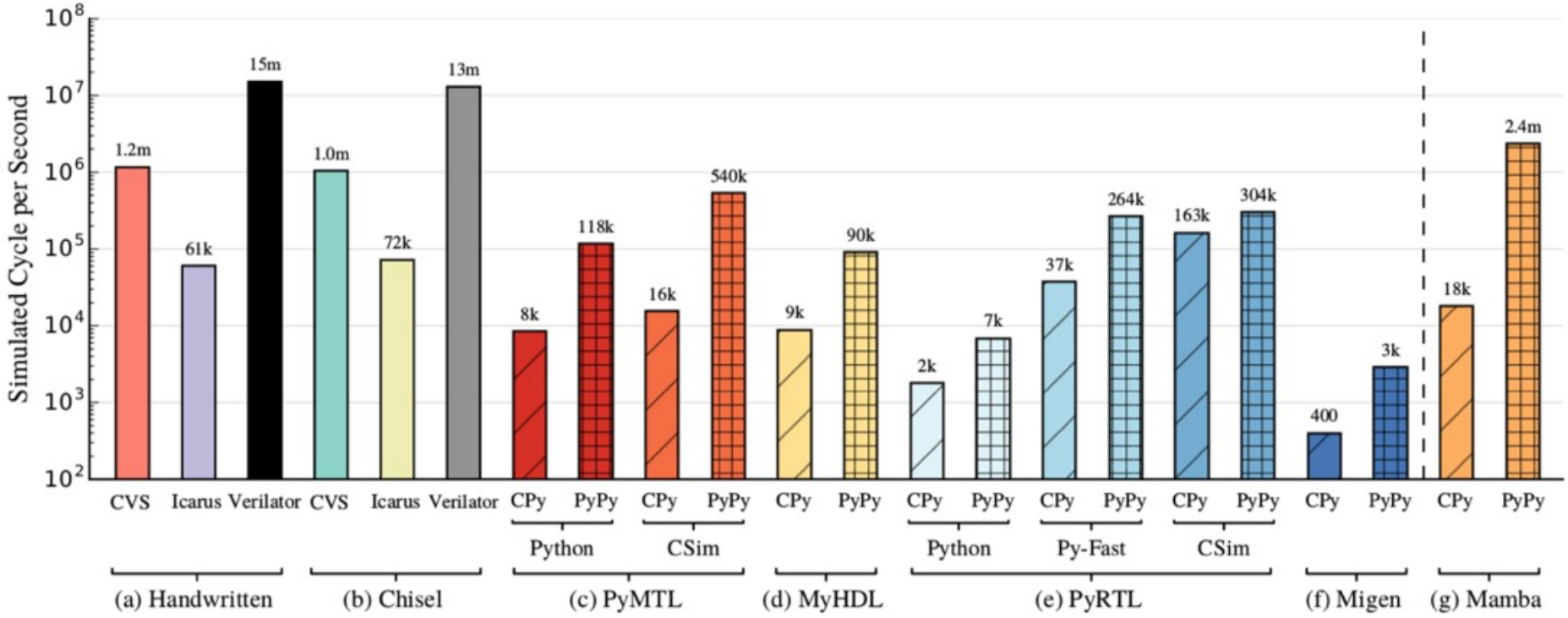
- Using CPython interpreter, Python-based HGSFs are much slower than commercial Verilog simulators; even slower than Icarus!

Productivity/Performance Gap



- Using PyPy JIT compiler, Python-based HGSFs achieve $\approx$10$\times$ speedup, but still significantly slower than commercial Verilog simulator

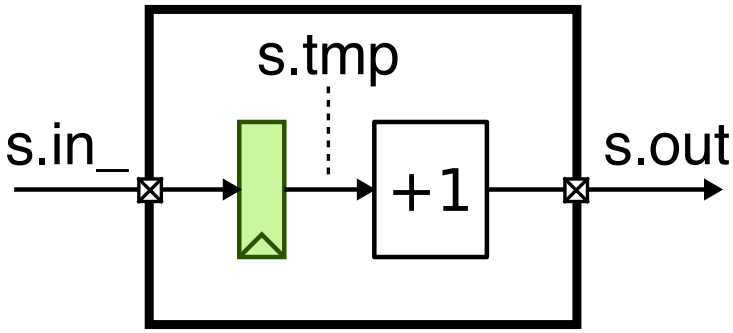
Productivity/Performance Gap



- Hybrid C/C++ co-simulation improves performance but:
  - only works for a synthesizable subset
  - may require designer to simultaneously work with C/C++ and Python

PyMTL v3 Syntax and Semantics

```
1  from pymtl import *
2
3  class RegIncrRTL( Model ):
4
5    def __init__( s, dtype ):
6      s.in_  = InValuePort ( dtype )
7      s.out  = OutValuePort( dtype )
8      s.tmp  = Wire        ( dtype )
9
10     @s.update_on_edge
11     def seq_logic():
12       s.tmp = s.in_
13
14     @s.update
15     def comb_logic():
16       s.out = s.tmp + 1
```

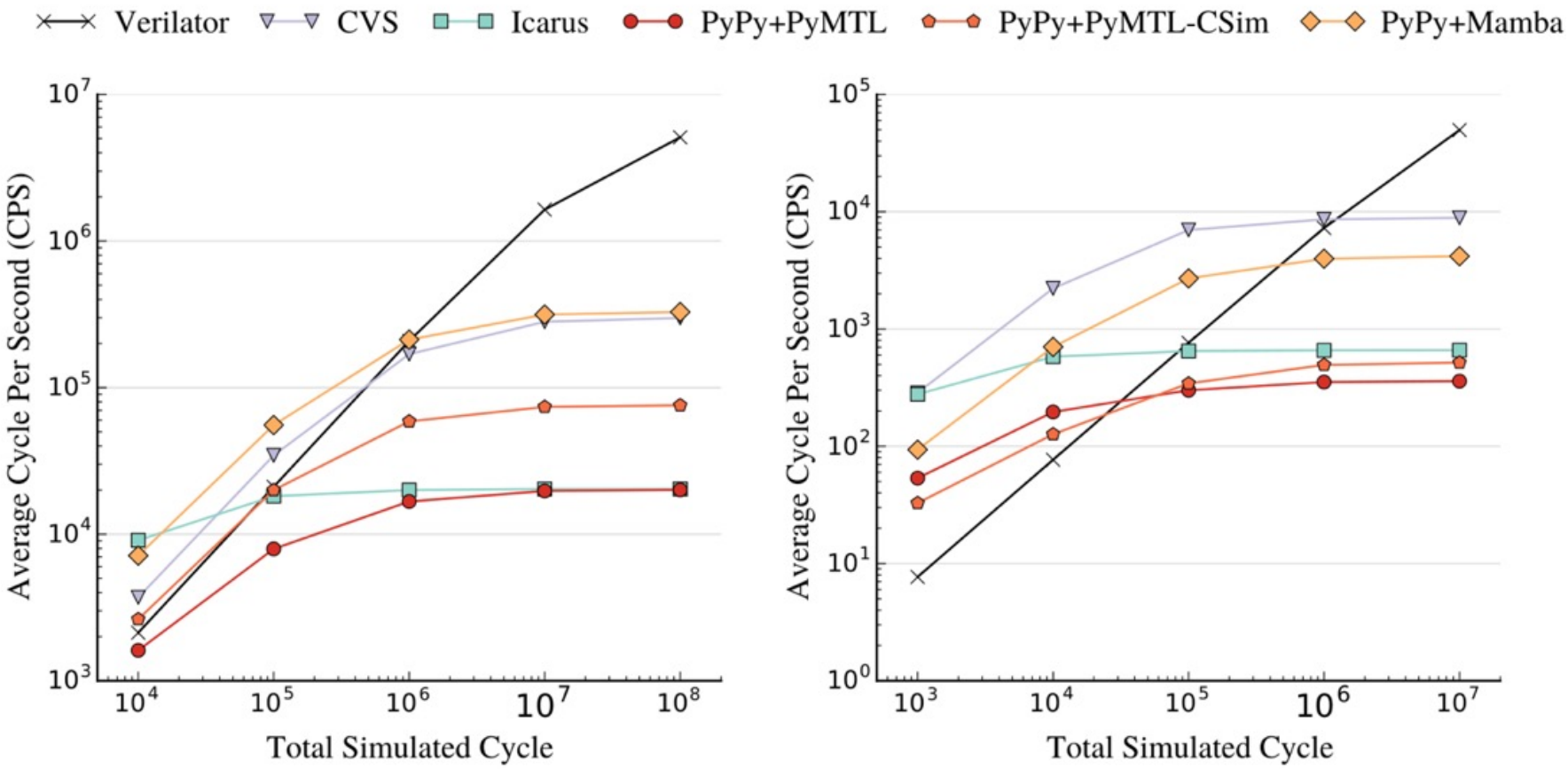


PyMTL v3 Performance

Technique	Divider	1-Core	16-core	32-core
Event-Driven	24K CPS	6.6K CPS	155 CPS	66 CPS
JIT-Aware HGSF				
+ Static Scheduling	13×	2.6×	1×	1.1×
+ Schedule Unrolling	16×	24×	0.4×	0.2×
+ Heuristic Toposort	18×	26×	0.5×	0.3×
+ Trace Breaking	19×	34×	2×	1.5×
+ Consolidation	27×	34×	47×	42×
HGSF-Aware JIT				
+ RPython Constructs	96×	48×	62×	61×
+ Huge Loop Support	96×	49×	65×	67×

- RISC-V RV32IM five-stage pipelined cores
- Only models cores, no interconnect nor caches

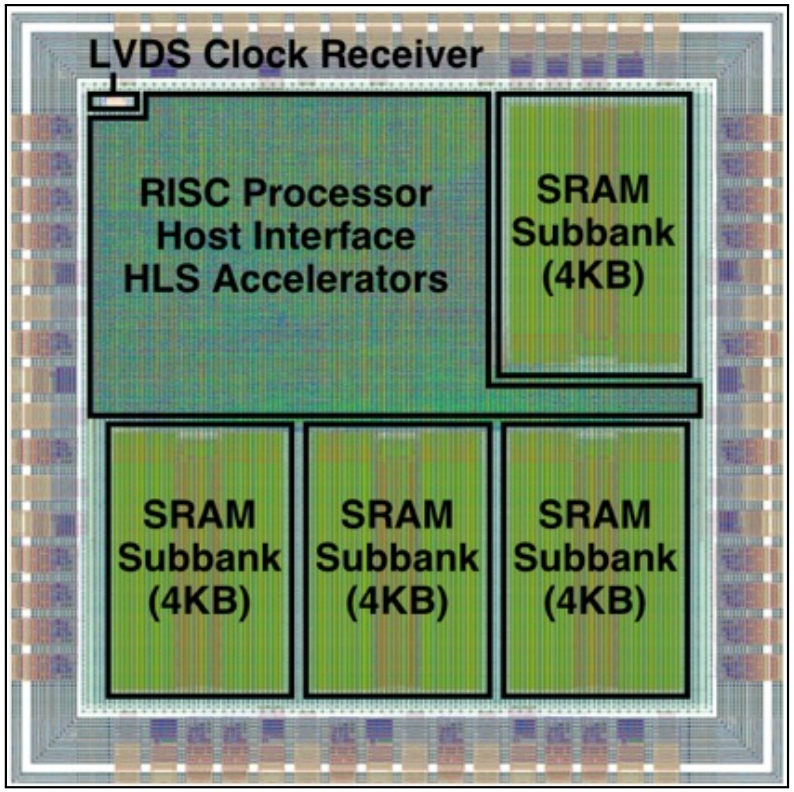
PyMTL v3 Performance with Overheads



Simulating 1 RISC-V Core **Simulating 32 RISC-V Cores**

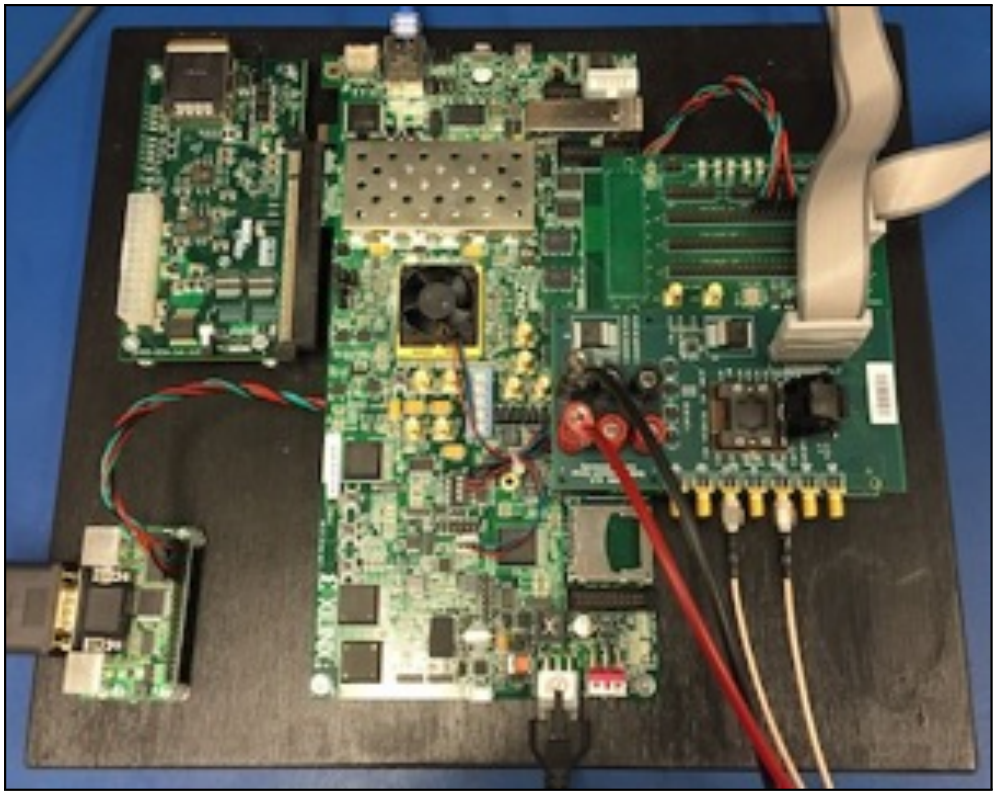
$$\text{Average Cycle Per Second} = \frac{\text{Simulated cycle}}{\textbf{Compilation time} + \textbf{Startup Overhead} + \text{Simulation time}}$$

PyMTL ASIC Tapeout



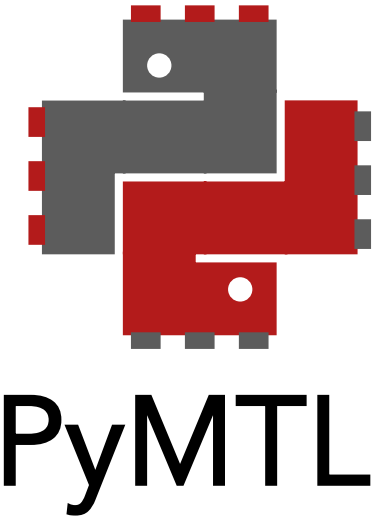
Tapeout-ready layout for RISC processor, 16KB SRAM, and HLS-generated accelerators

2x2mm 1.2M-trans in IBM 130nm



Xilinx ZC706 FPGA development board for FPGA prototyping

Custom designed FMC mezzanine card for ASIC test chips



PyMTL Take-Away Points

- PyMTL is a productive Python-based hardware generation and simulation framework to enable productive multi-level modeling and VLSI design
- PyMTL v3 leverages techniques to make the HGSF JIT-aware and the JIT HGSF JIT-aware to help close the performance/productivity gap
- Alpha versions of PyMTL v2 is available for researchers to experiment with

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

Shreesha Srinath, Christopher Torng, Berkin Ilbeyi, Moyang Wang
Shunning Jiang, Khalid Al-Hawaj, Tuan Ta, Lin Cheng
and many M.S./B.S. students

Equipment, Tools, and IP
Intel, NVIDIA, Synopsys, Cadence, Xilinx, ARM

Batten Research Group

Exploring cross-layer hardware specialization using a vertically integrated research methodology