TAPA: Fast, High-Frequency, Expressive Dataflow HLS Framework

Licheng Guo, Yuze Chi, Jason Lau, Jianyi Cheng, Linghao Song, Weikang Qiao, Zhenyuan Ruan, Zhiru Zhang, Jason Cong

UCLA, Google, Imperial College London, Cornell University

https://github.com/UCLA-VAST/tapa

https://tapa.rtfd.io/

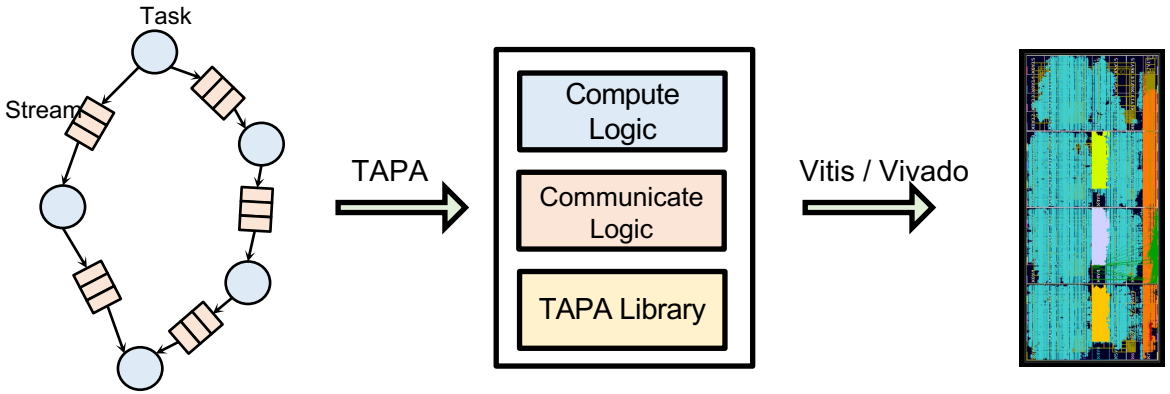
{lcguo, chiyuze, lau}@cs.ucla.edu

TAPA User Group

TAPA Docs

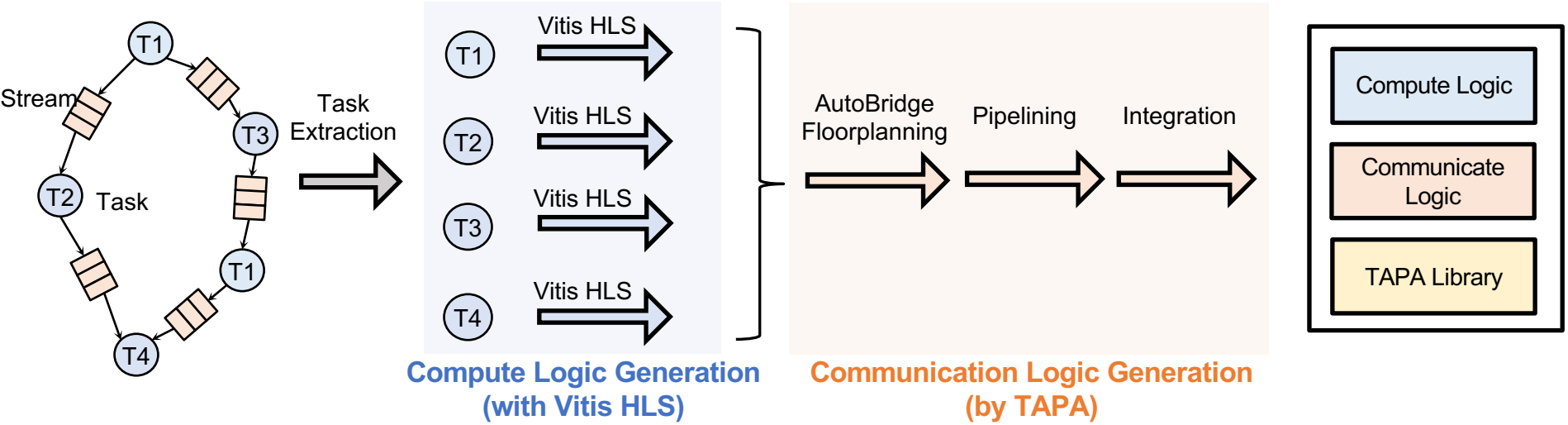
Overview

- TAPA is a dataflow HLS framework
- Input: a dataflow program in C++
- Output: Optimized RTL + Vivado configurations



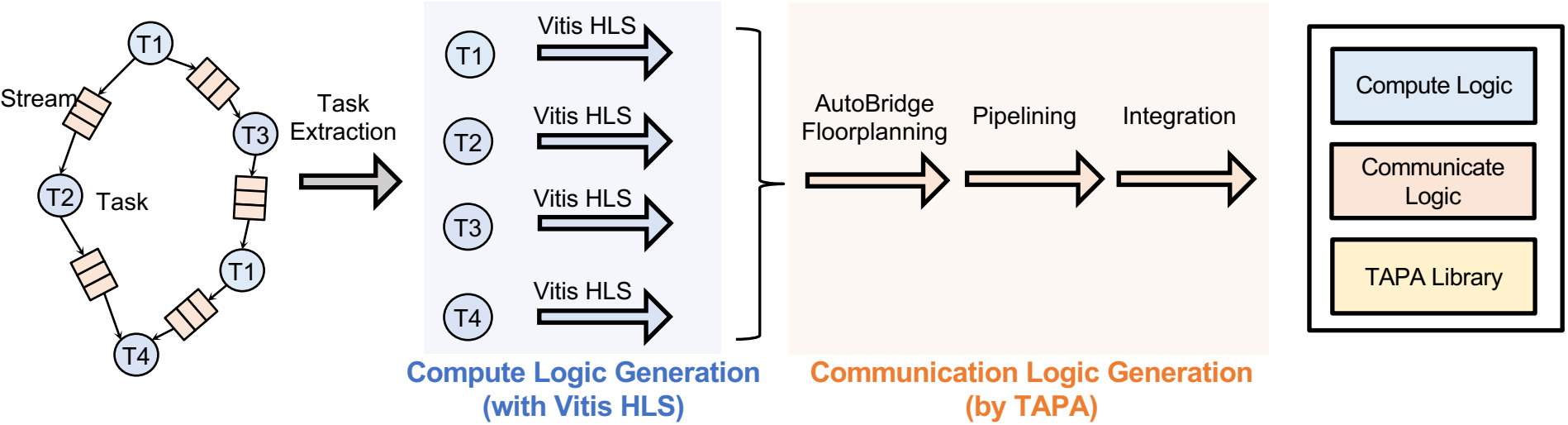
Overview

- TAPA programs explicitly decouple communication and computation
- Computation => compiled by Vitis HLS
- Communication => generated by TAPA



Overview

- Extends existing HLS tools by intelligently compose their outputs
- Communication logic is relatively simple but trick to get it right
- Bypass the limitation that commercial HLS tools are not open-sourced



Advantages of Decoupled Compute/Comm.

- High frequency:
  - Integrate C-level floorplanning and pipelining.
  - Avg. 2X frequency boost[1] (from 147 to 297 MHz on avg.) compared to Vitis HLS.
  - HBM-specific optimization
- Rich expressiveness:
  - Explicit control over parallelism
  - Dedicated APIs for flexible memory access
- Fast:
  - Parallel C-to-RTL compilation, 7X faster[2] on average than Vitis HLS
  - Lightweight simulation, 8X faster[2] on average than Vitis HLS

[1] AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs, FPGA 2021
[2] Extending High-Level Synthesis for Task-Parallel Programs, FCCM 2021

Some History

- 2018
  - Limitations of coupled comp/comm compilation identified & discussed internally
  - Domain-specific solutions to decouple the compilation of comp/comm in [STAccel, FCCM'18] by Zhenyuan and the RTL backend of [SODA, ICCAD'18] by Yuze
- 2020
  - Yuze generalized the solution and built an internal tool
- 2021
  - The tool was published with the name "TAPA" [FCCM'21]
  - Licheng added backend optimization into the framework for timing optimization [AutoBridge, FPGA'21]
- 2022
  - Multiple internal projects used TAPA [FPGA'21, FPGA'22, FPGA'22, DAC'22]
  - First external TAPA user from Zhenman's group at SFU, projects in submission
  - Currently, Jason Lau is leading the extension of TAPA for Versal
  - Next, we will integrate our [RapidStream, FPGA'22] project into TAPA

TAPA Publications

Original Publication:

- [FCCM'18] ST-Accel: A high-level programming platform for streaming applications on FPGA
- [FCCM'21] Extending High-Level Synthesis for Task-Parallel Programs
- [FPGA'21] AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs (**Best Paper Award**)
- [FPGA'22] RapidStream: Parallel Physical Implementation of FPGA HLS Designs (**Best Paper Award**)

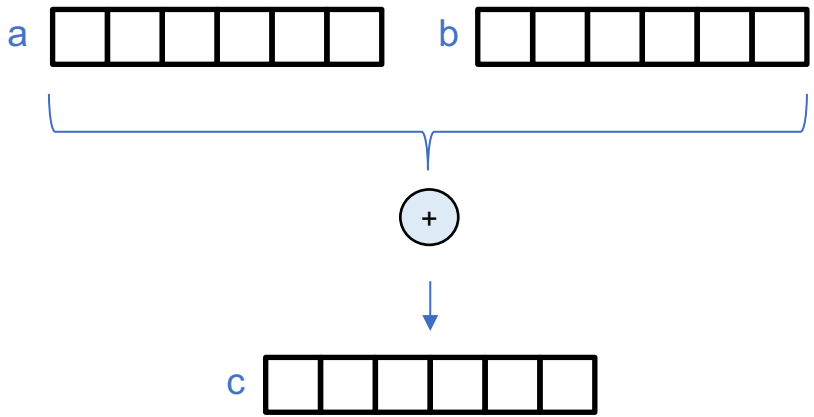
Successful Application:

- [FPGA'21] AutoSA: A polyhedral compiler for high-performance systolic arrays on fpga
- [FPGA'22] Accelerating SSSP for Power-Law Graphs
- [FPGA'22] Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication
- [DAC'22] Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication
- ……

Part 0: Hello World

Vector Add Example

- The code adds up two vectors of `floating` numbers, `a` and `b`
- produces a new vector `c` with the same length




```
#include <cstdint>

#include <tapa.h>

void Add(tapa::istream<float>& a, tapa::istream<float>& b,
         tapa::ostream<float>& c, uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    c << (a.read() + b.read());
  }
}

void Mmap2Stream(tapa::mmap<const float> mmap, uint64_t n,
                 tapa::ostream<float>& stream) {
  for (uint64_t i = 0; i < n; ++i) {
    stream << mmap[i];
  }
}

void Stream2Mmap(tapa::istream<float>& stream, tapa::mmap<float> mmap,
                 uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    stream >> mmap[i];
  }
}

void VecAdd(tapa::mmap<const float> a, tapa::mmap<const float> b,
            tapa::mmap<float> c, uint64_t n) {
  tapa::stream<float> a_q("a");
  tapa::stream<float> b_q("b");
  tapa::stream<float> c_q("c");

  tapa::task()
      .invoke(Mmap2Stream, a, n, a_q)
      .invoke(Mmap2Stream, b, n, b_q)
      .invoke(Add, a_q, b_q, c_q, n)
      .invoke(Stream2Mmap, c_q, c, n);
}
```

Vector Add Example

- We define three tasks:

```
void Add(tapa::istream<float>& a, tapa::istream<float>& b,
         tapa::ostream<float>& c, uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    c << (a.read() + b.read());
  }
}
```

- read from FIFO `a`
- read from FIFO `b`
- add them up
- write to FIFO `c`

```
void Mmap2Stream(tapa::mmap<const float> mmap, uint64_t n,
                 tapa::ostream<float>& stream) {
  for (uint64_t i = 0; i < n; ++i) {
    stream << mmap[i];
  }
}
```

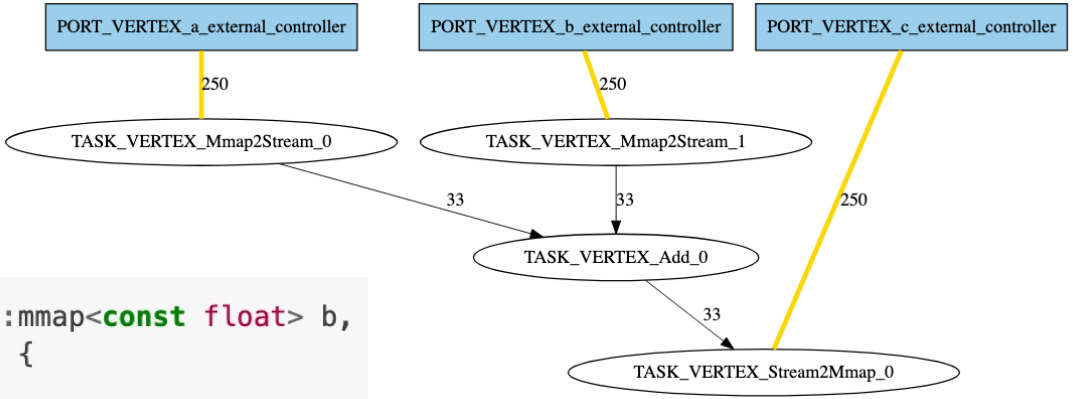
- read from external memory `mmap`
- write into FIFO `stream`

```
void Stream2Mmap(tapa::istream<float>& stream, tapa::mmap<float> mmap,
                 uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    stream >> mmap[i];
  }
}
```

- read from FIFO `stream`
- write to external memory `mmap`

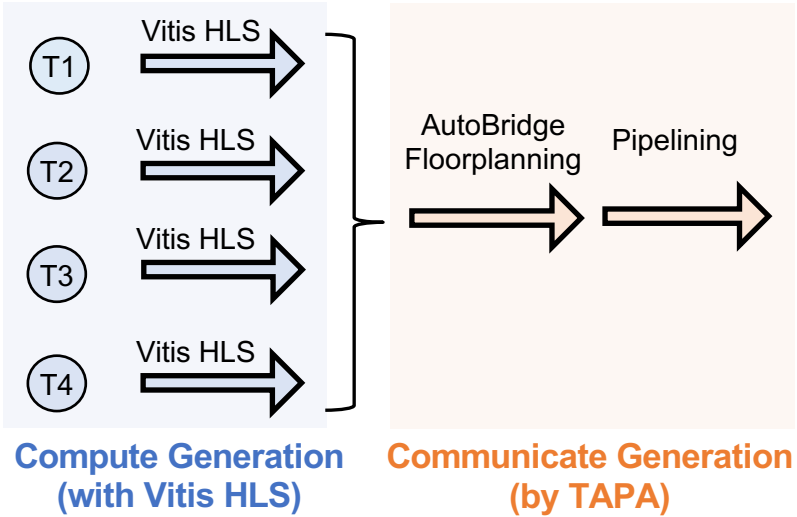
Vector Add Example

- Define how the tasks are connected




```
void VecAdd(tapa::mmap<const float> a, tapa::mmap<const float> b,
            tapa::mmap<float> c, uint64_t n) {
  tapa::stream<float> a_q("a");
  tapa::stream<float> b_q("b");
  tapa::stream<float> c_q("c");

  tapa::task()
      .invoke(Mmap2Stream, a, n, a_q)
      .invoke(Mmap2Stream, b, n, b_q)
      .invoke(Add, a_q, b_q, c_q, n)
      .invoke(Stream2Mmap, c_q, c, n);
}
```

Vector Add Example



RTL generated by Vitis HLS

```
void Add(tapa::istream<float>& a, tapa::istream<float>& b,
         tapa::ostream<float>& c, uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    c << (a.read() + b.read());
  }
}

void Mmap2Stream(tapa::mmap<const float> mmap, uint64_t n,
                 tapa::ostream<float>& stream) {
  for (uint64_t i = 0; i < n; ++i) {
    stream << mmap[i];
  }
}

void Stream2Mmap(tapa::istream<float>& stream, tapa::mmap<float> mmap,
                 uint64_t n) {
  for (uint64_t i = 0; i < n; ++i) {
    stream >> mmap[i];
  }
}
```

RTL generated by TAPA

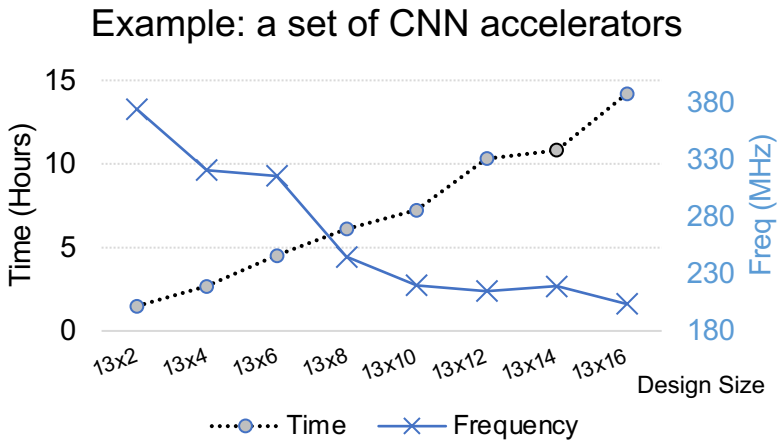
```
void VecAdd(tapa::mmap<const float> a, tapa::mmap<const float> b,
            tapa::mmap<float> c, uint64_t n) {
  tapa::stream<float> a_q("a");
  tapa::stream<float> b_q("b");
  tapa::stream<float> c_q("c");

  tapa::task()
      .invoke(Mmap2Stream, a, n, a_q)
      .invoke(Mmap2Stream, b, n, b_q)
      .invoke(Add, a_q, b_q, c_q, n)
      .invoke(Stream2Mmap, c_q, c, n);
}
```

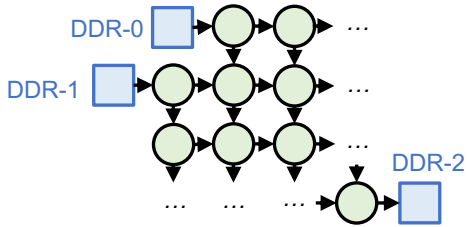
Demo: Hello World

Part 1: High Frequency

Challenges

- Frequency drops as designs become larger





Reason 1: Abstraction Gap

- HLS has no physical layout information
  - How far will these two registers be apart?
  - How congested will the area be?
- Current HLS relies on inaccurate pre-characterized delay models

```
void top() {
  temp = foo(...);
  bar(temp, ...);
}
```

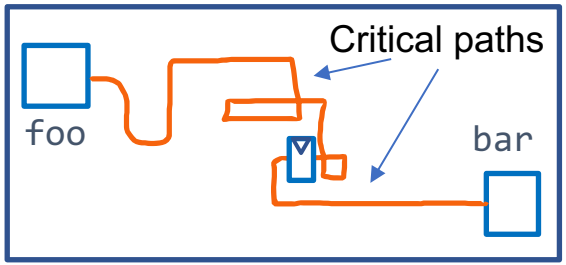
Source C++ code

HLS

```
always @ (posedge ap_clk)
  bar_in <= foo_out;
```

HLS registers the connection once (which looks reasonable)

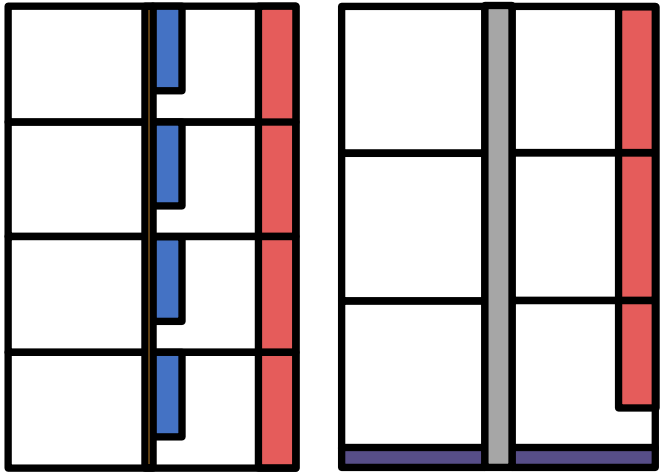
Placer Router



This is possible (and common!)

Reason 2: Increasing FPGA Complexity

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location

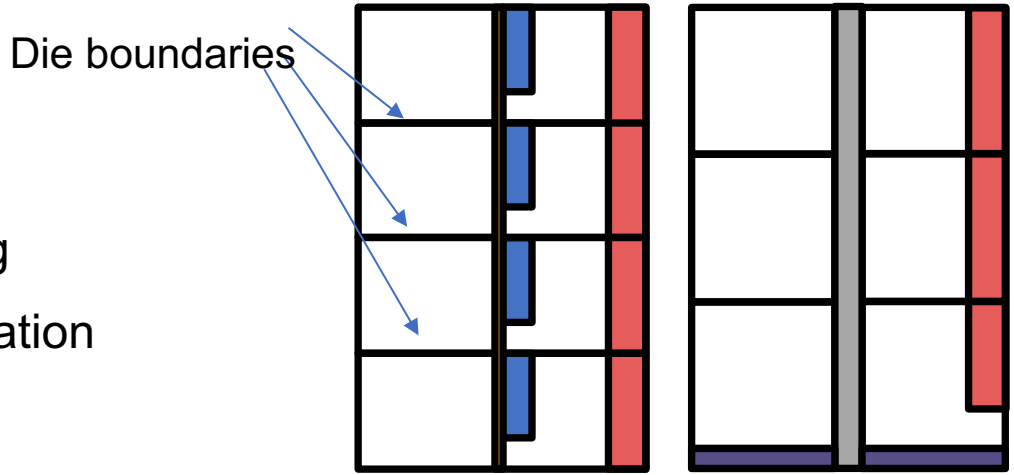


Xilinx Alveo U250

Xilinx Alveo U280

Reason 2: Increasing FPGA Complexity

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location

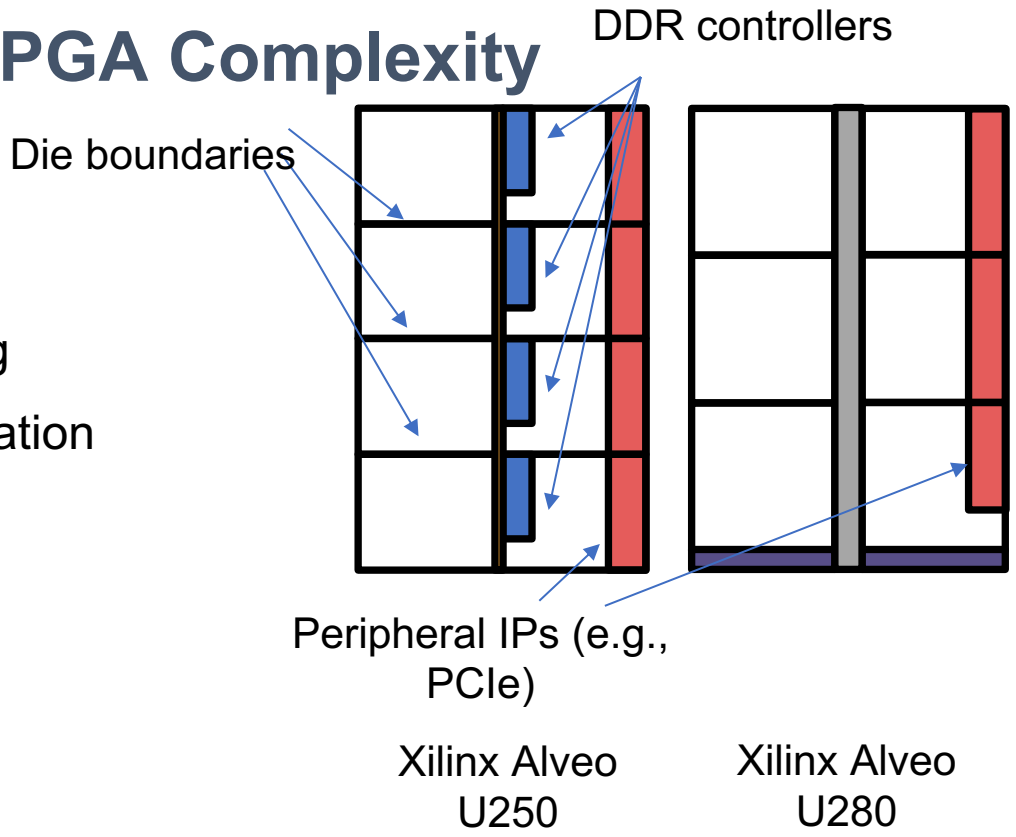


Xilinx Alveo U250

Xilinx Alveo U280

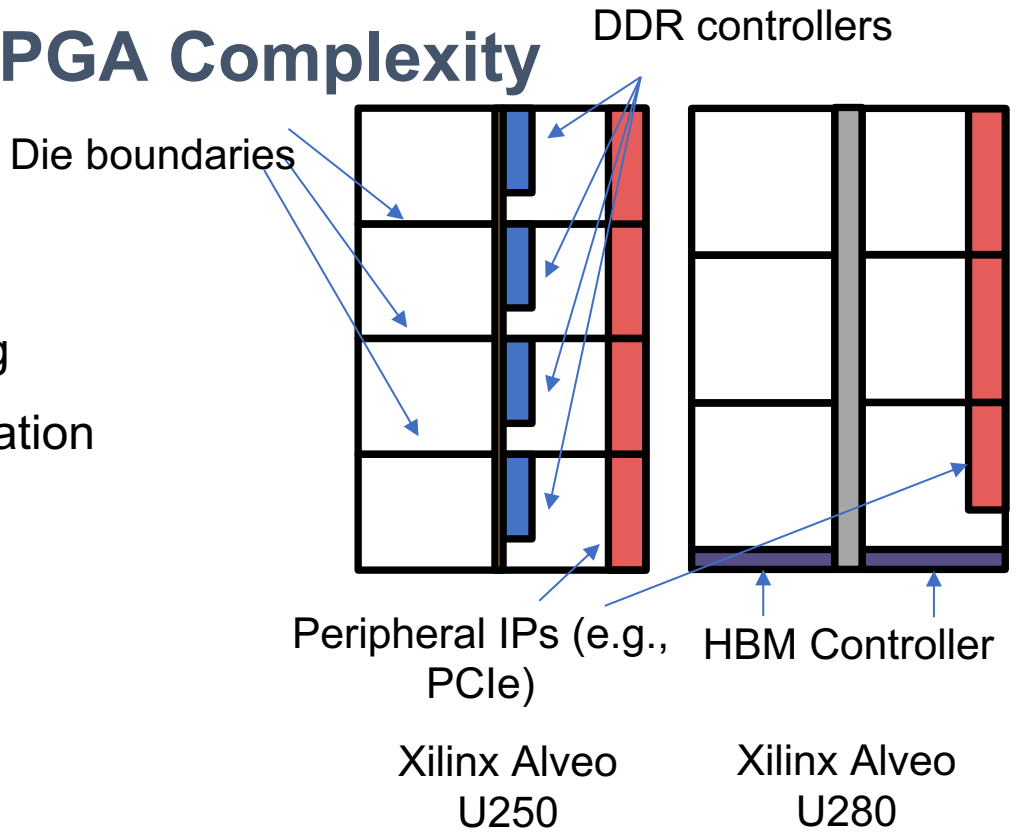
Reason 2: Increasing FPGA Complexity

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location



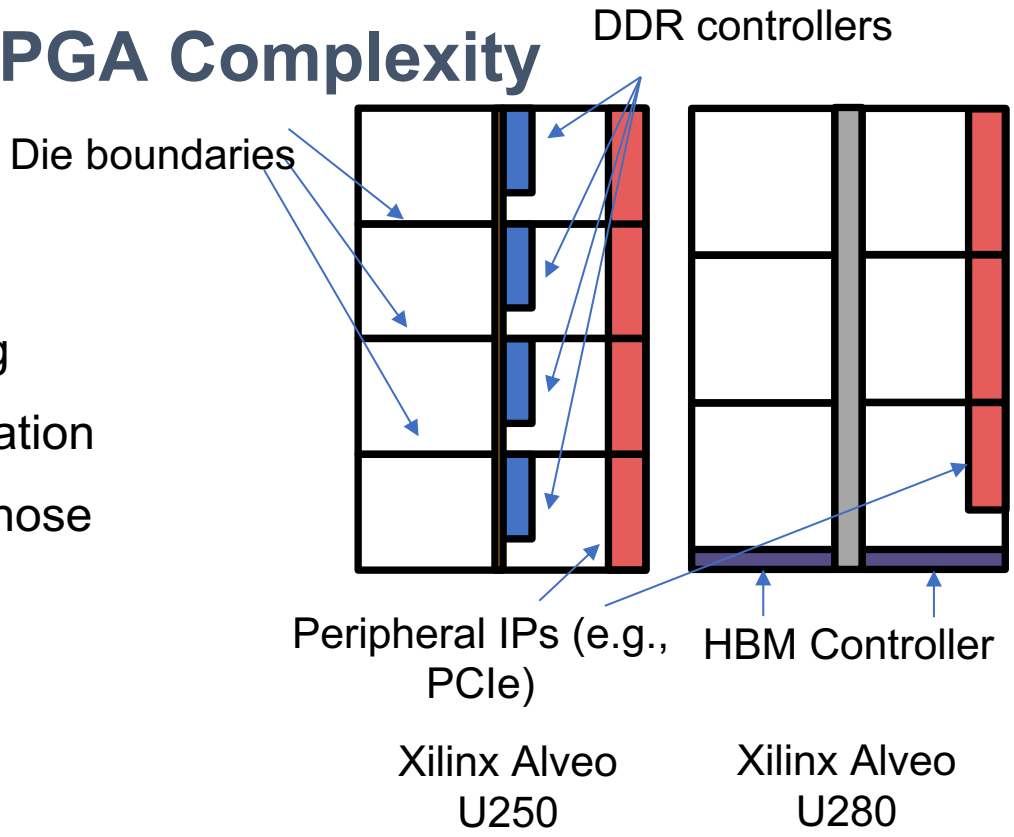
Reason 2: Increasing FPGA Complexity

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location



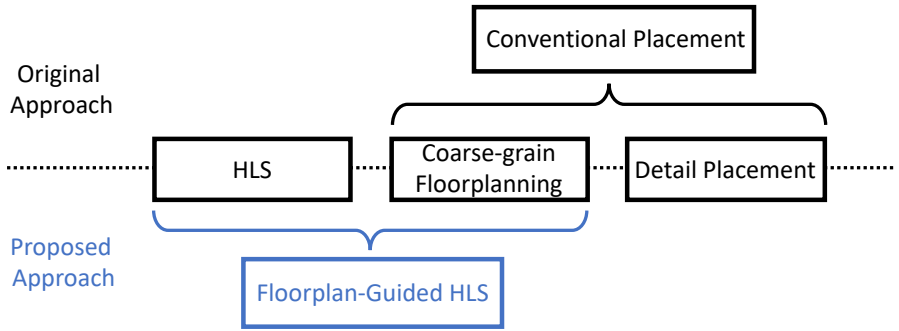
Reason 2: Increasing FPGA Complexity

- FPGAs are increasingly large
- Multiple dies integrated together
- High delay penalty for die-crossing
- Large IPs with pre-determined location
- HLS has limited consideration of those physical barriers



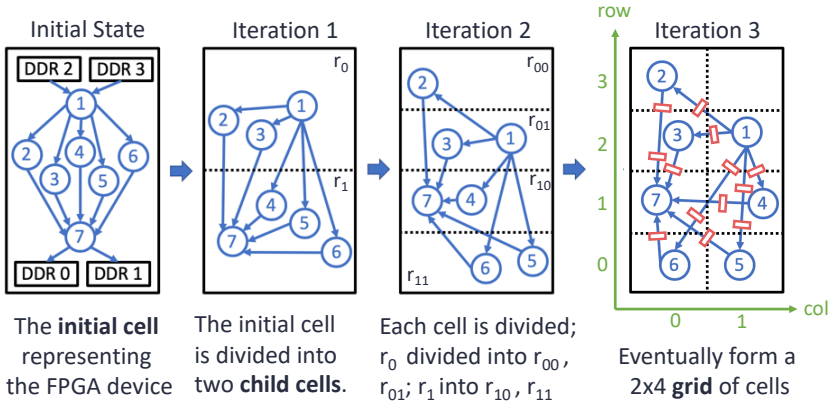
AutoBridge [FPGA'21 Best Paper Award]

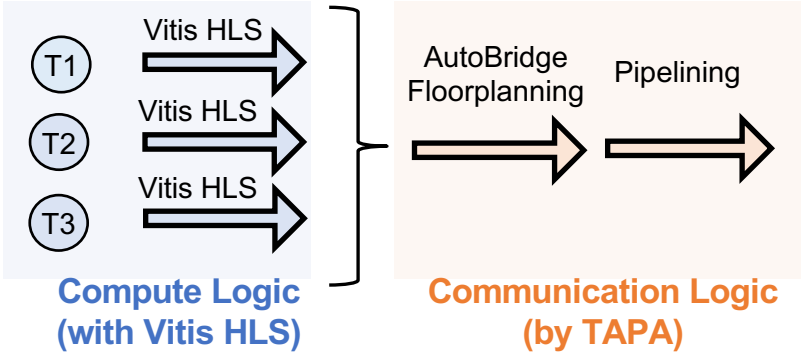
- Couples floorplanning with HLS pipelining
- Distribute the design across the whole device to reduce local congestion
- Pipeline the global interconnects to remove global critical paths



Implementation

- Vits HLS compiles each task => area estimation
- Logically partition the FPGA fabric
- Assign each task instance to one slot
- Based on the floorplanning, determine the pipeline level for each edge
- TAPA generates the pipeline structure to compose all task instances
- TAPA generates scripts to enforce the floorplan in Vivado

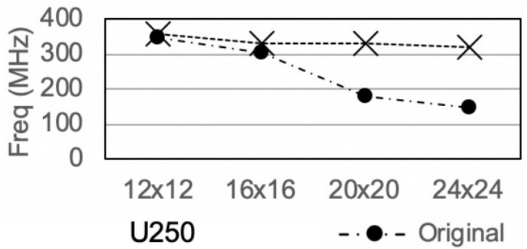




Case Study 1

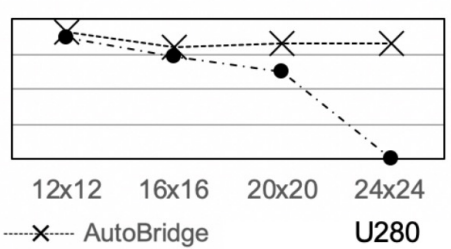
- Gaussian Elimination, 8 configurations

Opt: avg. 334 MHz (1.4X)



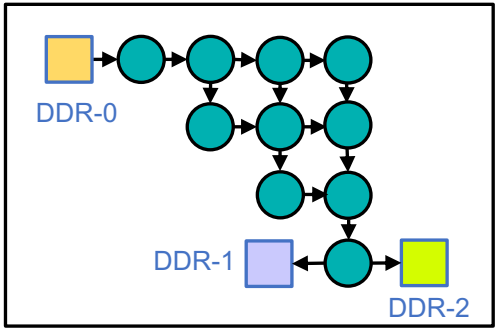
Default: avg. 245 MHz

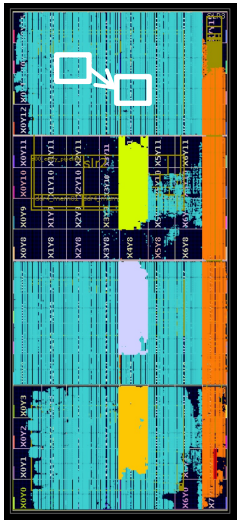
Opt: avg. 335 MHz (1.5X)

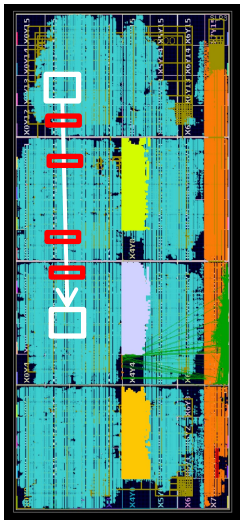


Default: avg. 223 MHz

- Difference in Resource Utilization
  - LUT: -0.14%
  - FF: -0.04%
  - BRAM: -0.03%
  - DSP: +0.00%







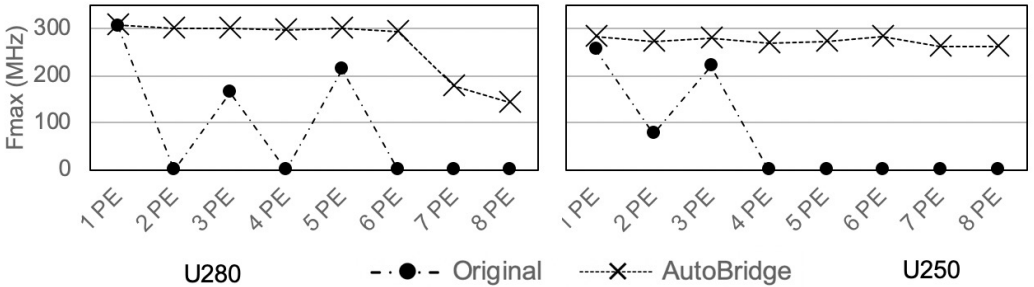
Default AutoBridge

Comparison of the 24x24 Design on U250

Case Study 2

- Stencil Computation, 16 configurations

Opt: avg 266 MHz (3.1X)

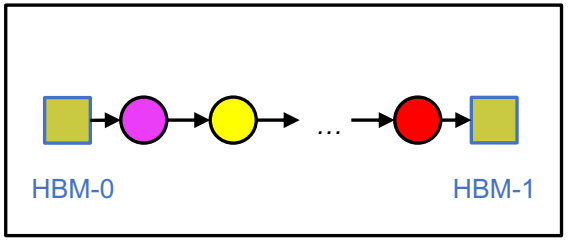


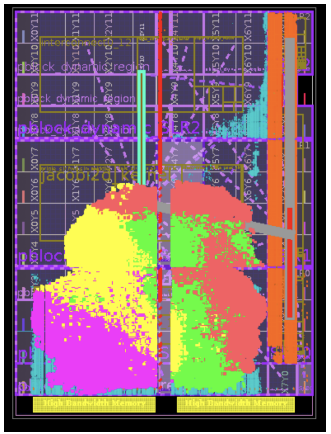
Default: avg. 86 MHz

Opt: avg. 273 MHz (3.9X)

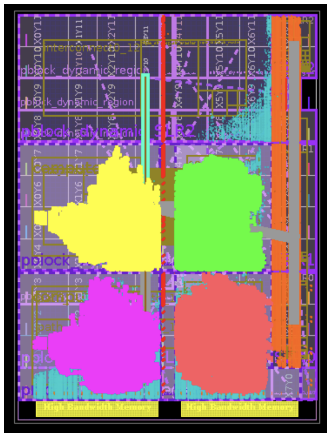
Default: avg. 69 MHz

- Difference in Resource Utilization
  - LUT: -0.26%
  - FF: +0.78%
  - BRAM: +4.68%
  - DSP: +0.00%





Default



AutoBridge

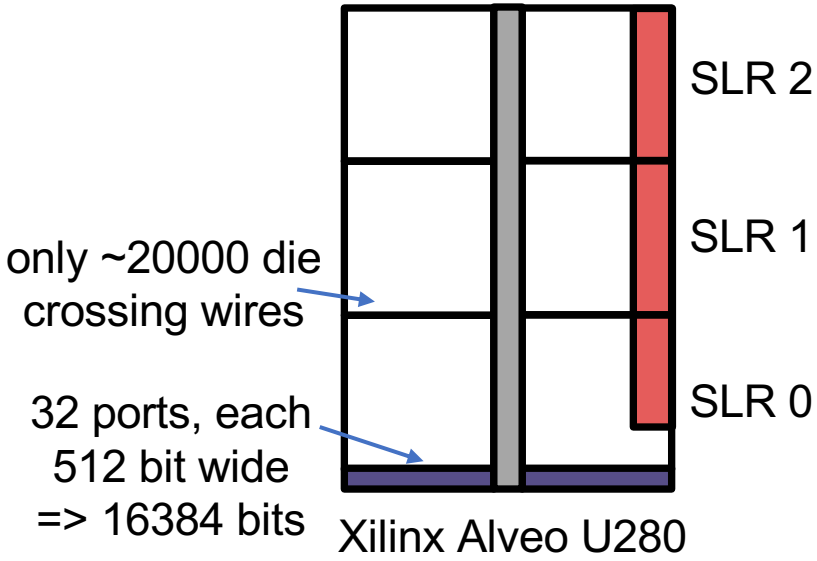
Comparison of the 4-PE Design on U280

Demo: Floorplanning & Pipelining

Part 1.1: HBM-Specific Timing Optimization

Challenges with HBM FPGAs

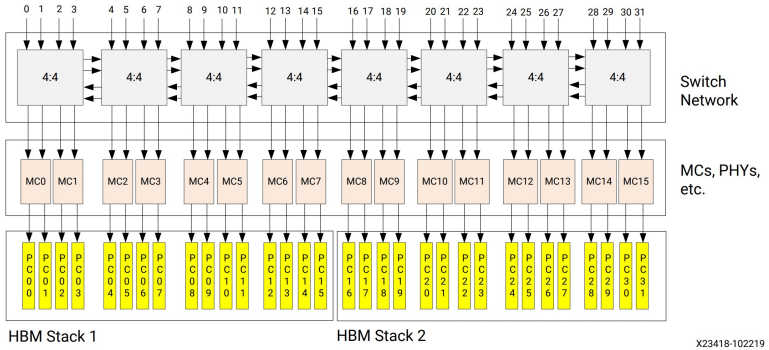
- All 32 channels clustered together
- An interface of 16384 bits in total
- An SLR boundary only has ~20K crossing wires
- Area overhead from HBM IO modules
- Significant resource pressure and routing congestion in the bottom SLR



HBM-Specific Optimization

- Automatic HBM channel binding
- Incorporate the binding selection into the floorplanning process
- Balance the resource utilization on the two sides

How to map virtual buffers to physical channels?




```
[connectivity]
sp=Serpens.edge_list_ch_0:HBM[0]
sp=Serpens.edge_list_ch_1:HBM[1]
sp=Serpens.edge_list_ch_2:HBM[2]
sp=Serpens.edge_list_ch_3:HBM[3]
sp=Serpens.edge_list_ch_4:HBM[4]
sp=Serpens.edge_list_ch_5:HBM[5]
sp=Serpens.edge_list_ch_6:HBM[6]
sp=Serpens.edge_list_ch_7:HBM[7]
sp=Serpens.edge_list_ch_8:HBM[8]
sp=Serpens.edge_list_ch_9:HBM[9]
sp=Serpens.edge_list_ch_10:HBM[10]
sp=Serpens.edge_list_ch_11:HBM[11]
sp=Serpens.edge_list_ch_12:HBM[12]
sp=Serpens.edge_list_ch_13:HBM[13]
sp=Serpens.edge_list_ch_14:HBM[14]
sp=Serpens.edge_list_ch_15:HBM[15]
sp=Serpens.edge_list_ch_16:HBM[16]
sp=Serpens.edge_list_ch_17:HBM[17]
sp=Serpens.edge_list_ch_18:HBM[18]
sp=Serpens.edge_list_ch_19:HBM[19]
sp=Serpens.edge_list_ch_20:HBM[20]
sp=Serpens.edge_list_ch_21:HBM[21]
sp=Serpens.edge_list_ch_22:HBM[22]
sp=Serpens.edge_list_ch_23:HBM[23]
sp=Serpens.edge_list_ptr:HBM[24]
sp=Serpens.vec_X:HBM[25]
sp=Serpens.vec_Y:HBM[26]
sp=Serpens.vec_Y_out:HBM[27]
```

No Longer Needed

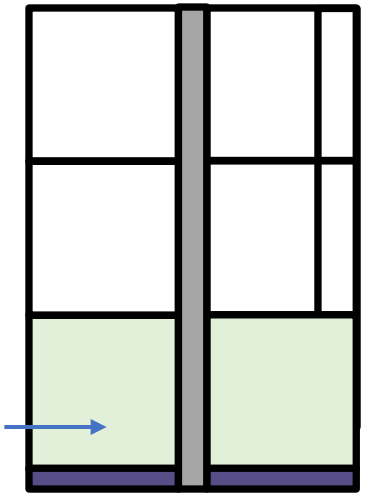
HBM-Specific Optimization

- Reduce the area overhead of HBM IO modules
- TAPA implements a customized IO module

Memory Interface	Clock/MHz	LUT	FF	BRAM
`#pragma HLS interface m_axi`	300	1189	3740	15
`async_mmap`	300	1466	162	0

- A small change for one instance, but we have 32 of them ☺
- With the HLS version, 32 IO instances cost 480 BRAM_36K, 71.4% of all BRAMs in SLR 0

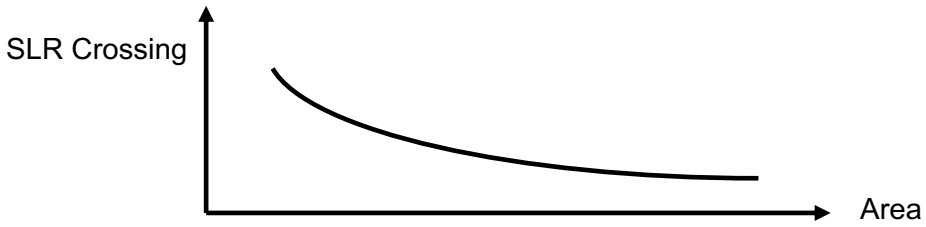
With Vitis HLS, 71.4% BRAMs consumed by HBM IO modules

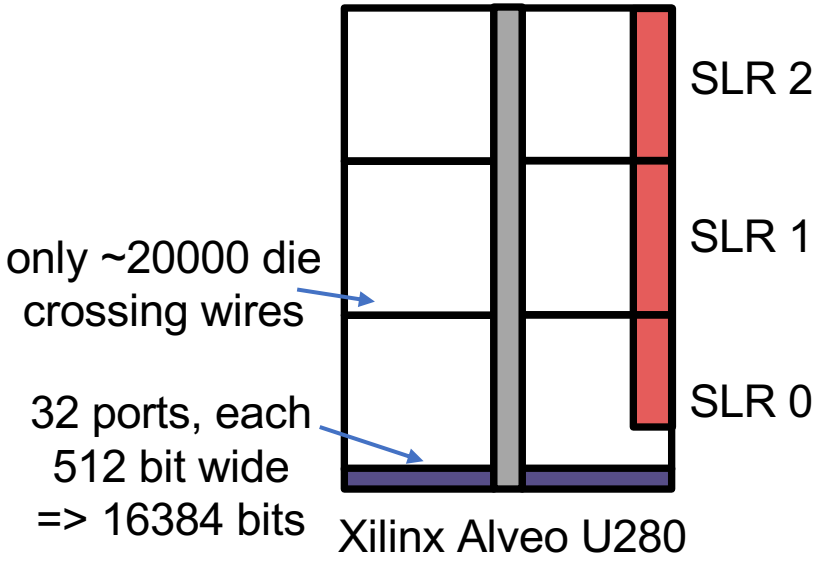


Xilinx Alveo U280

HBM-Specific Optimization

- HBM boards have both wire pressure and resource pressure at the bottom SLR
- Little variation in source usage / SLR crossing has a huge impact on the final outcome
- TAPA explore the whole pareto-optimal curve and generate multiple floorplan configs.
- Place and route each config in parallel.

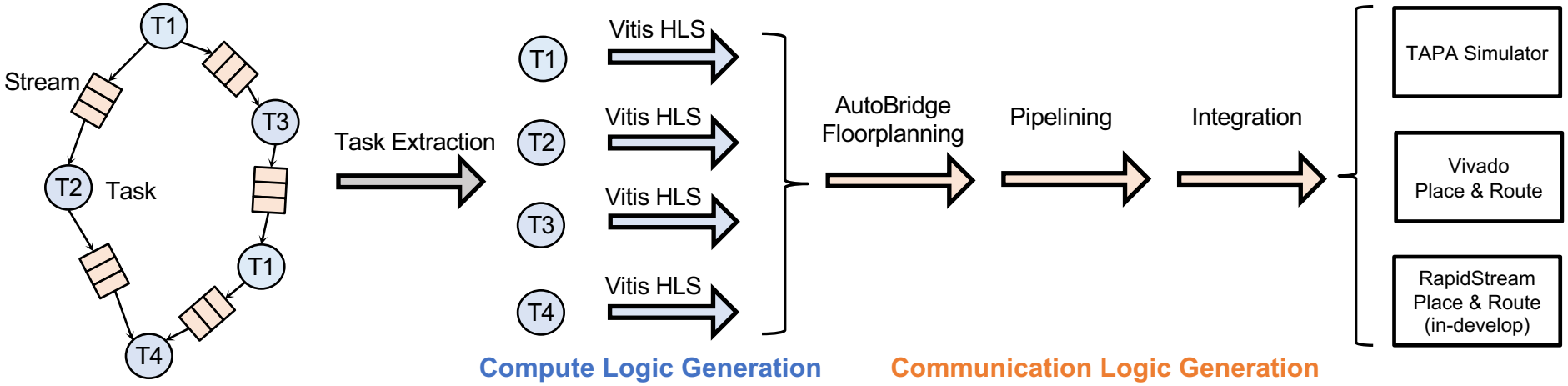




Part 2: Speed

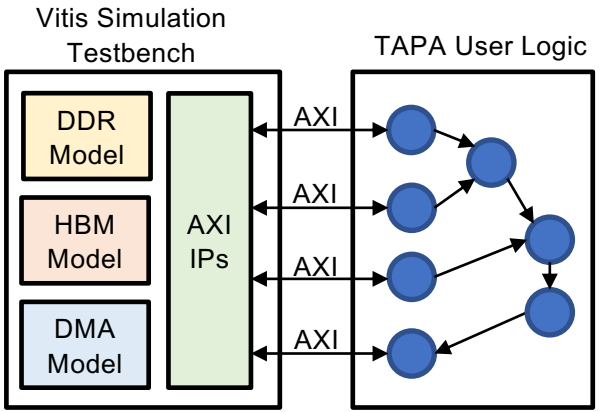
Speed Advantage

- TAPA invokes Vitis HLS to compile each parallel task separately
- TAPA generates customized RTL to stitch together individual tasks

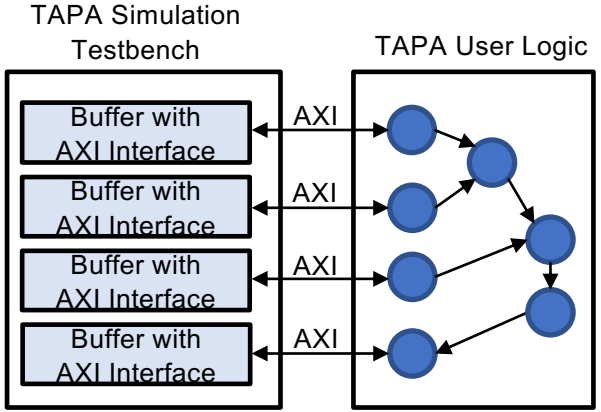


Speed Advantage

- Lightweight RTL simulation model
- Reduce the time overhead of setting up the simulation
- Much simplified when using tens of HBM channels



Setup time: >10 min even for vector add
Accurately mimic physical devices



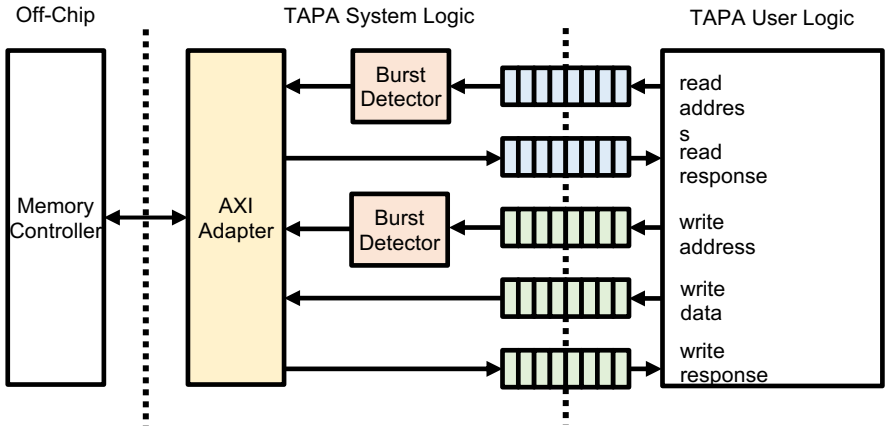
Setup time: always <1 second
Fully adhere to AXI protocol

More on Speed Later

Part 3: Richer Expressiveness

Flexible External Memory Access

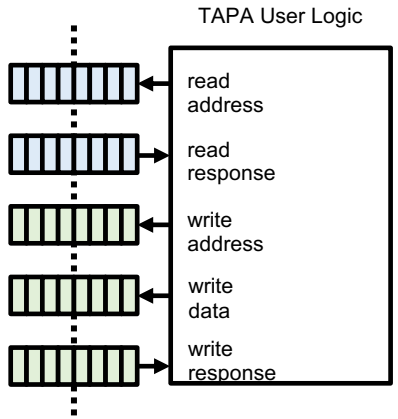
- Dedicated APIs to expose all AXI channels to the user
  - Abstract external memory as 5 independent streams
- Runtime burst detection
  - Automatically merging individual accesses to burst



Flexible External Memory Access

```cpp
void issue_read_addr(tapa::async_mmap<data_t>& mem, int n) {
  for (int i = 0; i < n; ) {
    if (!mem.read_addr.full()) {
      mem.read_addr.try_write(get_rand_addr());
      i++;
    }
  }
}

void receive_read_resp(tapa::async_mmap<data_t>& mem, int n) {
  for (int i = 0; i < n; ) {
    if (!mem.read_data.empty()) {
      data_t d = mem.read_data.read();
      i++;
      // ...
    }
  }
}
```

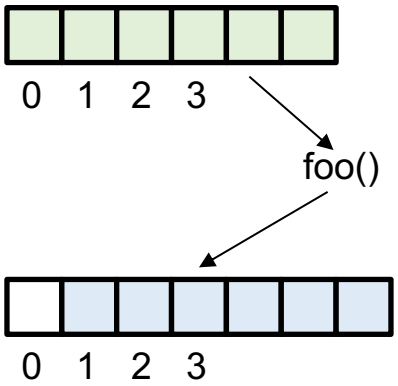
```cpp
void top(tapa::mmap<data_t> mem, int n) {
  tapa::task()
    .invoke(issue_read_addr, mem, n)
    .invoke(receive_read_resp, mem, n)
    // ...
    ;
}
```



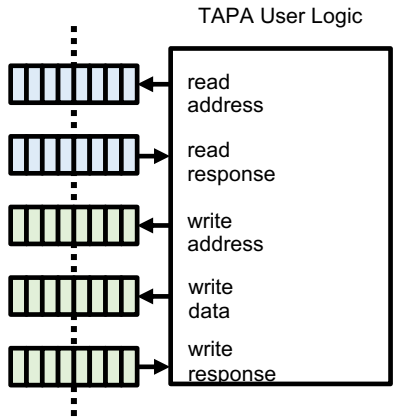
Flexible External Memory Access

A toy example:

- read 1000 elements from addr 0
- do some processing
- write back to addr 1?



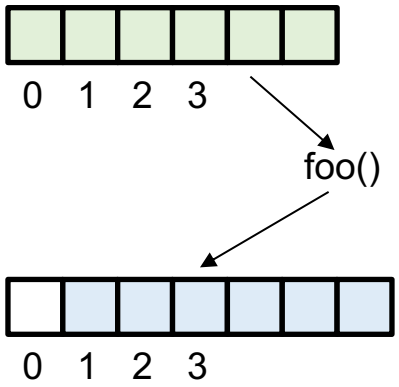

```
for(int i_rd_req=0, i_rd_resp=0, i_wr_req=0; i_wr_req<n; ) {
#pragma HLS pipeline II=1

  bool can_read = !hbm.read_data.empty();
  bool can_write = !hbm.write_addr.full() &&
                   !hbm.write_data.full();

  // issue read requests to read_addr channel
  if (i_rd_req < n && hbm.read_addr.try_write(i_rd_req))
    ++i_rd_req;

  // receive read response and write out
  if (can_read && can_write) {
    Elem elem = hbm.read_data.read();
    hbm.write_addr.write(i_wr_req+1);
    hbm.write_data.write(foo(elem));
    ++i_wr_req; ++i_rd_resp;
  }
}
```



Flexible External Memory Access

A toy example:

- read 1000 elements from addr 0
- do some processing
- write back to addr 1?



Can you do this easily with Vitis HLS?

```
for(int i = 0; i < 1000; ++i)
  mem[i+1] = foo(mem[i]);
```

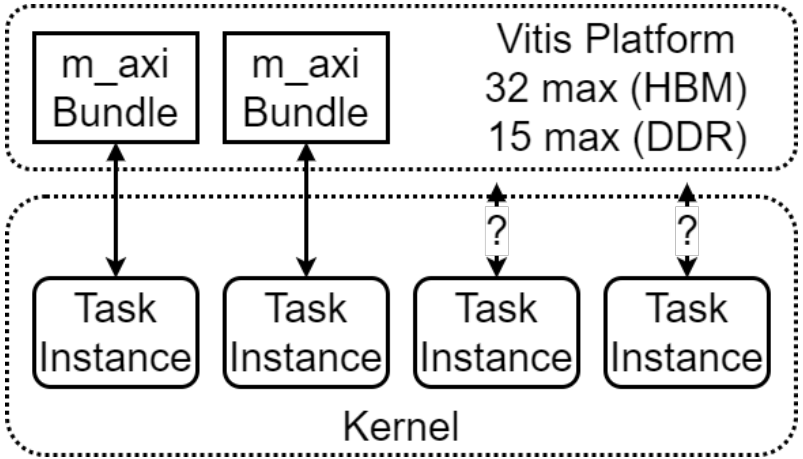
Incorrect semantics

```
buffer[1000];
for(int i = 0; i < 1000; ++i)
  buffer[i] = foo(mem[i]);

for(int i = 0; i < 1000; ++i)
  mem[i+1] = buffer[i];
```

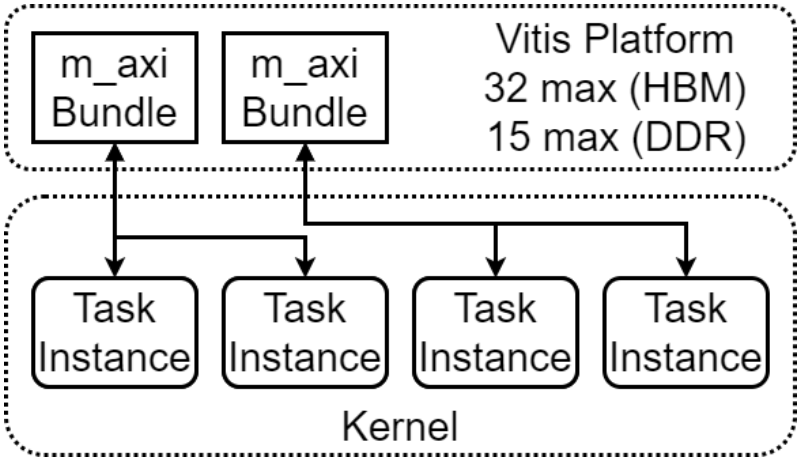
Area overhead + 2X cycle count

Flexible External Memory Access

- Vitis HLS does *not* support shared external memory access



- TAPA *does* support shared external memory access



Flexible Task Definition

- Hierarchical Tasks
  - TAPA tasks are recursively defined.
  - The children tasks instantiated by an upper-level task can itself be a parent of its children.
- Detached Task

```
using float_v16 = tapa::vec_t<float, 16>;
void FloatvMultConst(
    tapa::istream<float_v16> & fifo_in,
    tapa::ostream<float_v16> & fifo_out
) {
  while (1) {
    float_v16 tmp = fifo_in.read() * 3;
    fifo_out.write(tmp);
  }
}
```

Flexible Stream Access

- Peeking a stream

```
for (bool is_valid_0, is_valid_1;;) {
  const auto pkt_0 = pkt_in_q0.peek(valid_0);
  const auto pkt_1 = pkt_in_q1.peek(valid_1);
  ... // decide which input(s) can be consumed
  if (...) pkt_in_q0.read(nullptr);
  if (...) pkt_in_q1.read(nullptr);
}
```

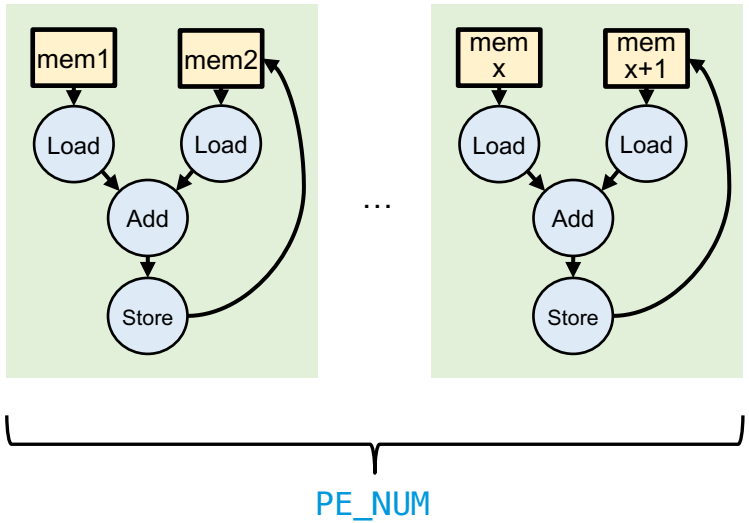
- End of Transaction

```
void Mmap2Stream(tapa::mmap<const float> mmap, uint64_t n,
                 tapa::ostream<tapa::vec_t<float, 2>>& stream) {
  [[tapa::pipeline(2)]] for (uint64_t i = 0; i < n; ++i) {
    tapa::vec_t<float, 2> tmp;
    tmp.set(0, mmap[i * 2]);
    tmp.set(1, mmap[i * 2 + 1]);
    stream.write(tmp);
  }
  stream.close();
}
```

Parameterized APIs

- Instantiate the `Add, Load, Store` tasks for `PE_NUM` times
- Each group connects to one external memory port
- Adjust the size and parallelism of the design by changing `PE_NUM`

```
void VecAdd(mmaps<float, PE_NUM> mem1,
            mmaps<float, PE_NUM> mem2,
            uint64_t n) {
  streams<float, PE_NUM, 8> a("a");
  streams<float, PE_NUM, 8> b("b");
  streams<float, PE_NUM, 8> c("c");

  task().invoke(Load, mem1, n, a)
        .invoke(Load, mem2, n, b)
        .invoke(Add, a, b, c)
        .invoke(Store, c, mem2, n);
}
```

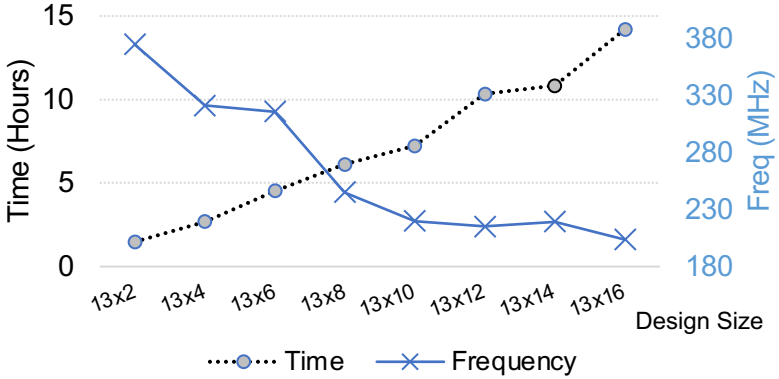


Part 4: Next Steps

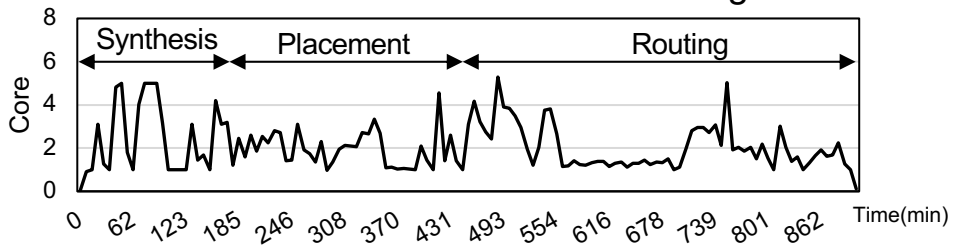
Challenges

- Physical design algorithms are hard to parallelize
- Frequency drops as designs become larger

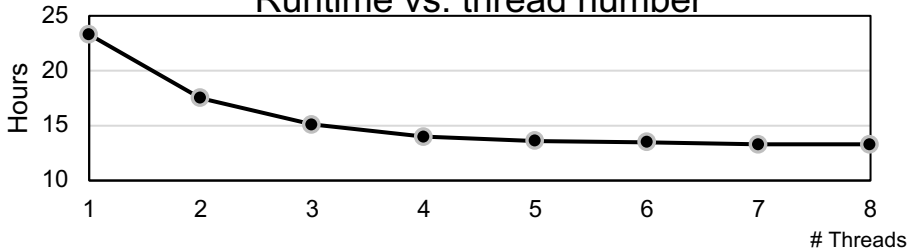
Example: a set of CNN accelerators



About 2 cores are active on average



Runtime vs. thread number

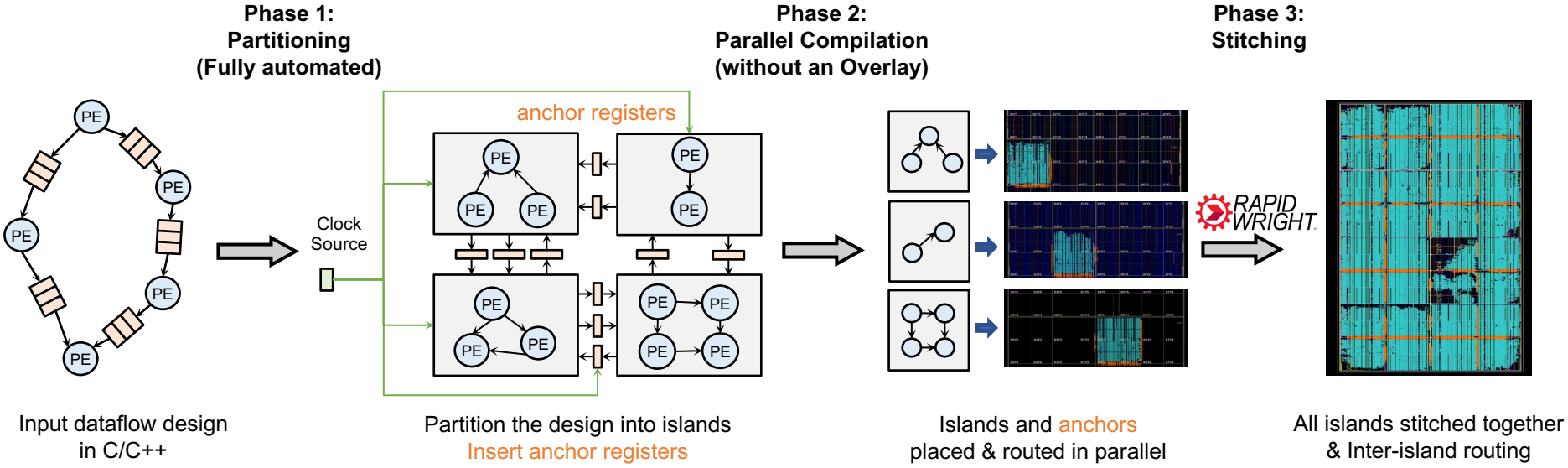


Tested on a 56-core server

RapidStream

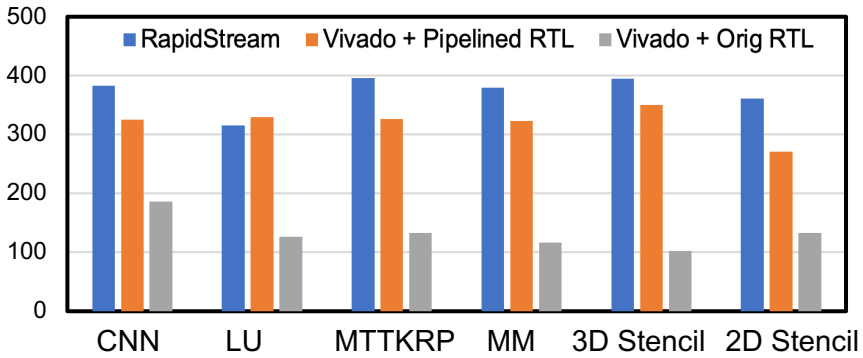
- Split compilation of HLS dataflow designs
- A fully automated flow that maps C++ designs to routed implementation
- Achieves high frequency and negligible increase in cycle counts
- Compared to Vivado, we get **5-7X** initial speedup, up to **1.3X** frequency improvement, up to **10X** speedup with ongoing efforts
- Fully open-sourced at https://github.com/Licheng-Guo/RapidStream

Overview

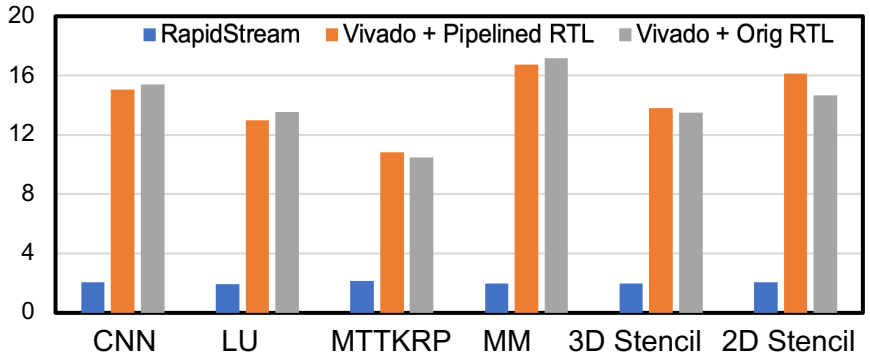

Phase 1:
Partitioning
(Fully automated)
Phase 2:
Parallel Compilation
(without an Overlay)
Phase 3:
Stitching
PE
Clock
Source
anchor registers
RAPID
WRIGHT
Input dataflow design
in C/C++
Partition the design into islands
Insert anchor registers
Islands and anchors
placed & routed in parallel
All islands stitched together
& Inter-island routing

Overview

- Tested on 6 large scale dataflow designs targeting Xilinx U250 FPGA with 4 SLRs (dies)
- Distribute to 4 Xeon servers, each with 56 cores
- Divide the FPGA into 32 islands (8 rows, 4 columns)
- 5-7X speedup (from C++ to fully routed checkpoint)
- Up to 1.3X frequency improvement



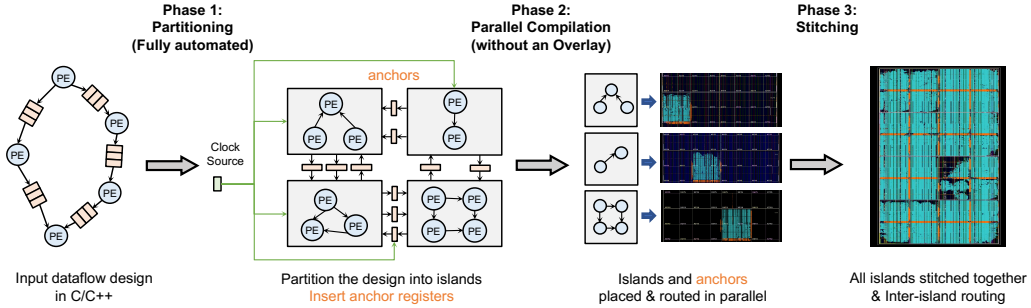
Frequency (MHz)

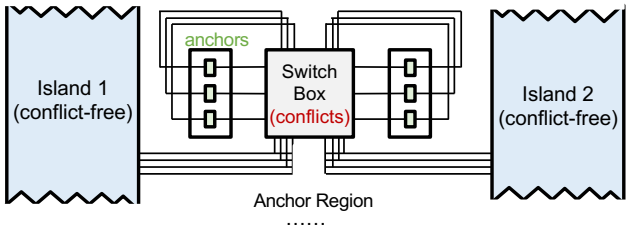


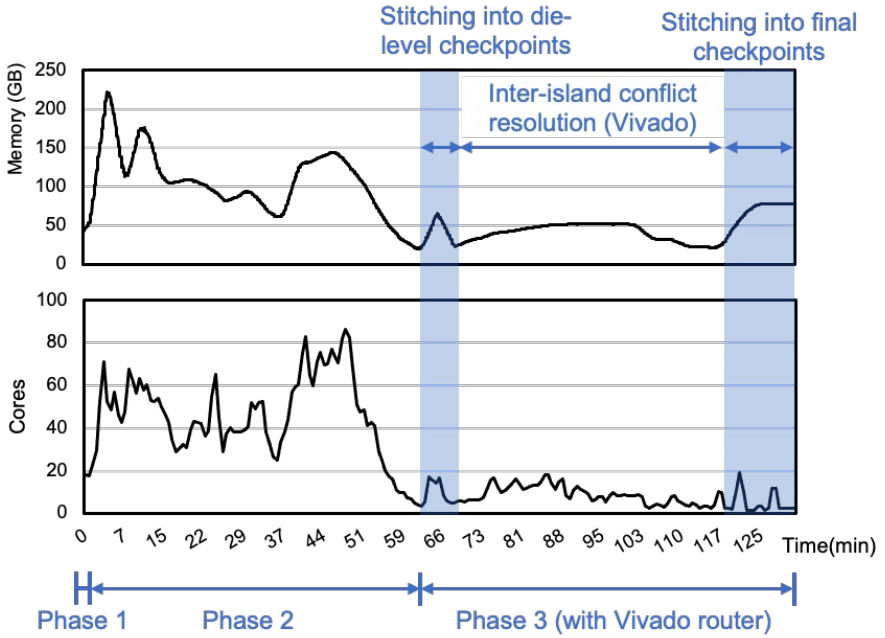
Runtime (hour)

Evaluation

- 26 cores active on average
- Traditional methods only have 2 cores active on average
- Long tail in Phase 3: standard routers are not designed for partial re-routing







Integration

- We need a CPU-FPGA communication infrastructure (DMA, DDR, etc.)
- Direction 1: develop a customized infra using standard IPs
  - Pros: full control of the infra
  - Cons: hard to maintain
- Direction 2: hack a Vitis-generated checkpoint and reuse its infra.
  - Pros: compatible with the Vitis eco-system
  - Cons: the infra is a black box, not sure if we can integrate the whole flow
- Welcome any suggestion!

Thank You!

https://github.com/UCLA-VAST/tapa

https://tapa.rtfd.io/

{lcguo, chiyuze, lau}@cs.ucla.edu

Acknowledgement

This work is partially supported by the CRISP Program, the CDSC Industrial Partnership Program, the Xilinx Adaptive Compute Clusters Program, the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA), NSF NeuroNex Award DBI-1707408, a Google Faculty Award, the NSF RTML program (CCF1937599), NIH Brain Initiative (U01MH117079). We thank Gurobi and GNU Parallel for their academic licenses.