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

Transistors (Thousands)
Frequency (MHz)
Typical Power (W)
SPECint Performance
~9%/year
~15%/year
Number of Cores
Intel 48-Core Prototype
AMD 4-Core Opteron
Intel P4
DEC Alpha 21264
MIPS R2K
- Research Overview -

- LTA Motivation
- LTA SW
- LTA HW
- LTA Evaluation
- PyMTL Motivation
- PyMTL v2
- PyMTL v3

- Simple Processor
- Embedded Architectures
- More Flexible Accelerator
- Less Flexible Accelerator
- Custom ASIC

- Performance (Tasks per Second)
- Energy Efficiency (Tasks per Joule)

- Design Power Constraint
- Flexibility vs. Specialization

- Design Performance Constraint

- High-Performance Architectures
- Less Flexible Accelerator
- More Flexible Accelerator
- Simple Processor
- Embedded Architectures
- Custom ASIC

- Flexibility vs. Specialization

- Design Power Constraint
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.

Key Metrics: Cycle Count, Cycle Time, Area, Energy

Experimenting with full-chip layout, FPGA prototypes, and test chips is a key part of our research methodology.
# Projects Within the Batten Research Group

<table>
<thead>
<tr>
<th>GPGPU Architecture</th>
<th>Integrated Voltage Regulation</th>
<th>Accelerator Centric SoC Design</th>
<th>Deep Processing in Memory</th>
<th>Accel for Loop-Level Parallelism</th>
<th>Accel for Task-Level Parallelism</th>
<th>PyMTL/Pydgin Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ISCA'13] [MICRO'14a] (AFOSR)</td>
<td>[MICRO'14b] [TCAS'18]</td>
<td>[HOTCHIPS'17] [DARPA'16] [FPGA'17] (DARPA, NSF)</td>
<td>[MICRO'14c] (DARPA, AFOSR, NSF)</td>
<td>[MICRO'16] (AFOSR, NSF)</td>
<td>[MICRO'14d] [ISPASS'15, '16] [DAC'18] (NSF, DARPA)</td>
<td>[MICRO'14d] [ISPASS'15, '16] [DAC'18] (NSF, DARPA)</td>
</tr>
<tr>
<td>Apps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uArch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLSI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circuits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tech</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```c
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

```c
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

```c
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

- Graph showing speedup comparison for different benchmarks (sgemm, dct8x8m, mriq, rgb2cmyk, bfs-nd, maxmatch, strsearch) under different parallelization techniques (scalar, tbb, avx, tbb-avx).

- The graph is divided into two categories: Regular and Irregular.

- The y-axis represents speedup, with values ranging from 0 to 14.

- The x-axis represents benchmarks.

- Benchmarks are categorized based on their regularity: Regular (left) and Irregular (right).

- The graph illustrates the performance improvements achieved with different parallelization techniques across various benchmarks.
Loop-Task Accelerator (LTA) Vision

- **Motivation**
- **Challenge #1: LTA SW**
- **Challenge #2: LTA HW**
- **Evaluation**
LTA SW: API and ISA Hint

```c
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

$rs   $a0   $a1   $a2   $a3

*loop_task_func *args  0   N  step

Hint that hardware can potentially accelerate task execution
```
LTA SW: Task-Based Runtime

Parallel For

*func  *args  0  127

Steal

*func  *args  0  63

Steal

*func  *args  64  127

Steal

*func  *args  64  95

Steal

*func  *args  96  127

Task Group

32-35  36-39

μthread  μthread

Task Group

40-43  44-47

μthread  μthread

Task Group

48-51  52-55

μthread  μthread

Task Group

56-59  60-63

μthread  μthread
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)
Decoupling better for irregular workloads (hide latencies)
More decoupling (more task groups) in either space or time improves performance on irregular workloads at the cost of area/energy.
Does it matter whether we decouple in space or in time?
Research Overview  LTA Motivation  LTA SW  • LTA HW •  LTA Evaluation  PyMTL Motivation  PyMTL v2  PyMTL v3

Cornell University

Christopher Batten

17 / 45
LTA HW: Microarchitectural Template

- L1 Instruction Cache
- IMem Xbar
- Task Distributer
- IMU
- L1 Data Cache
- L1 Instruction Cache
- 32B
- 4B
- FPU Xbar
- MDU Xbar
- PDB
- Mem Ports
- From GPP
- μTask Queue
- Lane Group
- Lane Group
- Lane Group
- IU
- IU
- IU
- Seq
- Seq
- Seq
- SLFU
- SLFU
- LSU
- DMU Interface
- FPU Interface
- WCU
- Writeback Arbiter
- FPU Xbar
- MDU Xbar
- LTA HW
- LTA Motivation
- LTA SW • LTA HW • LTA Evaluation
- PyMTL Motivation
- PyMTL v2
- PyMTL v3

Cornell University
Christopher Batten
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

- Spatial decoupling
- Temporal decoupling
- Resource constraints
Evaluation: Design-Space Exploration

Prefer spatial decoupling over temporal decoupling
Evaluation: Design-Space Exploration

- Spatial decoupling
- Temporal decoupling
- Resource constraints
Evaluation: Design-Space Exploration

Reduce spatial decoupling to improve energy efficiency
Evaluation: Multicore LTA Performance

- Regular
- Irregular

- Speedup comparison for different benchmarks:
  - mriq: 0.34x
  - sgemm: 0.40x
  - bilateral: 0.37x
  - dct8x8m: 0.37x
  - nbody: 0.37x
  - radix-2: 1.07x
  - rgb2cmyk: 0.52x
  - radix-1: 2.07x
  - maxmatch: 2.9x
  - dict: 4.4x
  - bfs-nd: 10.7x
  - rdups: 2.9x
  - sarray: 10.7x
  - bfs-d: 2.9x
  - strsearch: 2.9x
  - mis: 2.9x
  - knn: 2.9x

- Geo. Mean: 2.9x
Evaluation: Area-Normalized Performance

- CMP-IO
- CMP-O3
- CMP-IO+LTA-8/4x4/1
- CMP+IO+LTA-8/8x4/1

Geometric Mean

1.8x
1.6x
1.2x
1.0x
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

Derek Lockhart, Gary Zibrat, Christopher Batten
47th ACM/IEEE Int’l Symp. on Microarchitecture (MICRO)

Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks

Shunning Jiang, Berkin Ilbeyi, Christopher Batten
55th ACM/IEEE Design Automation Conf. (DAC)
San Francisco, CA, June 2018
Multi-Level Modeling Methodologies

Applications

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

Algorithms

Instruction Set Architecture

Compilers

Microarchitecture

VLSI

Transistors
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
- **DUT**
- **Sim**
- **TB**

- HDL (Verilog)
- FPGA/ASIC

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

### HPF Hardware Preprocessing Framework
- **DUT**
- **Sim**
- **TB**

- Mixed (Verilog+Perl)
- HDL (Verilog)

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

Example: Genesis2

### HGF Hardware Generation Framework
- **DUT**
- **Sim**
- **TB**

- Host Language (Scala)
- HDL (Verilog)

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

Example: Chisel
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

Host Language
(Python)

HDL
(Verilog)

DUT
Sim
TB
cosim
gen
synth
FPGA/ASIC

DUT'
Sim
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 Specifications (Python)

- Test & Sim Harnesses
- Model
- Config

Elaboration

Model Instance

PyMTL "Kernel" (Python)

PyMTL Passes (Python)

- Simulation Pass
- Translation Pass
- Analysis Pass
- Transform Pass

Analysis Output
New Model
Simulatable Model
Verilog
from pymtl import *

class RegIncrRTL( Model ):

    def __init__( s, dtype ):
        s.in_ = InPort( dtype )
        s.out = OutPort( dtype )
        s.tmp = Wire( dtype )

    @s.tick_rtl
    def seq_logic():
        s.tmp.next = s.in_

    @s.combinational
    def comb_logic():
        s.out.value = s.tmp + 1
```python
class GcdUnitFL(Model):
    def __init__(s):
        # Interface
        s.req = InValRdyBundle(GcdUnitReqMsg())
        s.resp = OutValRdyBundle(Bits(16))

        # Adapters (e.g., TLM Transactors)
        s.req_q = InValRdyQueueAdapter(s.req)
        s.resp_q = OutValRdyQueueAdapter(s.resp)

        # Concurrent block
        @s.tick_fl
def block():
            req_msg = s.req_q.popleft()
            result = gcd(req_msg.a, req_msg.b)
            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

SimJIT-RTL Tool

- Verilog Source
- Verilator
- RTL C++ Source
- Translation Cache
- C Shared Library
- Wrapper Gen
- LLVM/GCC
- C Interface Source
- Translation
- PyMTL RTL Model Instance
- PyMTL CFFI Model Instance

Research Overview | LTA Motivation | LTA SW | LTA HW | LTA Evaluation | PyMTL Motivation | PyMTL v2 | PyMTL v3
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 $20 \times$ faster than Icarus
- Verilator requires C++ testbench, only works with synthesizable code, takes significant time to compile, but is $200 \times$ faster than Icarus
Productivity/Performance Gap

- Chisel (HGF) generates Verilog and uses Verilog simulator
Using CPython interpreter, Python-based HGSFs are much slower than commercial Verilog simulators; even slower than Icarus!
Using PyPy JIT compiler, Python-based HGSFs achieve $\approx 10 \times$ speedup, but still significantly slower than commercial Verilog simulator.
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
from pymtl import *

class RegIncrRTL( Model ):

def __init__( s, dtype ):
    s.in_ = InValuePort ( dtype )
    s.out = OutValuePort( dtype )
    s.tmp = Wire ( dtype )

@s.update_on_edge
def seq_logic():
    s.tmp = s.in_

@s.update
def comb_logic():
    s.out = s.tmp + 1
## PyMTL v3 Performance

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

- 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

Average Cycle Per Second = \frac{Simulated cycle}{Compilation time + Startup Overhead + 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
Research Overview

LTA Motivation
LTA SW
LTA HW
LTA Evaluation
PyMTL Motivation
PyMTL v2
PyMTL v3

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

Cornell University
Christopher Batten
Exploring cross-layer hardware specialization using a vertically integrated research methodology.

**LTA SW**
- Task-Based Parallel Programming Framework
- LTA ISA Hint
- Loop-Task
- GPP
- SIMD
- LTA

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

**PyMTL**
- Cornell University
- Christopher Batten