Extending High-Level Synthesis for Task-Parallel Programs

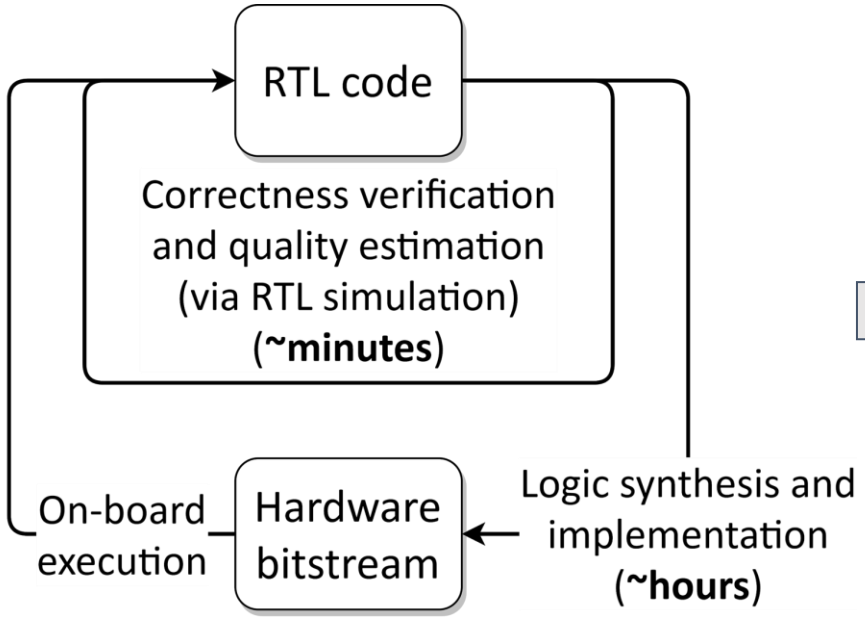
Yuze Chi*, Licheng Guo*, Jason Lau*, Young-kyu Choi*†, Jie Wang*, Jason Cong*

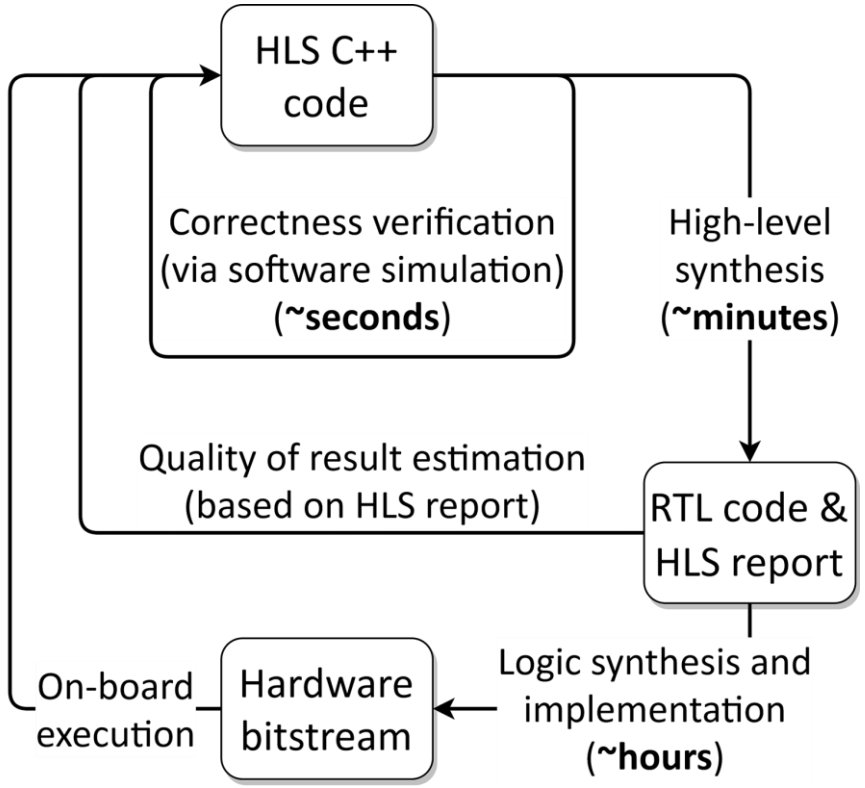
**University of California, Los Angeles, †Inha University*

{chiyuze,cong}@cs.ucla.edu

Widespread Adoption of High-Level Synthesis

- Rapid development cycle: Productivity!





- Competitive quality of result
  - SDC scheduling [DAC'06, TCAD'11], frequency optimizations [DAC'20, FPGA'21]

[DAC'06]: Cong and Zhang. An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation. In DAC, 2006.
[TCAD'11]: Cong et al. High-Level Synthesis for FPGAs: From Prototyping to Deployment. TCAD, 2011.
[DAC'20]: Guo et al. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In DAC, 2020.
[FPGA'21]: Guo et al. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In FPGA, 2021.

Limitations of Current High-Level

```
void Gemm(float a[N][N], float b[N][N],
          float c[N][N]) {
  for (int i = 0; i < N; ++i) {
    for (int k = 0; k < N; ++k) {
#pragma HLS pipeline II = 1
      for (int j = 0; j < N; ++j) {
#pragma HLS unroll
        c[i][k] += a[i][j] * b[j][k];
      }
    }
  }
}
```

```
void Gemm(float a[N][N], float b[N][N],
          float c[N][N]) {
  hls::stream<float> a_00, a_01, ...
  hls::stream<float> b_00, b_01, ...
  hls::stream<float> c_00, c_01, ...
  hls::stream<float> fifo_00_10, ...
#pragma HLS dataflow
  Load(a, a_00, a_01, a_10, a_11);
  Load(b, b_00, b_01, b_10, b_11);
  Compute(a_00, b_00, c_00, ...);
  Compute(a_01, b_01, c_01, ...);
  Compute(a_10, b_10, c_10, ...);
  Compute(a_11, b_11, c_11, ...);
  Store(c, c_00, c_01, c_10, c_11);
}
```

Why Go Task-Parallel?

- Data parallelism may be limited
  - PageRank / graph convolutional networks / on-chip network switching
- Task parallelism can better accommodate variable DRAM latency
- Task parallelism shows better scalability
  - Data-parallel applications can be implemented as task-parallel programs
    - Systolic arrays [ICCAD'18, PolySA]
    - Stencil [ICCAD'18, SODA]

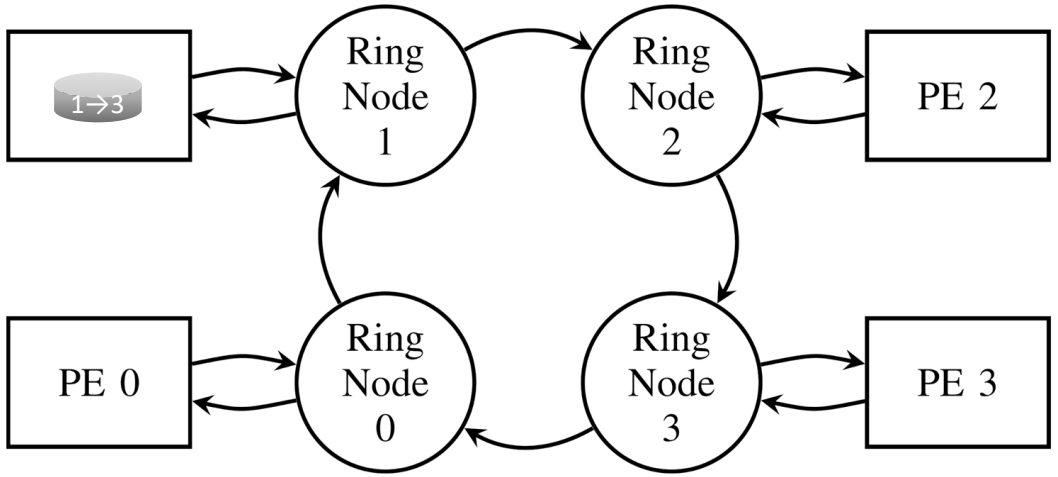
[ICCAD'18]: Cong and Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In ICCAD, 2018.
[ICCAD'18]: Chi et al. SODA: Stencil with Optimized Dataflow Architecture. In ICCAD, 2018.

Real-World Task-Parallel Benchmarks

Benchmark	Application	#Tasks	#Task Instances	#Channels
cannon	Cannon's matrix multiplication algorithm	5	91	344
cnn	Systolic array generated by PolySA [ICCAD'18]	14	209	366
gaussian	Stencil generated by SODA [ICCAD'18]	15	564	1602
gcn	Graph convolutional network [ICLR'17]	5	12	25
gemm	Systolic array generated by PolySA [ICCAD'18]	14	207	364
network	8×8 Omega switching network	3	14	32
page_rank	Edge-centric PageRank accelerator	4	18	89

[ICLR'17]: Kipf and Welling. Semi-Supervised Classification with Graph Convolutional Networks. In ICLR, 2017.
[ICCAD'18]: Cong and Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In ICCAD, 2018.
[ICCAD'18]: Chi et al. SODA: Stencil with Optimized Dataflow Architecture. In ICCAD, 2018.

A Motivating Example

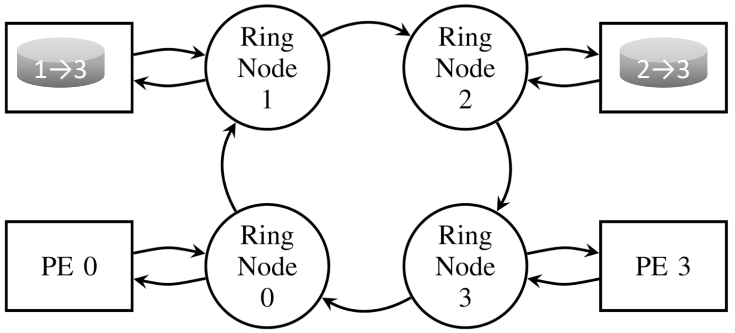


- 4 PEs interconnected w/ a ring
  - PEs send packets to ring nodes & receive packets from ring nodes
  - Ring nodes forward packets *conditionally* based on the destination PE specified in the packet header

Extending Kernel Programming Interfaces

```
struct PktEoT {
  Pkt pkt;
  bool eot;
};
void RingNode(stream<Pkt>& node_in,  stream<PktEoT>& pe_in,
              stream<Pkt>& node_out, stream<Pkt>& pe_out) {
  Pkt node_pkt;
  bool node_pkt_valid = false;
  PktEoT pe_pkt;
  bool pe_pkt_valid = false;
  while (!(pe_pkt_valid && pe_pkt.eot)) {
    if (!pe_pkt_valid)
      pe_pkt_valid = pe_in.read_nb(pe_pkt);
    if (!node_pkt_valid)
      node_pkt_valid = node_in.read_nb(node_pkt);
    if (pe_pkt_valid) {
      node_out.write(pe_pkt.pkt);
      pe_pkt_valid = false;
      if (node_pkt_valid && IsForThisNode(node_pkt)) {
        pe_out.write(node_pkt);
        node_pkt_valid = false;
      }
    } else if (node_pkt_valid) {
      Pkt pkt = node_pkt;
      node_pkt_valid = false;
      (IsForThisNode(pkt) ? pe_out : node_out).write(pkt);
    }
  }  // Highlighted are destructive read operations and
}    // non-destructive read (peek) operations.
```

Kernel code w/ Vivado HLS




```
void RingNode(istream<Pkt>& node_in,  istream<Pkt>& pe_in,
              ostream<Pkt>& node_out, ostream<Pkt>& pe_out) {
  while (!pe_in.eot()) {
    if (!pe_in.empty()) {
      node_out.write(pe_in.read());
      if (!node_in.empty() && IsForThisNode(node_in.peek()))
        pe_out.write(node_in.read());
    } else if (!node_in.empty()) {
      Pkt pkt = node_in.read();
      (IsForThisNode(pkt) ? pe_out : node_out).write(pkt);
    }
  }  // Highlighted are destructive read operations and
}    // non-destructive read (peek) operations.
```

Equivalent kernel code w/ TAPA

For kernel code, TAPA is much shorter to write with peeking and transaction API support.

Simplifying Host-Kernel Communication Interfaces

```
auto devices = xcl::get_xil_devices();
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
for (int i = 0; i < devices.size(); i++) {
  auto device = devices[i];
  OCL_CHECK(err, context = cl::Context(...));
  OCL_CHECK(err, q = cl::CommandQueue(...));
  cl::Program program(...);
  if (err == CL_SUCCESS) {
    OCL_CHECK(err, kernel = cl::Kernel(...));
    break;
  }
}
OCL_CHECK(err, cl::Buffer buffer1(...));
OCL_CHECK(err, cl::Buffer buffer2(...));
...
OCL_CHECK(err, err = kernel.setArg(0, buffer1));
OCL_CHECK(err, err = kernel.setArg(1, buffer2));
OCL_CHECK(err, err = kernel.setArg(2, size));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects(...));
OCL_CHECK(err, err = q.enqueueTask(kernel));
OCL_CHECK(err, err = q.enqueueMigrateMemObjects(...);
q.finish();
```

Host code for Vivado HLS code as an OpenCL kernel

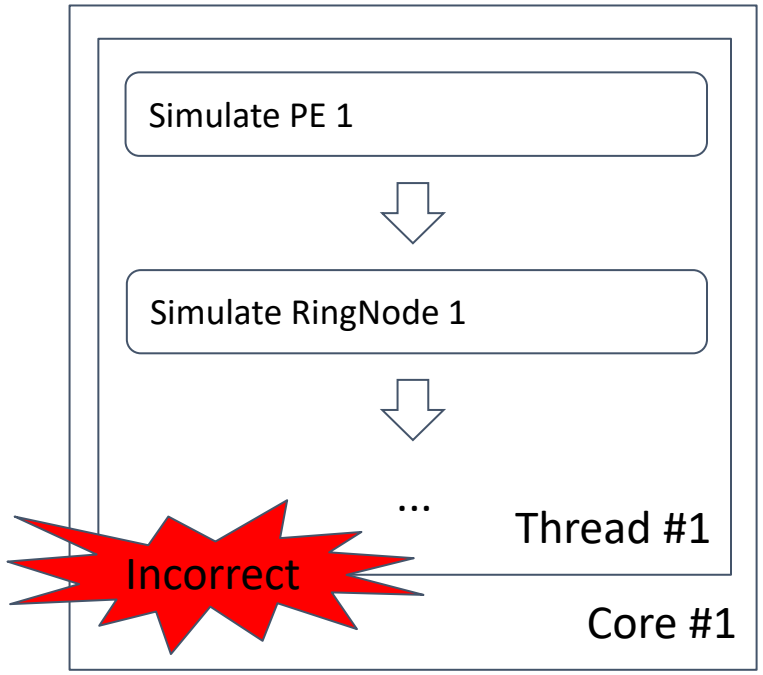
```
tapa::invoke(
    kernel,
    path_to_bitstream,
    tapa::read_only_mmap<int>(buffer1),
    tapa::write_only_mmap<int>(buffer2),
    size);
```

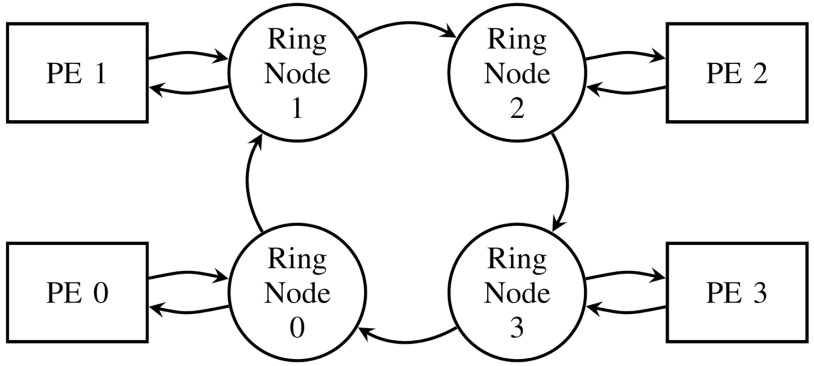
Equivalent host code for TAPA

For host code, TAPA is also much shorter to write by allowing programmers call the kernel using a single function call.

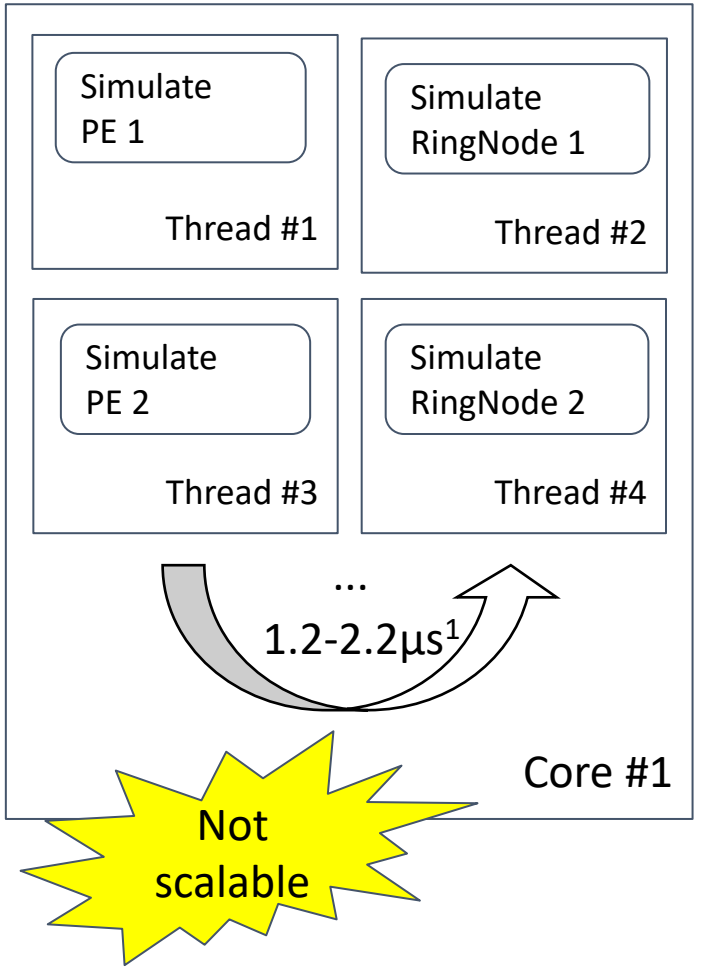
Software Simulation

Sequential simulator

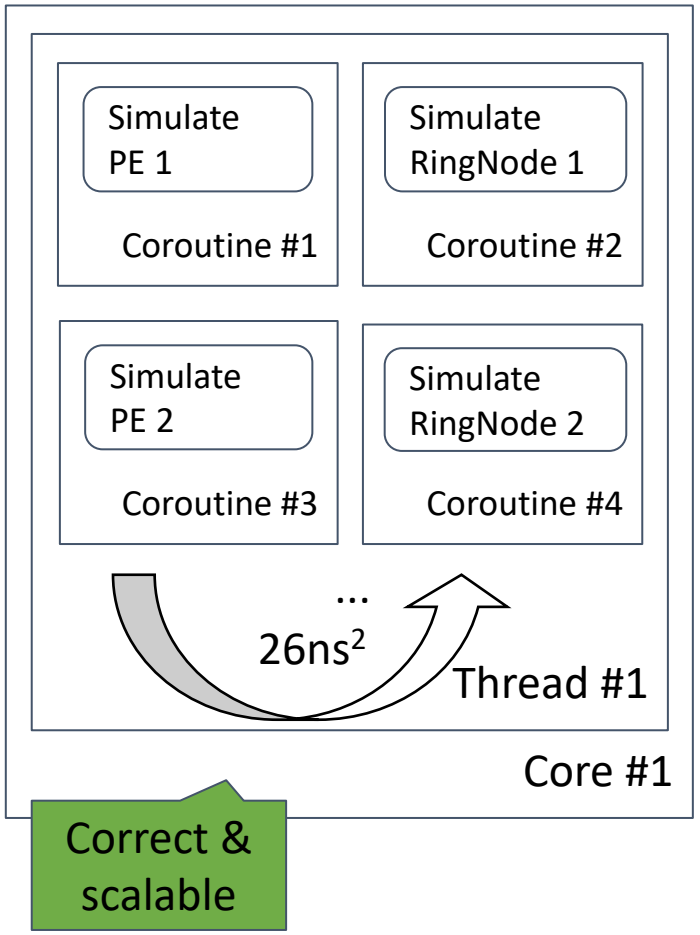




Multi-thread simulator

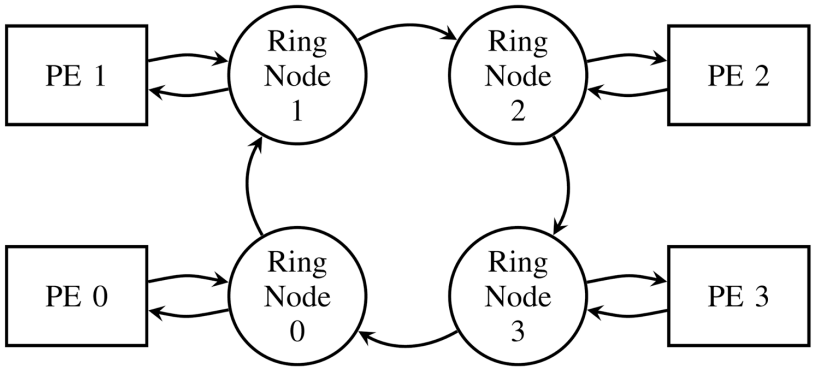


TAPA's coroutine-based simulator



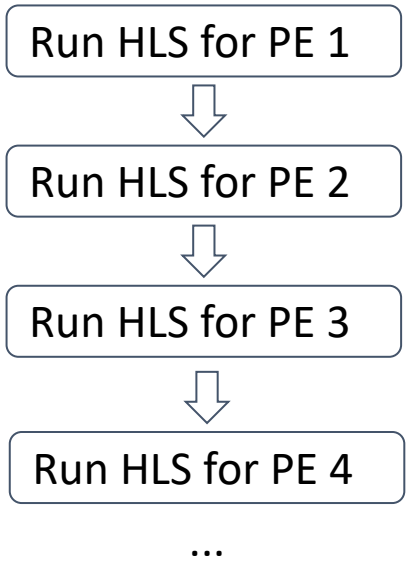
[1]: https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/
[2]: https://www.boost.org/doc/libs/1_73_0/libs/coroutine2/doc/html/coroutine2/performance.html

RTL Code Generation



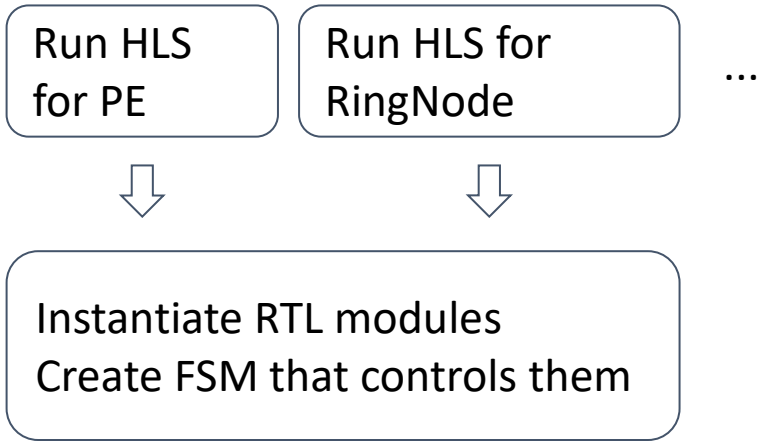
All PEs are instances of the same task (C++ function). All ring nodes are also instances of the same task.

Existing flow



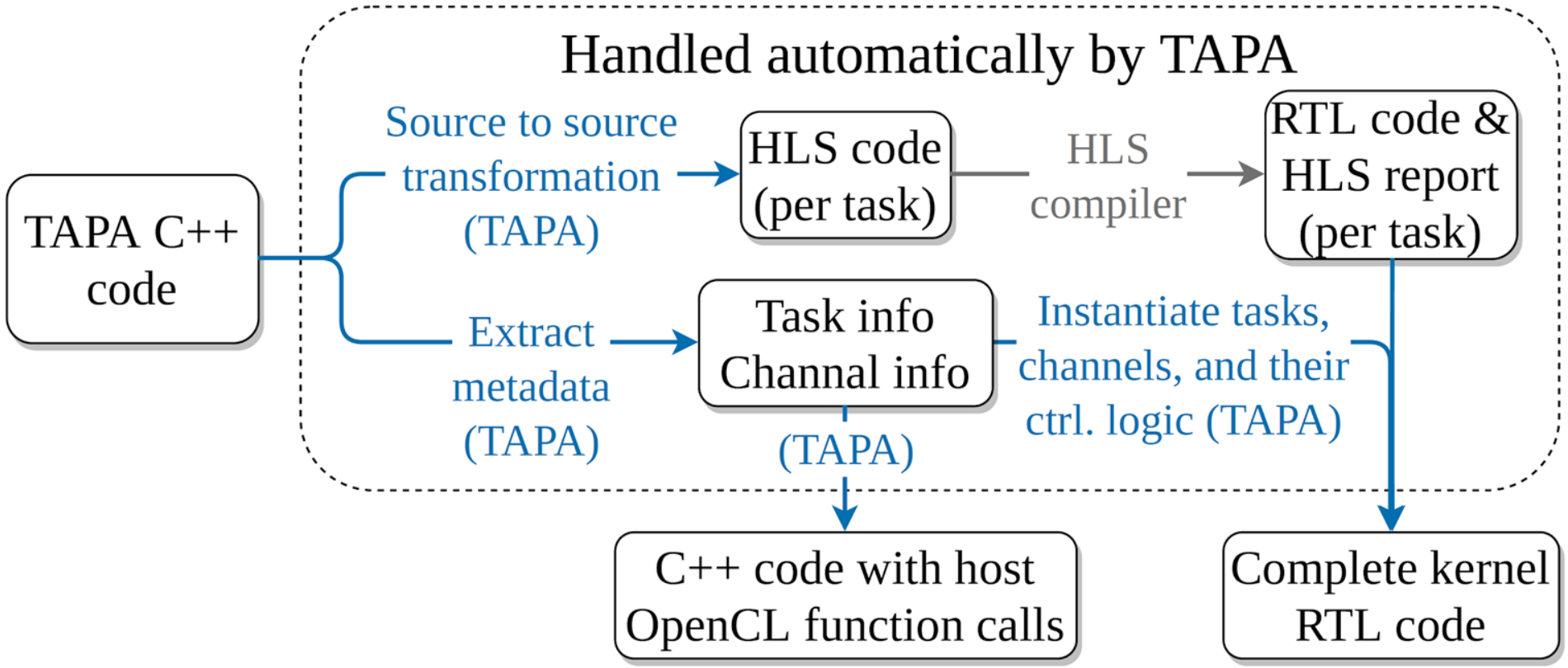
Unnecessary work

TAPA's hierarchical approach

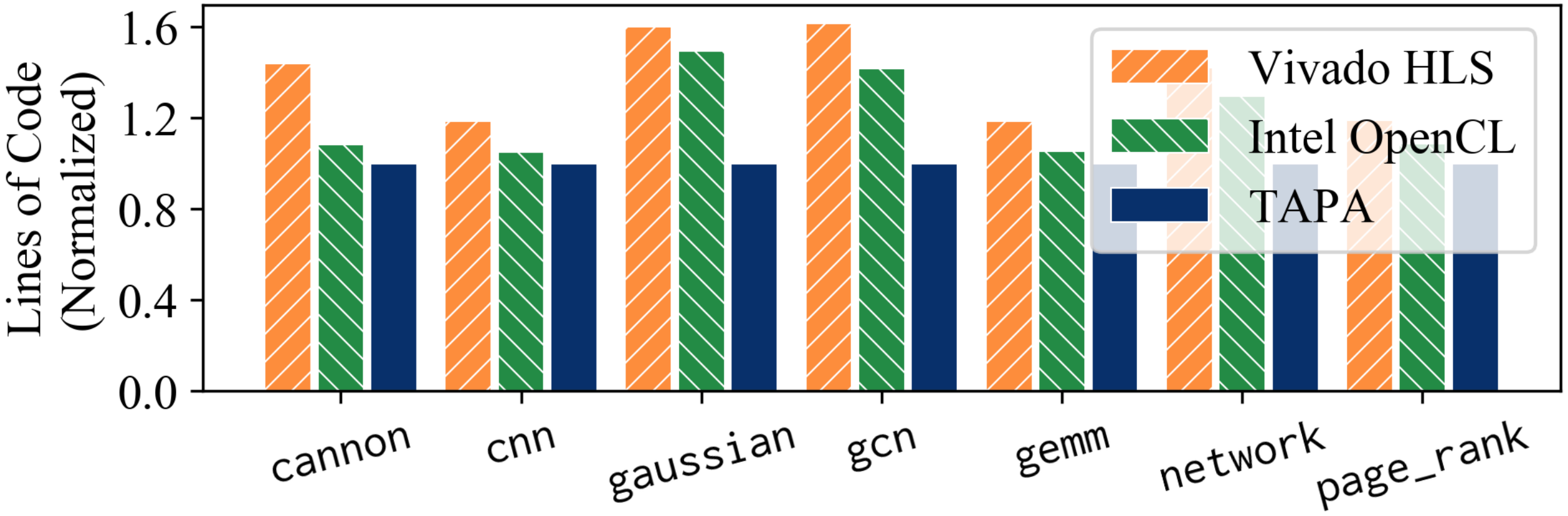


No redundant work!

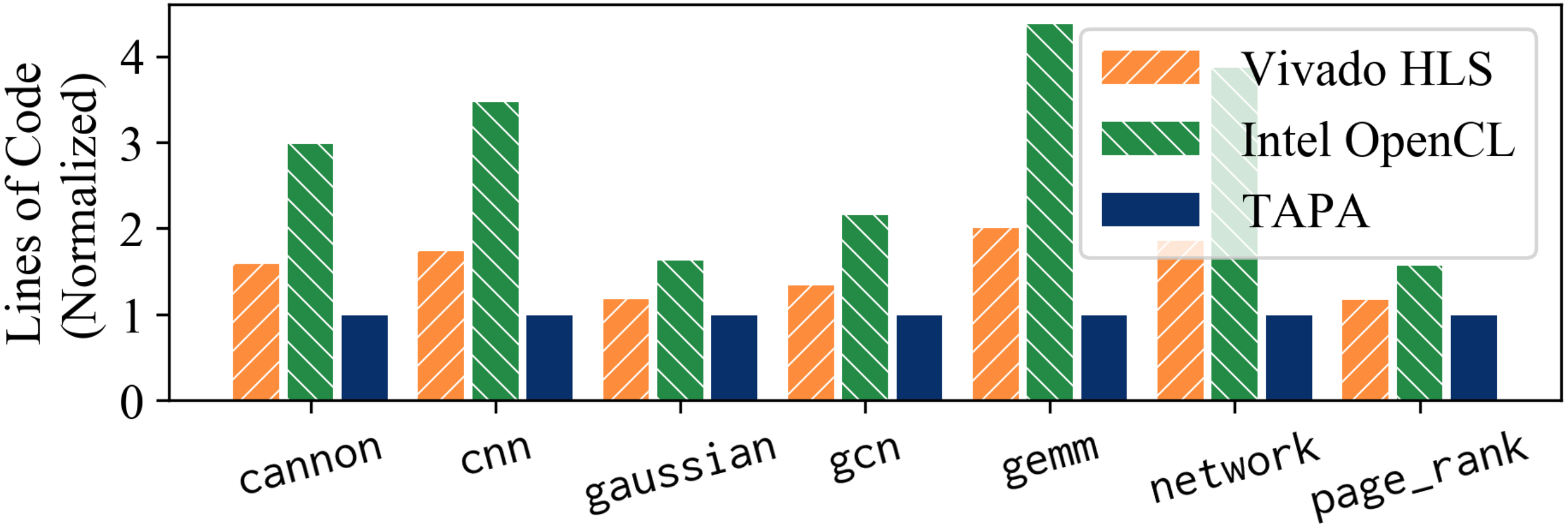
Overall TAPA Workflow



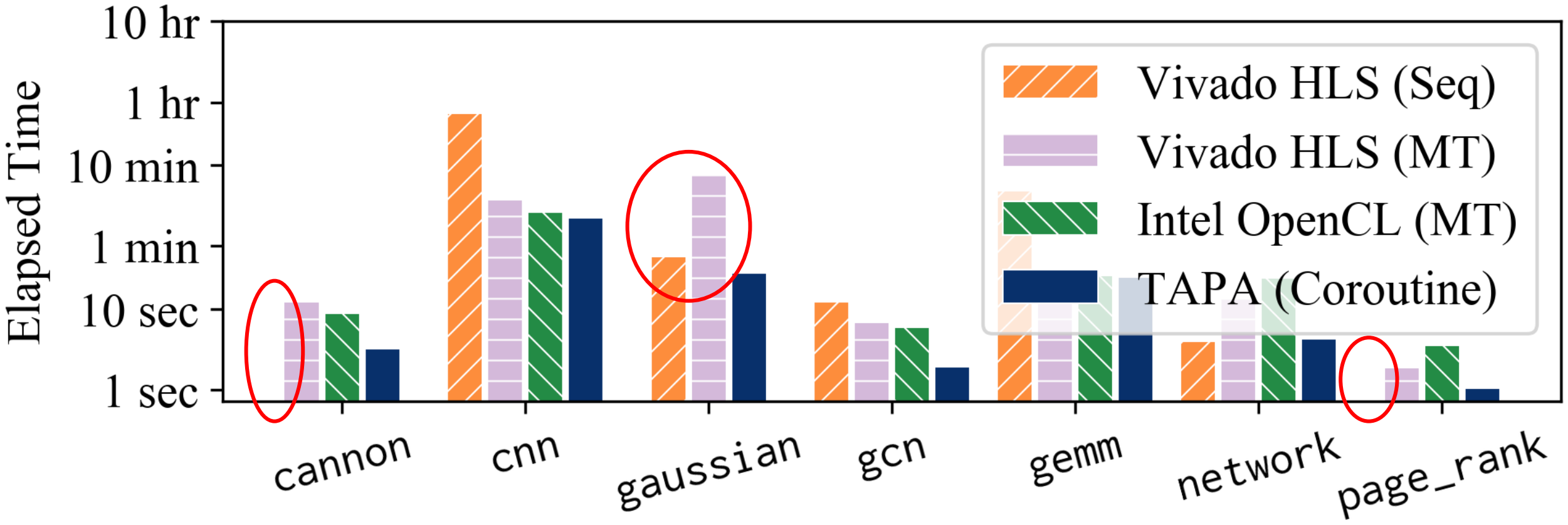
Kernel LoC Reduction



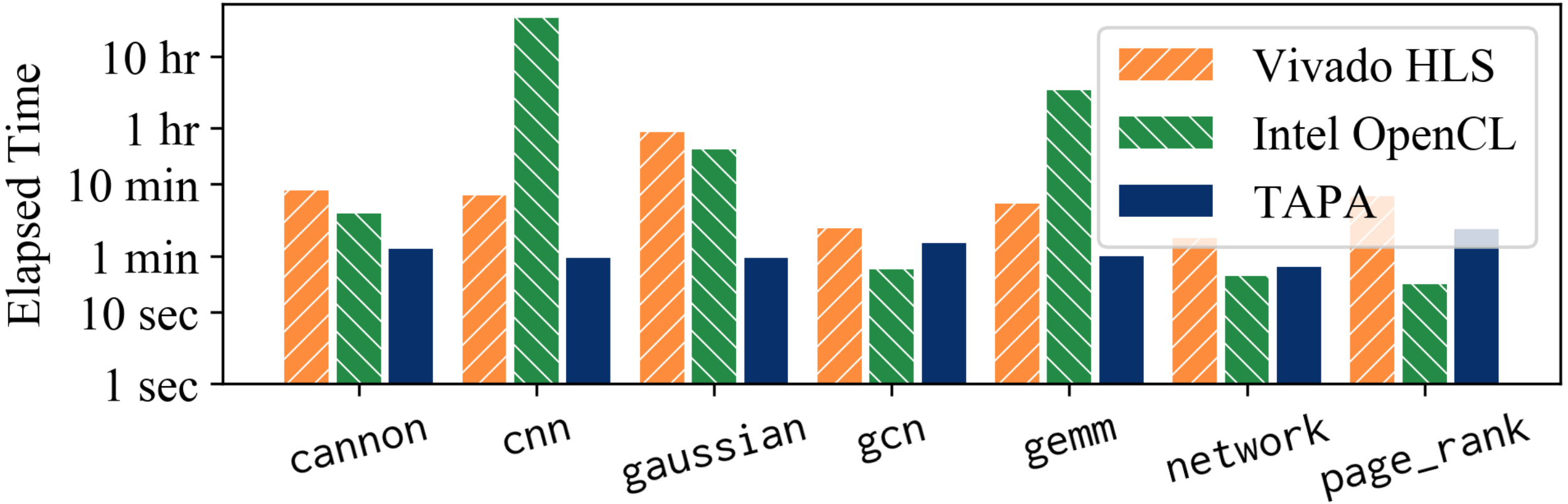
Host LoC Reduction



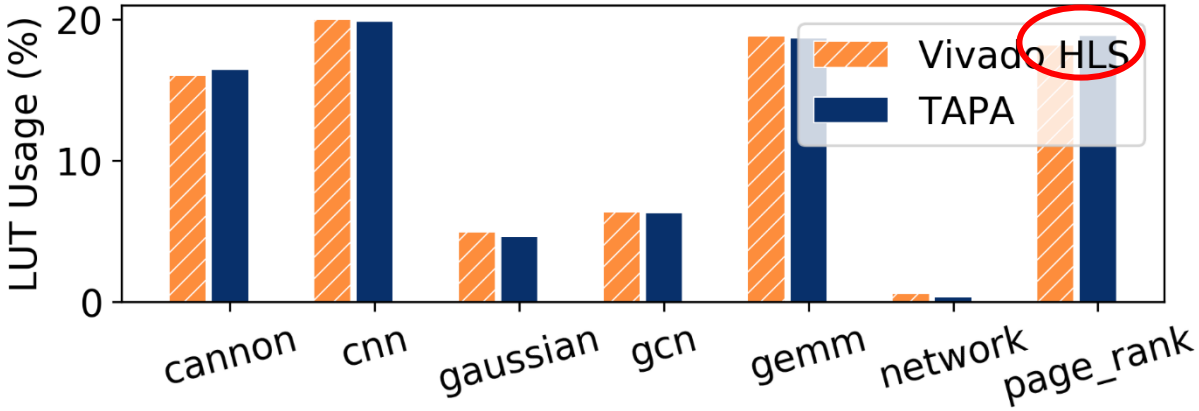
Simulation Time Reduction

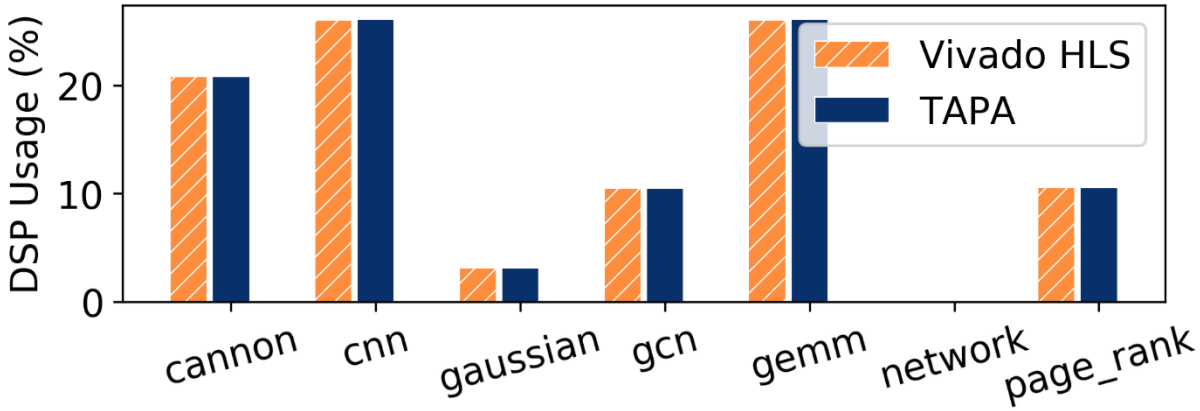


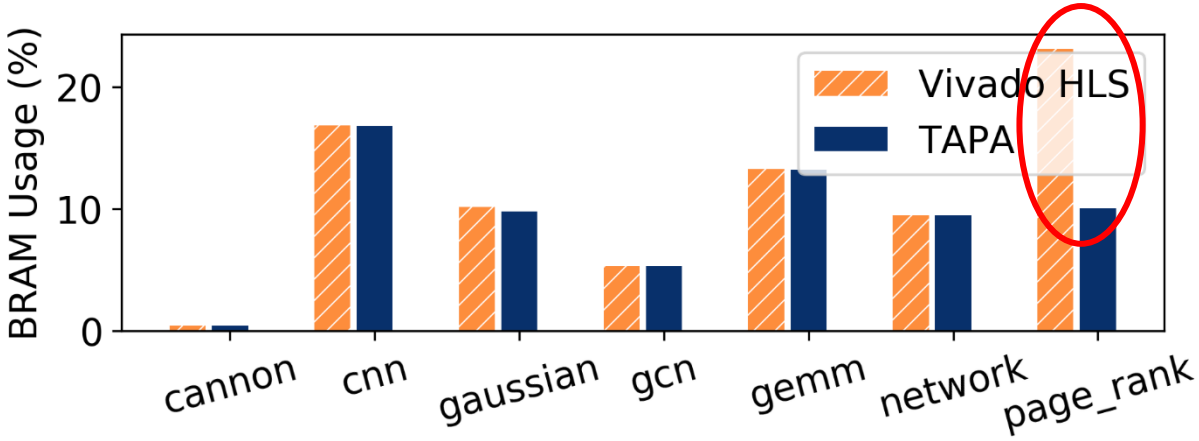
Code Generation Time Reduction

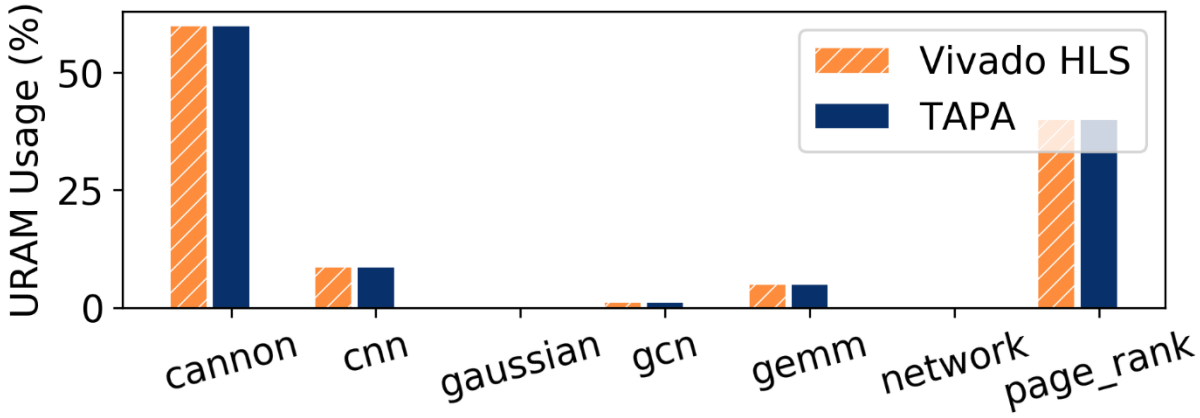


Resource Usage Comparison









Beyond Productivity Enhancement

- Extended AXI4 memory-mapped interface
  - Asynchronous access w/ separate AR/R/AW/W/B channels
  - Shared access among multiple task instances
- Detached tasks/smarter FIFO templates/AutoBridge integration/…
- For more details
  - TAPA is open-source: https://github.com/UCLA-VAST/tapa

Q&A

This work is partially supported by a Google Faculty Award, the NSF RTML program (CCF1937599), NIH Brain Initiative (U01MH117079), the Xilinx Adaptive Compute Clusters (XACC) program, and CRISP, one of six JUMP centers.