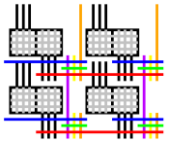
HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation

Yuanlong Xiao, Aditya Hota, **Dongjoon(DJ) Park**, and André DeHon
<ylxiao, ahota, dopark>@seas.upenn.edu, andre@ieee.org

Implementation of Computation Group
University of Pennsylvania
August 29th, 2022



Penn

Story

Have you ever had this problem:

You just spent 2 hours compiling your FPGA design

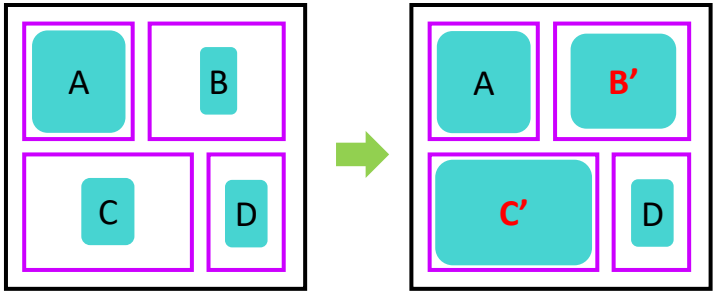
…and you discover a small change you need to make in one function
(e.g. buffer size, sign error)

Now, you must wait *another* 2 hours
before you use/test the modified design?

Story

- Problem
  - ❑ More developers use High-Level-Synthesis (C/C++) and expect fast incremental compile... but FPGA compilation is slow!
  - ❑ **Partial Reconfiguration** (PR) can be useful in **fast incremental development**, but it requires hardware expertise (PR tool flow, floorplanning PR regions)

Smaller problem size than compiling the entire design!



<Incremental development using PR>

Story

- Problem
  - More developers use High-Level-Synthesis (C/C++) and expect fast incremental compile... but FPGA compilation is slow!
  - **Partial Reconfiguration** (PR) can be useful in **fast incremental development**, but it requires hardware expertise (PR tool flow, floorplanning PR regions)
- Idea - HiPR
  - Creates an **application-customized static design to support fast, PR-based, incremental development**

- What HiPR will deliver
  - Enables **PR-functions to be defined at C-level** and **floorplans** PR regions for HLS users
  - Automates fast, PR-based, incremental compile (compatible with Xilinx Vitis)
  - Decreases incremental compile from **hours** to **7-20 minutes** without performance loss

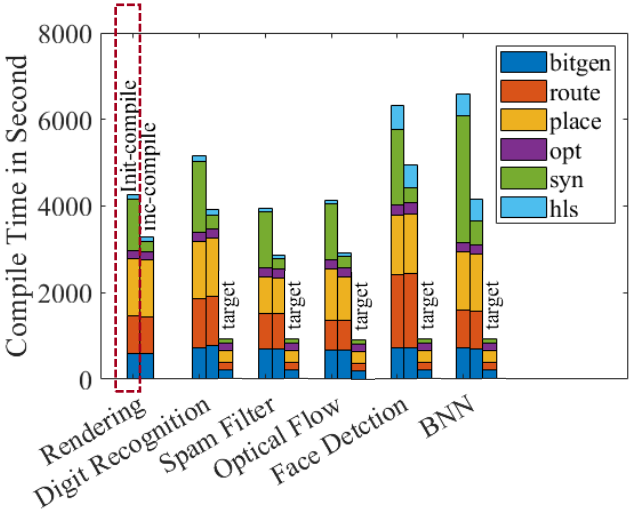
Outline

- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

Outline

- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

Motivation: Initial-compile & Incremental-compile by Vitis

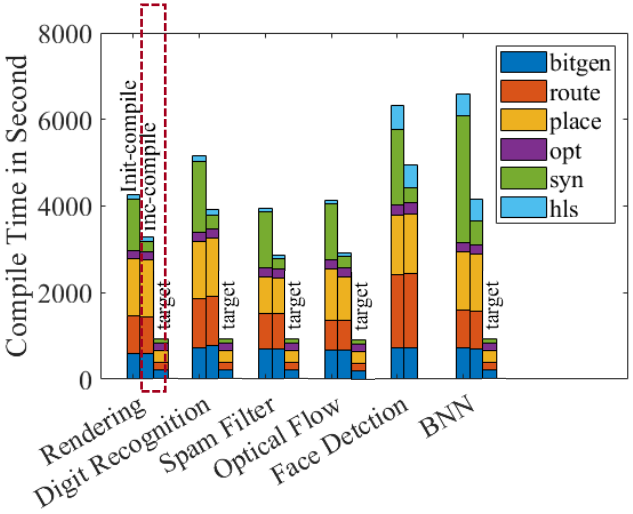


<Benchmark[1] Compile Time Breakdowns with Vitis (on Xilinx Alveo U50)>

- Slow FPGA Compilation with Vitis
  - It takes **65-109 min** for initial-compilation

[1] Yuan Zhou et al. Rosetta: *A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs*. ISFPGA'18

Motivation: Initial-compile & Incremental-compile by Vitis

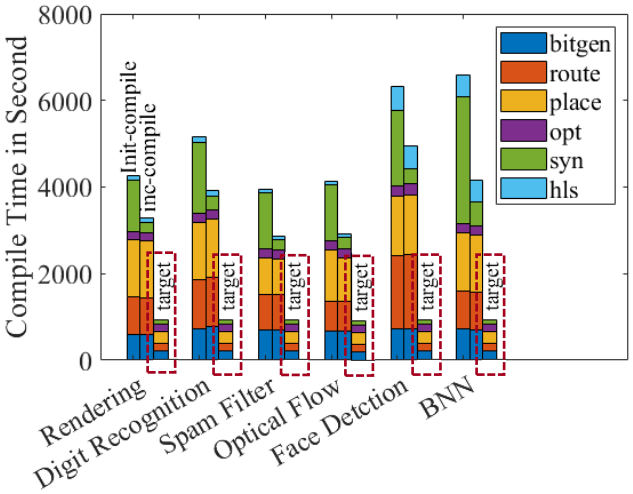


<Benchmark[1] Compile Time Breakdowns with Vitis (on Xilinx Alveo U50)>

- Slow FPGA Compilation with Vitis
  - It takes **65-109 min** for initial-compilation
  - With only a small change in source file, incremental compilation time is still long (**48-82 min**)
    - Place/Route/Bit-gen is still long!

[1] Yuan Zhou et al. Rosetta: *A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs*. ISFPGA'18

Motivation: Initial-compile & Incremental-compile by Vitis



<Benchmark[1] Compile Time Breakdowns with Vitis (on Xilinx Alveo U50)>

- Slow FPGA Compilation with Vitis
  - It takes **65-109 min** for initial-compilation
  - With only a small change in source file, incremental compilation time is still long (**48-82 min**)
    - Place/Route/Bit-gen is still long!
- Can we
  - Decrease the incremental compile to less than **20 minutes** with PR?
  - Enable HLS developers to use **PR techniques at C-level**?

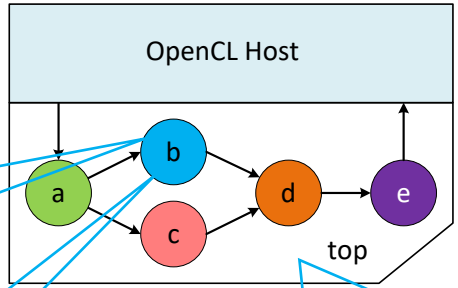
[1] Yuan Zhou et al. Rosetta: *A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs*. ISFPGA'18

Outline

- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

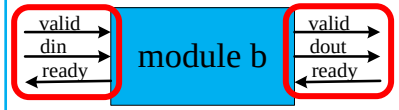
HiPR (High-level Partial Reconfiguration): Compute Model

- Prepare an application based on latency insensitive computing model[17] (e.g.: operators: a, b, c, d, e)



b.cpp file

```
1 void b(hls::stream< ap_uint<32> > & Input_1,
2         hls::stream< ap_uint<32> > & Output_1) {
3 #pragma HLS INTERFACE axis register port=Input_1
4 #pragma HLS INTERFACE axis register port=Output_1
5   ap_fixed<48, 27> buf[2];
6   ap_fixed <32, 13> tmp_in, tmp_out;
7   for(int r=0; r<MAX_NUM; r++) {
8     tmp_in(31, 0)=Input_1.read();
9     ap_fixed<96, 56> t1 = (ap_fixed<96,56>) tmp_in;
10    tmp_in(31, 0)=Input_1.read();
11    ap_fixed<96, 56> t2 = (ap_fixed<96,56>) tmp_in;
12    ... /* computation */
13    tmp_out = (ap_fixed<32, 13>) (buf[0] + buf[1]);
14    Output_1.write(tmp_out(31, 0));
15 }}
```

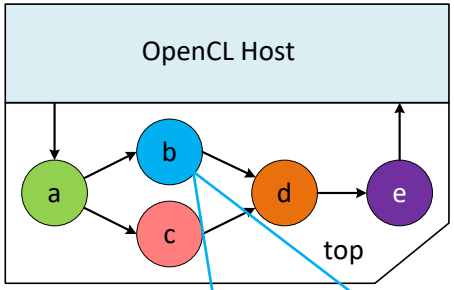



```
1  void top(hls::stream< ap_uint<32> > & Input_1,
2           hls::stream< ap_uint<32> > & Output_1) {
3    hls::stream< ap_uint<32> > a2b
4  #pragma HLS STREAM variable=a2b
5    ... /* stream link definitions */
6    hls::stream< ap_uint<32> > d2e;
7  #pragma HLS STREAM variable=d2e
8    ... /* dataflow graph description */
9    a(Input_1, a2b, a2c);
10   b(a2b, b2d);
11   c(a2c, c2d);
12   d(b2d, c2d, d2e);
13   e(d2e, Output_1);
14 }
15
```

[17] G. Kahn, "The semantics of a simple language for parallel programming," in Proceedings of the IFIP CONGRESS 74. North-Holland Publishing Company, 1974, pp. 471–475.

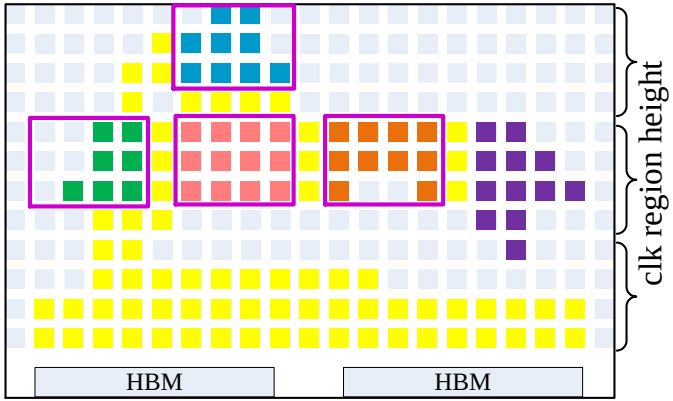
HiPR (High-level Partial Reconfiguration): Compute Model

- Prepare an application based on latency insensitive computing model[17] (e.g.: operators: a, b, c, d, e)
- Define PR-functions at C-level with *pragmas*



b.hpp file

```
1 void b(hls::stream< ap_uint<32> > & Input_1,
2         hls::stream< ap_uint<32> > & Output_1) {
3 #pragma HLS PR clb=4 bram=2.4 dsp=8
```



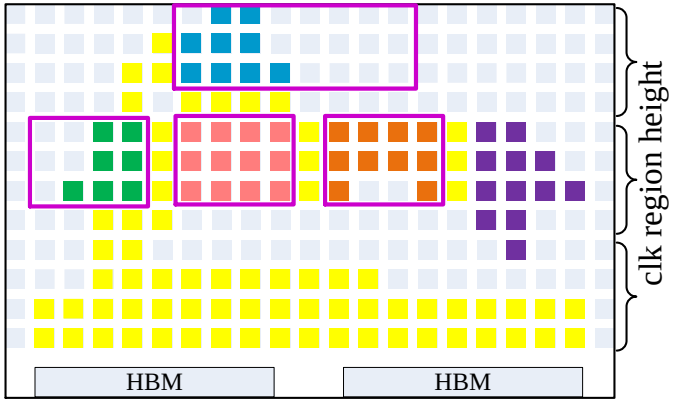
- HLS PR: operator b is reconfigurable
- clb=4, bram=2.4, dsp=8: the PR region should have 4 times more CLB than what operator b needs now

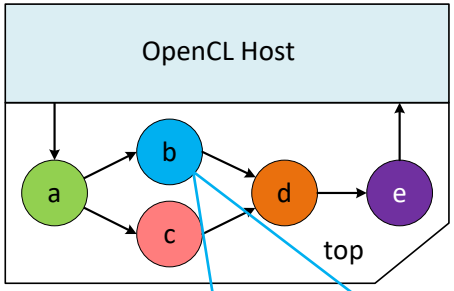
[17] G. Kahn, "The semantics of a simple language for parallel programming," in Proceedings of the IFIP CONGRESS 74. North-Holland Publishing Company, 1974, pp. 471–475.

HiPR (High-level Partial Reconfiguration): Compute Model

- Prepare an application based on latency insensitive computing model[17] (e.g.: operators: a, b, c, d, e)
- Define PR-functions at C-level with *pragmas*
- What if the revised function is larger?
  (e.g. Increase buffer size, increase parallelism)

➔ Reserve more resources for future tuning





b.hpp file

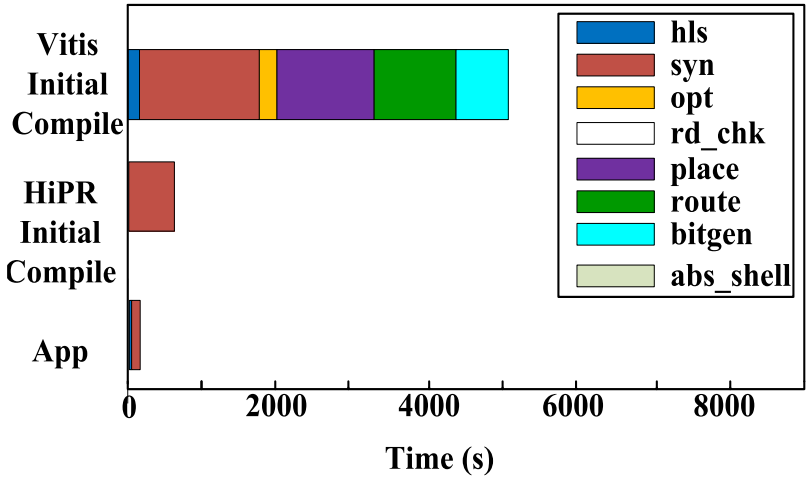
```
1 void b(hls::stream< ap_uint<32> > & Input_1,
2         hls::stream< ap_uint<32> > & Output_1) {
3 #pragma HLS PR clb=4 bram=2.4 dsp=8
```

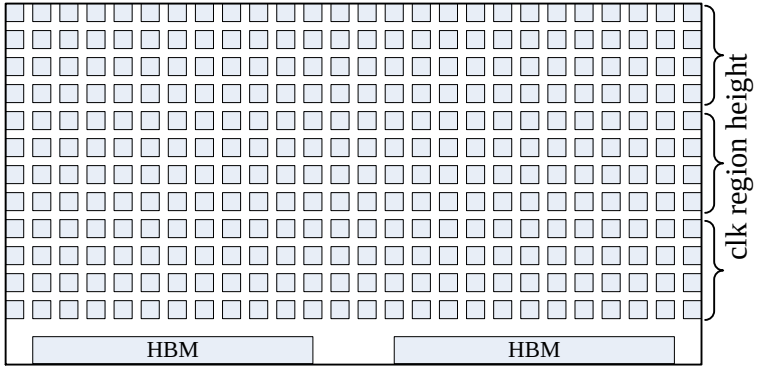
- HLS PR: operator b is reconfigurable
- clb=4, bram=2.4, dsp=8: the PR region should have 4 times more CLB than what operator b needs now

[17] G. Kahn, "The semantics of a simple language for parallel programming," in Proceedings of the IFIP CONGRESS 74. North-Holland Publishing Company, 1974, pp. 471–475.

HiPR Toolflow: Initial-Compile

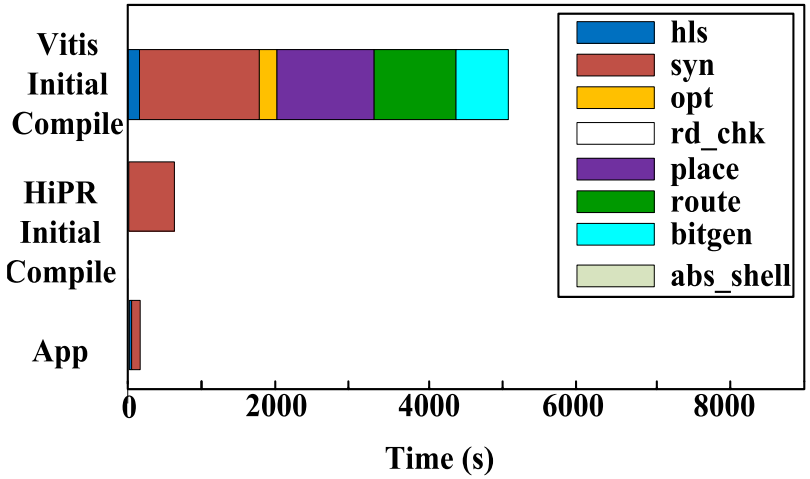
- Synthesize each operator in parallel

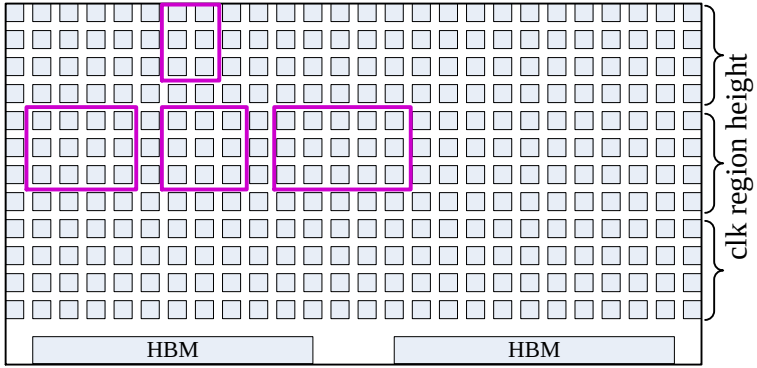




HiPR Toolflow: Initial-Compile

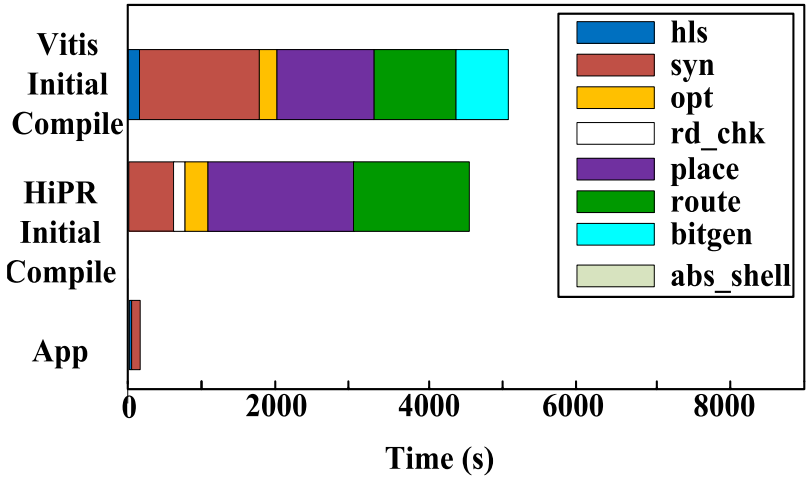
- Synthesize each operator in parallel
- Floorplan the PR regions based on PR pragma and connectivity

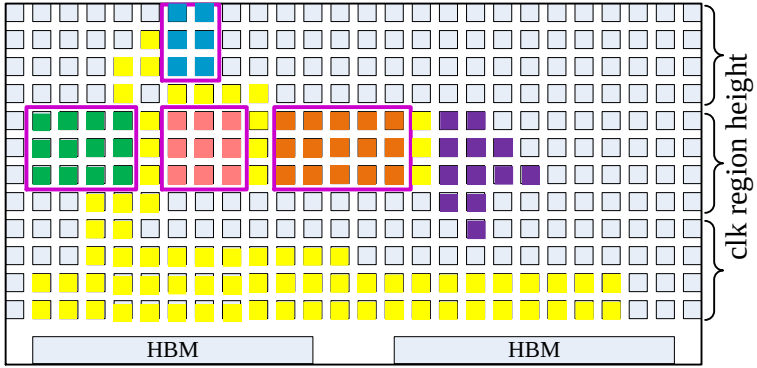




HiPR Toolflow: Initial-Compile

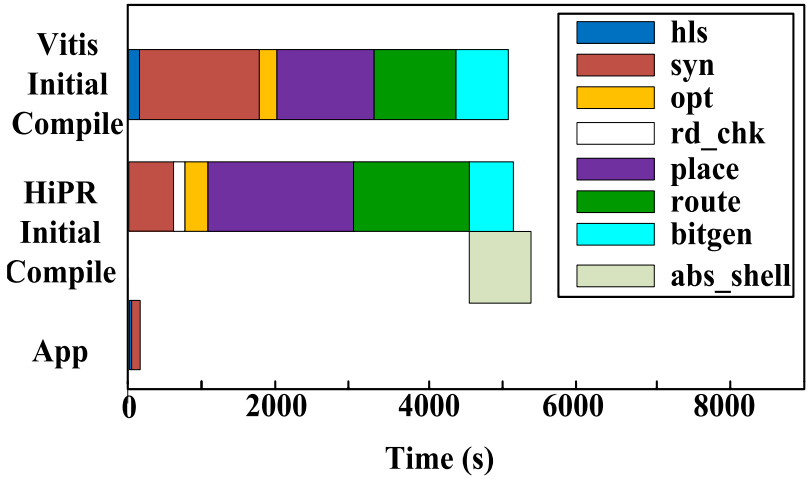
- Synthesize each operator in parallel
- Floorplan the PR regions based on PR pragma and connectivity
- Place/Route to generate a fully routed design with placeholders

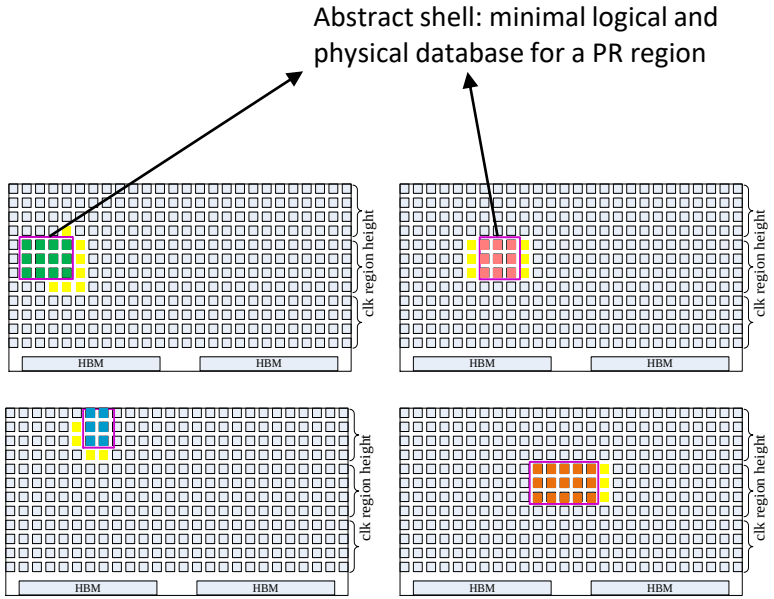




HiPR Toolflow: Initial-Compile

- Synthesize each operator in parallel
- Floorplan the PR regions based on PR pragma and connectivity
- Place/Route to generate a fully routed design with placeholders
- Generate a separate *abstract shell*[22] for each PR region

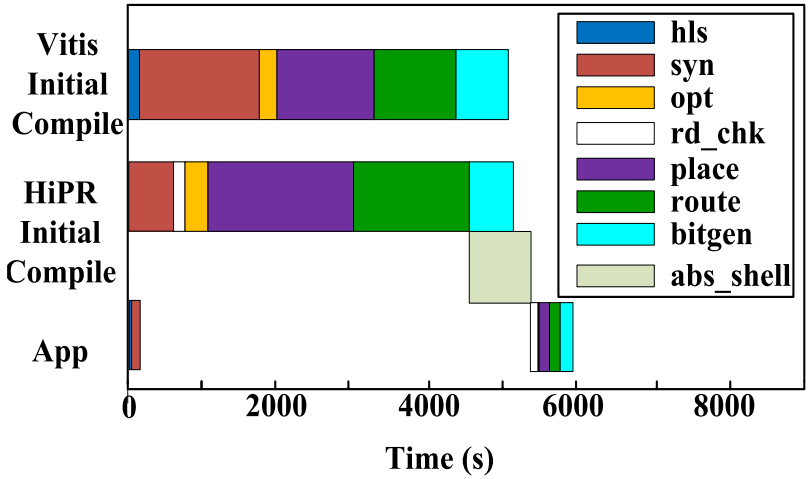


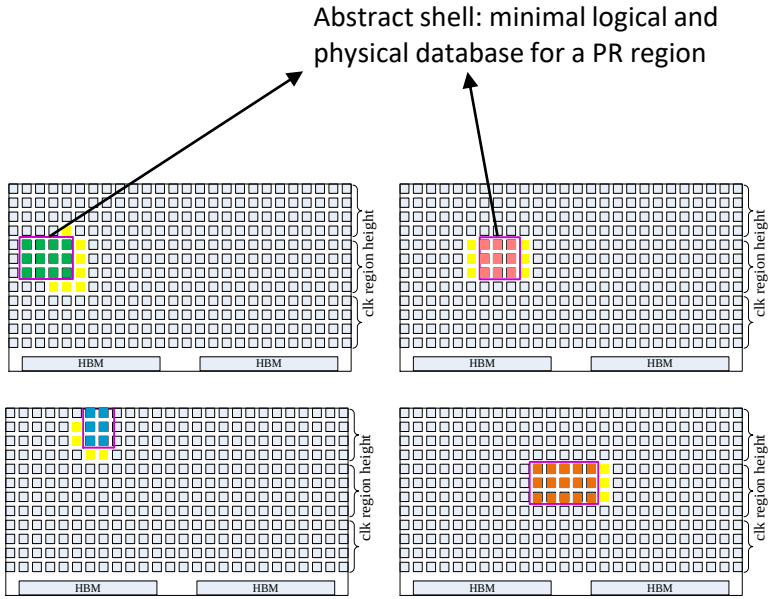


[22] UG909: Vivado Design Suite User Guide: Dynamic Function eXchange, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021.

HiPR Toolflow: Initial-Compile

- Synthesize each operator in parallel
- Floorplan the PR regions based on PR pragma and connectivity
- Place/Route to generate a fully routed design with placeholders
- Generate a separate *abstract shell*[22] for each PR region
- Place/Route/Bit-gen each operator separately in parallel

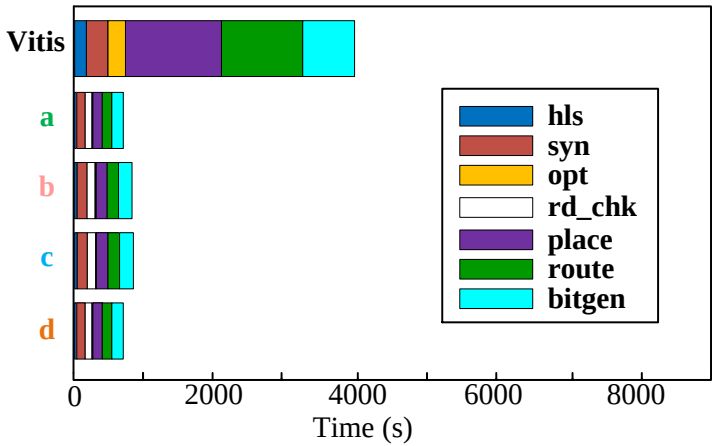


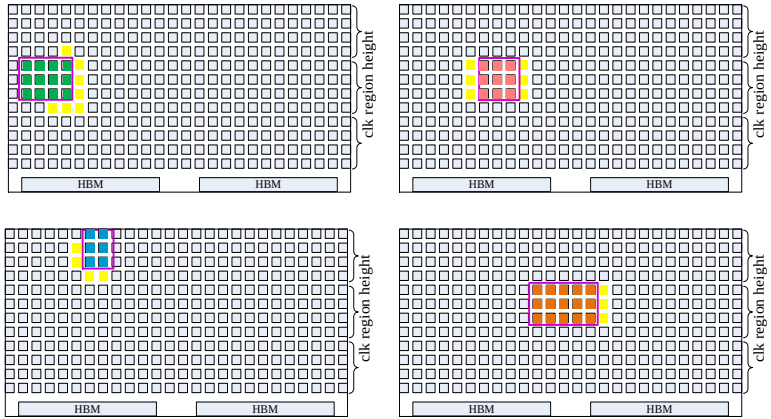


[22] UG909: Vivado Design Suite User Guide: Dynamic Function eXchange, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021.

HiPR Toolflow: Incremental-Compile

- Re-compile only the modified function
- Compiles separately in parallel
  - Smaller problem size ➔ faster compilation
  - Compilation time is determined by the longest among the parallel compile runs



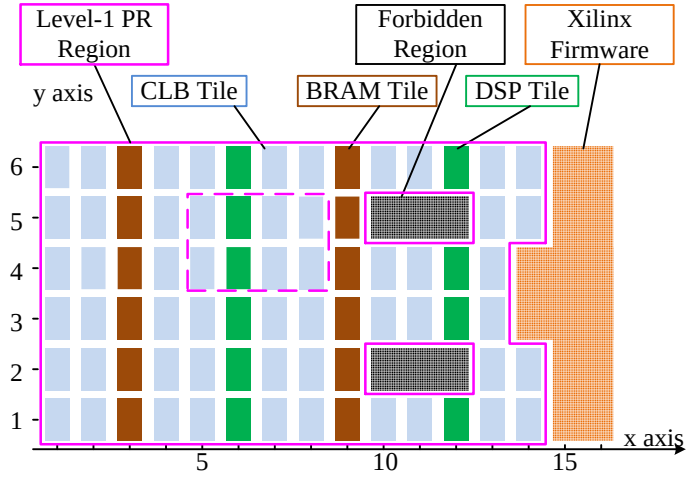


Outline

- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

Floorplan: Architecture Model

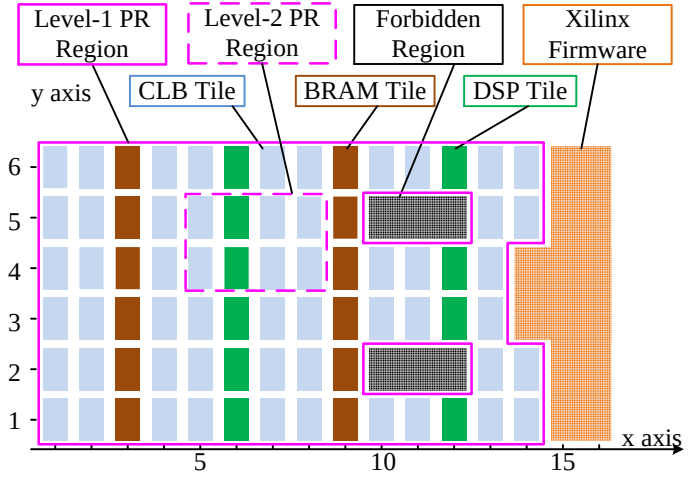
- Architecture model for a device
  - Resource Vector <CLB, CLB, BRAM,…,CLB>
  - Forbidden Region <X, Y, W, H> (<10,5,3,1>, …)
- Hierarchical PR[22]
  - Xilinx Datacenter Platform provides Level-1 PR region



[22] UG909: Vivado Design Suite User Guide: Dynamic Function eXchange, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021.

Floorplan: Architecture Model

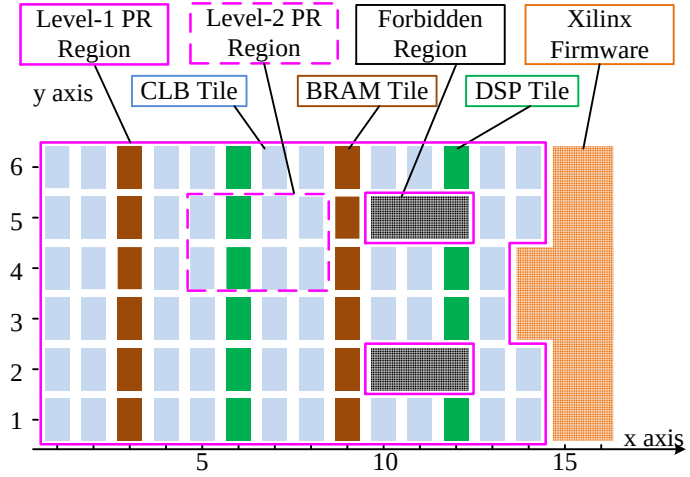
- Architecture model for a device
  - Resource Vector <CLB, CLB, BRAM,…,CLB>
  - Forbidden Region <X, Y, W, H> (<10,5,3,1>, …)
- Hierarchical PR[22]
  - Xilinx Datacenter Platform provides Level-1 PR region
  - Level-2 PR regions for PR-functions are defined using Hierarchical PR



[22] UG909: Vivado Design Suite User Guide: Dynamic Function eXchange, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021.

Floorplan: Architecture Model

- Architecture model for a device
  - Resource Vector <CLB, CLB, BRAM,…,CLB>
  - Forbidden Region <X, Y, W, H> (<10,5,3,1>, …)
- Hierarchical PR[22]
  - Xilinx Datacenter Platform provides Level-1 PR region
  - Level-2 PR regions for PR-functions are defined using Hierarchical PR
- Output Constraints
  - Level-2 PR regions, <X, Y, W, H>
  - XDC constraints file



[22] UG909: Vivado Design Suite User Guide: Dynamic Function eXchange, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, June 2021.

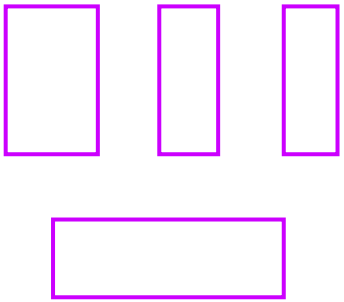
Floorplan: Simulated Annealing

- Cost Function
  - $min:\ \alpha * \frac{\text{TotalLinkLength}}{\text{MaxTotalLinkLength}} + \beta * \frac{\text{TotalWastedResource}}{\text{MaxTotalWastedResource}} + \text{Overlapping} \quad (\alpha + \beta \leq 0.5)$

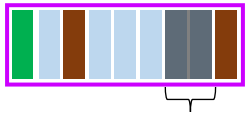
Minimize the distance between PR regions

Minimize the extra area reserved in PR regions

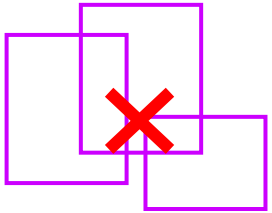
Overlapping PR regions is NOT allowed



e.g.) need 4 CLB, 2 BRAM, 1 DSP



extra area!



Floorplan: Simulated Annealing

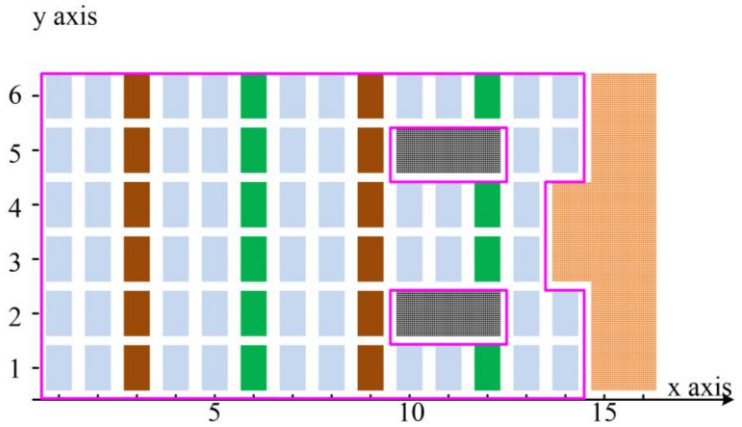
- Cost Function
  - $min:\ \alpha * \frac{\text{TotalLinkLength}}{\text{MaxTotalLinkLength}} + \beta * \frac{\text{TotalWastedResource}}{\text{MaxTotalWastedResource}} + \text{Overlapping} \quad (\alpha + \beta \leq 0.5)$

Minimize the distance between PR regions

Minimize the extra area reserved in PR regions

Overlapping PR regions is NOT allowed

- Simulated Annealing
  - Randomly selects an operator
  - Randomly generates <X, Y>
  - Greedily generates PR region for the operator



<Example scenario of simulated annealing algorithm in floorplanning>

Outline

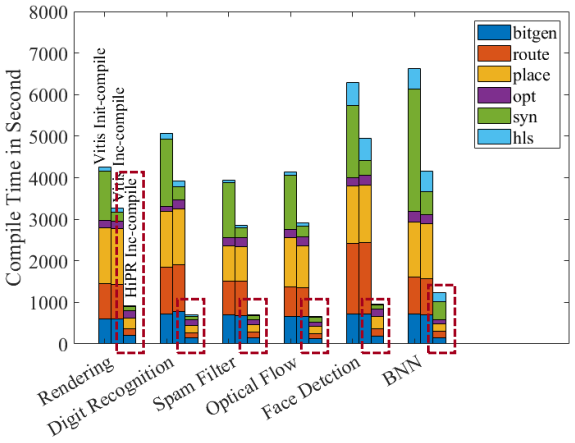
- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

Evaluation: Platform

- Compile servers: Google Cloud Platform (GCP)
  - 32 compute nodes, each with 8-thread, 2.8GHz Intel Xeon Cascade Lake Processors
  - Parallel Task Manager Slurm
- HiPR uses Vitis 2021.1
  - **Alveo U50 Data Center Card** with Virtex UltraScale+ XCU50
  - 751K LUTs, 2,300 BRAM18, 5,936 DSPs
- **Rosetta HLS Benchmark** [1]
  - 6 HLS Benchmark designs
    - 3-D Rendering, Digit-Recognition, Spam-filter, Optical-flow, BNN, Face-detection
  - We decompose each benchmark into a cluster of operators with latency insensitive streams

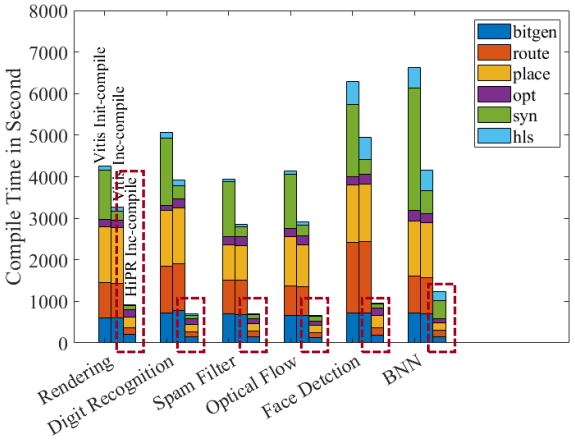
[1] Yuan Zhou et al. Rosetta: *A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs*. ISFPGA'18

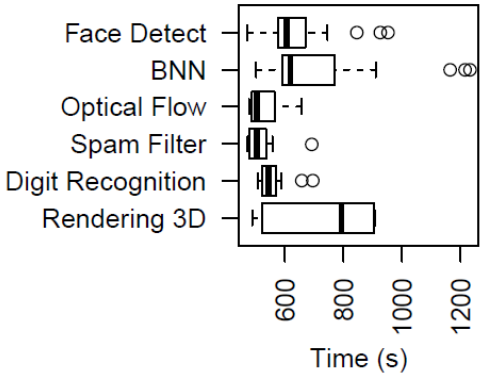
Evaluation: Incremental Compile



- Assume all operators have to be recompiled
- HiPR takes 7-20 mins for incremental compile while Vitis takes 48-82 mins (3-10x speedup)

Evaluation: Incremental Compile





- Assume all operators have to be recompiled
- HiPR takes 7-20 mins for incremental compile while Vitis takes 48-82 mins (3-10x speedup)
- Median compile times are around 11 mins

Evaluation: Initial Compile

- For initial compile, HiPR takes more time (15-67%) than Vitis flow
- Usually done once and amortized over time
  - As long as the interconnections between operators don't change

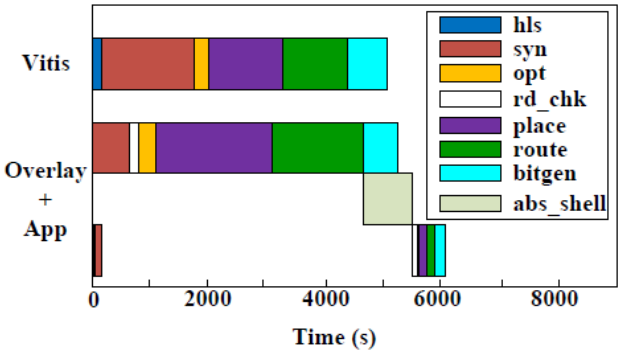


Fig. 8. Initial-compile Times Breakdown (Digit Recognition)

TAB III: Initial Compile Times Comparisons (in seconds)

Benchmark	Vitis Flow	HiPR Flow	Overhead
3d-rendering	4264	7152	67%
Digit recognition	5173	6125	18%
Spam Filter	3942	4541	15%
Optical Flow	4139	6880	66%
Face Detect	6288	8851	40%
Binary NN	6584	9632	46%

Evaluation: Performance

TAB IV: Performance Comparison: Vitis vs. HiPR

Benchmark	Vitis Flow		HiPR Flow	
	Freq (MHz)	Runtime (ms)	Freq (MHz)	Runtime (ms)
3d-rendering	200	2.2	200	1.6
Digit recognition	250	9.2	250	6.3
Spam Filter	300	18.6	300	20.0
Optical Flow	200	13.6	200	7.5
Face Detect	200	21.0	200	22.0
Binary NN	150	5250	150	4700

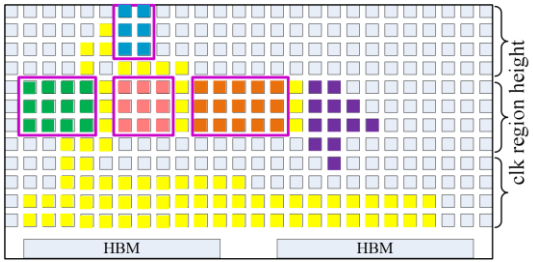
- Re-write the benchmark in form of latency-insensitive style
- Smaller and localized blocks with pipelined interconnect make it easier to meet timing
- HiPR matches the clock frequency and the application runtime of Vitis flow

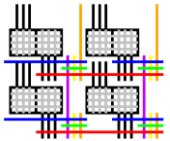
Outline

- Motivation
- Approach
- Floorplan
- Evaluation
- Conclusion

Conclusion

- HiPR: An open-source framework for HLS developers (https://github.com/icgrp/hipr)
- Bridge the gap between HLS and PR technique by adding a C-level PR pragma
- Decrease the incremental compile times from 48-82 minutes to 7-20 minutes (3-10x) without performance loss





Q & A

Thank you!