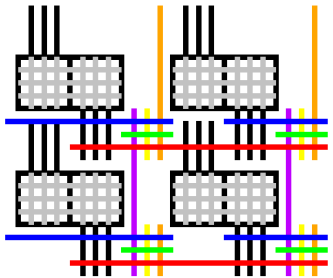
HiPR: Fast, Incremental Custom Partial Reconfiguration for HLS Developers

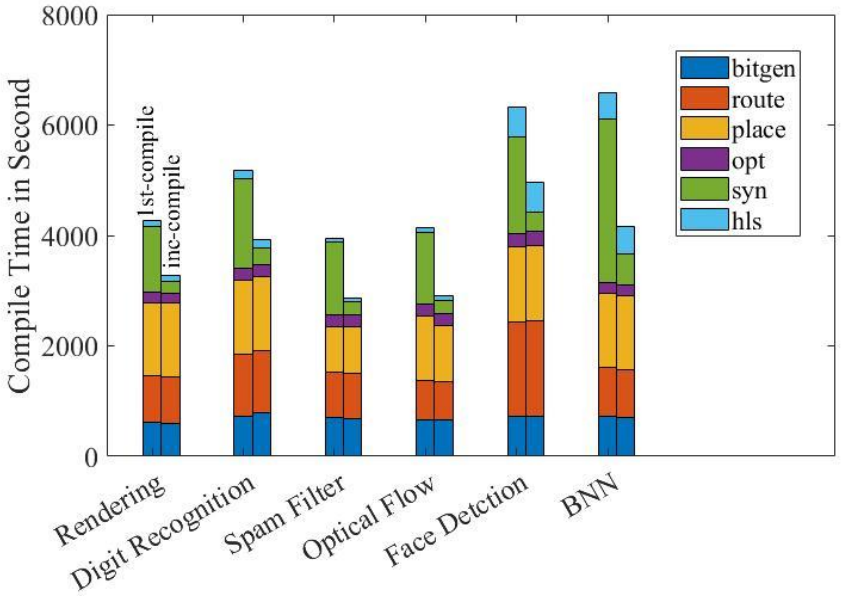
Yuanlong Xiao, and André DeHon
ylxiao@seas.upenn.edu, andre@ieee.org

Implementation of Computation Group, University of Pennsylvania
March 1st, 2022



Motivation

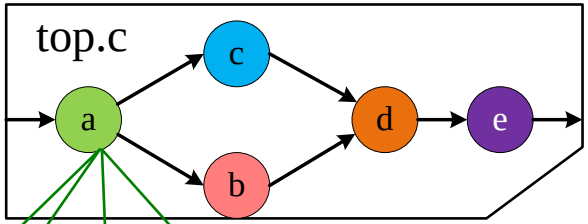
- Slow FPGA compilation
  - It takes more time for initial-compilation
  - Inc-compilation time is **still long** (place-route-bitgen)
- Can we
  - use **Initial-compile** to **set up the overlay** (define DFX regions for target function in HLS code)?
  - Only **incrementally-compile** the **target function** later for tuning and design space exploration?
  - **Incremental-compile** should be **much faster**



Vitis Compile Time Breakdown:
Initial Compile vs Incremental Compile

HiPR (**Hi**gh-Level **P**artial **R**econfiguration)

- Still start from latency insensitive computing model
- Define **DFX C function** instead of Verilog module

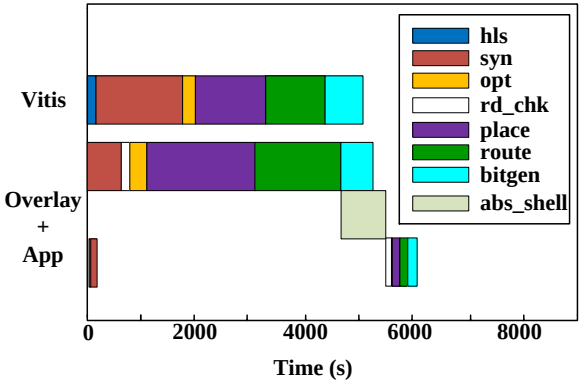


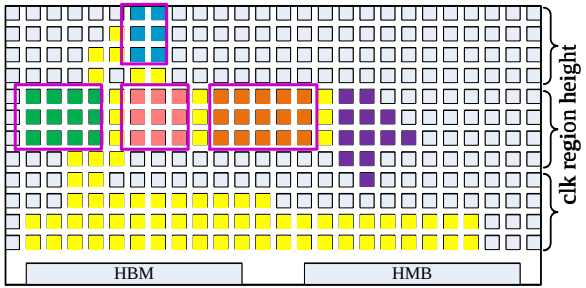

```
/****************header file**********************/
void a(hls::stream< ap_uint<32>> & Input_1,
       hls::stream< ap_uint<32>> & Output_1);
#pragma HLS PR clb=4 bram=2.4 dsp=1.2
```

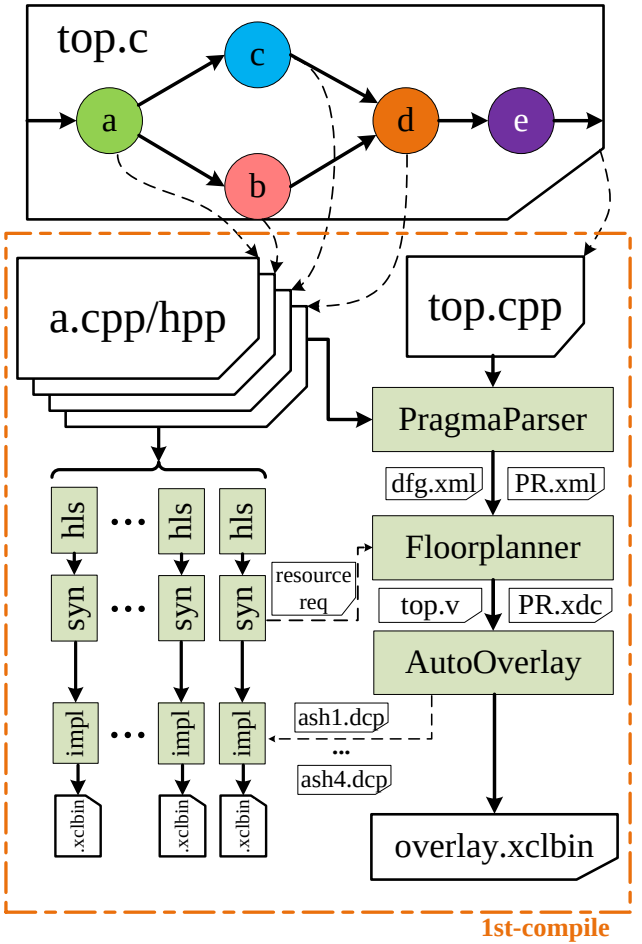
```
void a(hls::stream< ap_uint<32>> & Input_1,
       hls::stream< ap_uint<32>> & Output_1,
       hls::stream< ap_uint<32>> & Output_2){
#pragma HLS INTERFACE axis register port=Input_1
#pragma HLS INTERFACE axis register port=Output_1
#pragma HLS INTERFACE axis register port=Output_2
  ap_fixed<48,27> buf[2];
  ap_uint<32, 13> din;
  ap_fixed<32,13> dout;
  OUTER: for(int r=0; r<MAX_HEIGHT; r++){
    INNER: for(int c=0; c<MAX_WIDTH; c++){
      ... /* data type conversion */
      din(i*32+31, i*32) = Input_1.read();
      ap_fixed<96,56> t1 = (ap_fixed<96,56>) din;
      din(i*32+31, i*32) = Input_1.read();
      ap_fixed<96,56> t2 = (ap_fixed<96,56>) din;
      ... /* computation */
      out_tmp = (ap_fixed<32,13>) buf[0];
      Output_1.write(out_tmp(31, 0));
      outputs = (ap_fixed<32,13>) buf[1];
      Output_2.write(out_tmp(31, 0));
}}}
```

HiPR tool flow: initial-compile

- Compile all-sub functions cpp to post-syn netlist
- Parse DFG, generate floorplan for target PR functions
- Initial overlay place&route

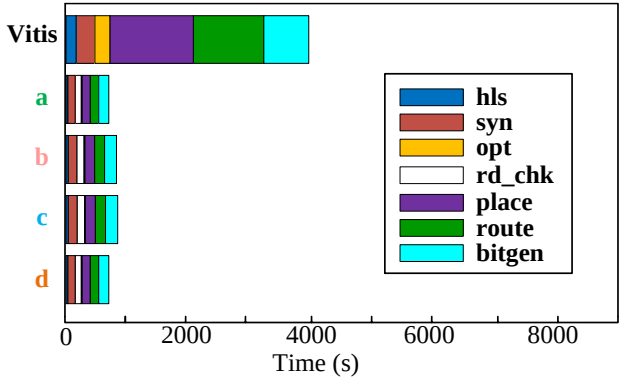


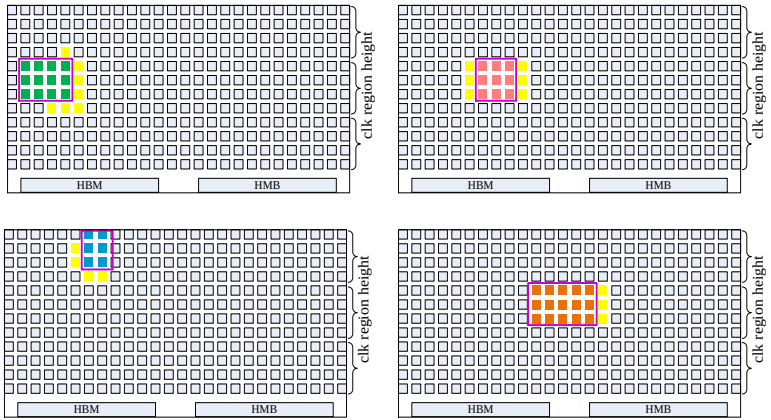


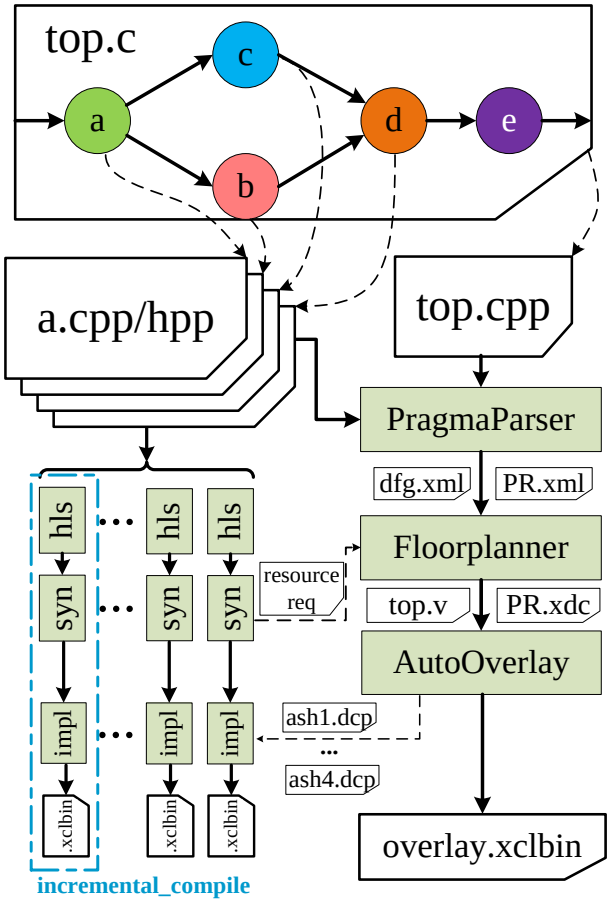


HiPR tool flow: incremental-compile

- Generate abstract-shells for different pages
- Place&route all DFX functions in parallel

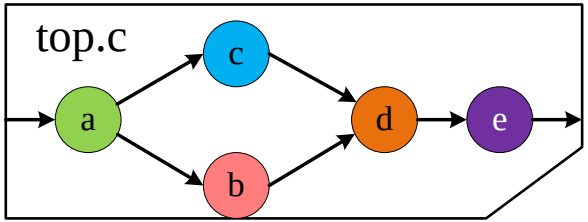




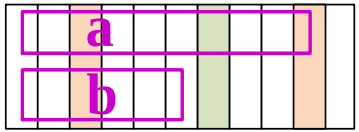


HiPR tool flow: floorplanner

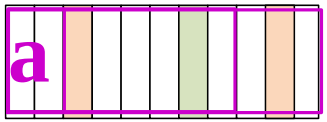
- No Stacked PR blocks within one clock region
- Gradually include more columns of different resources



Operator	CLBs	BRAMs	DSPs
a	1	2	1
b	2	2	1
c	7	2	0
d	8	2	1

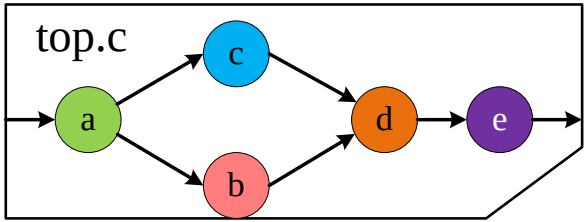


(a) Unsupported Stacked PR



(b) supported Stacked PR

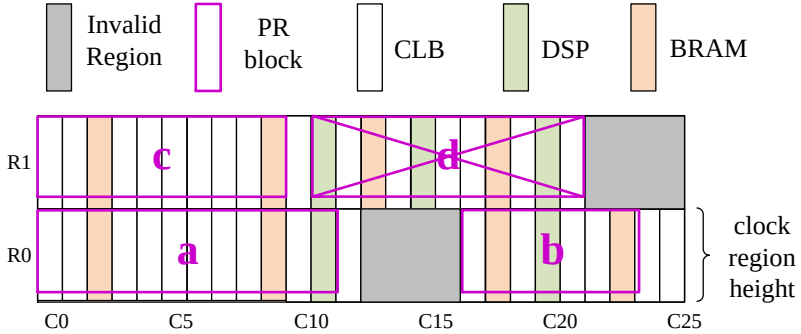
HiPR tool flow: floorplanner



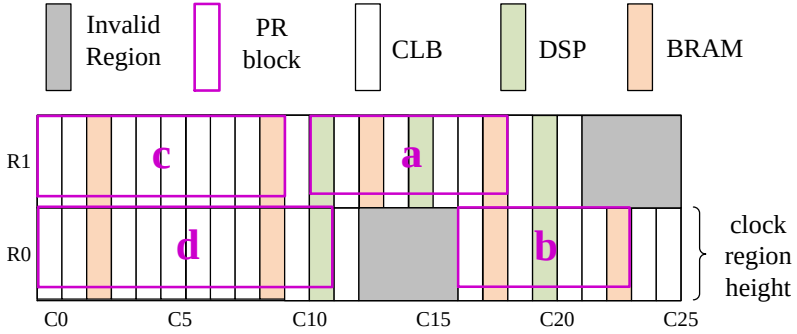
- No Stacked PR region within one clock region
- Gradually include more columns of different resource

Operator	CLBs	BRAMs	DSPs	Place seq 1	Place seq 2
a	1	2	1	1	4
b	2	2	1	2	2
c	7	2	0	3	3
d	8	2	1	4	1

- Generate pblocks with seq 1
- **Could not fit** LUTs requirements for operator d



- Generate pblocks with seq 2
- **Could fit** LUTs requirements for all operators

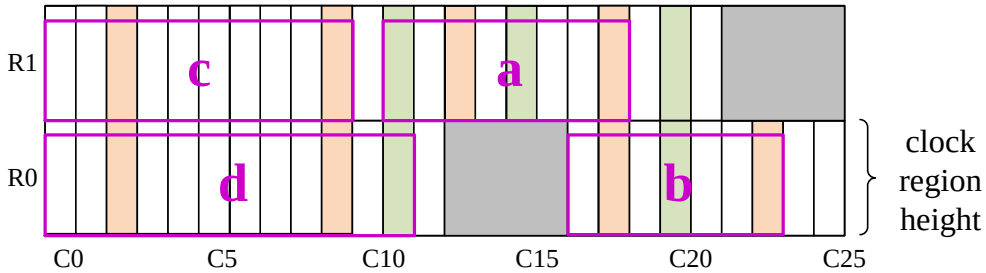


HiPR tool flow: floorplanner

- Use Simulated Annealing to swap the sequence of 2 operators
- Cost Function

$$Update\_cost() = \alpha \sum_{all\ fuctions} Overlapped\ Area + \beta \sum_{all\ links} SquareDistance(src, dest)$$

Operator	CLBs	BRAMs	DSPs	Place seq 1	Place seq 2
a	1	2	1	1	4
b	2	2	1	2	2
c	7	2	0	3	3
d	8	2	1	4	1

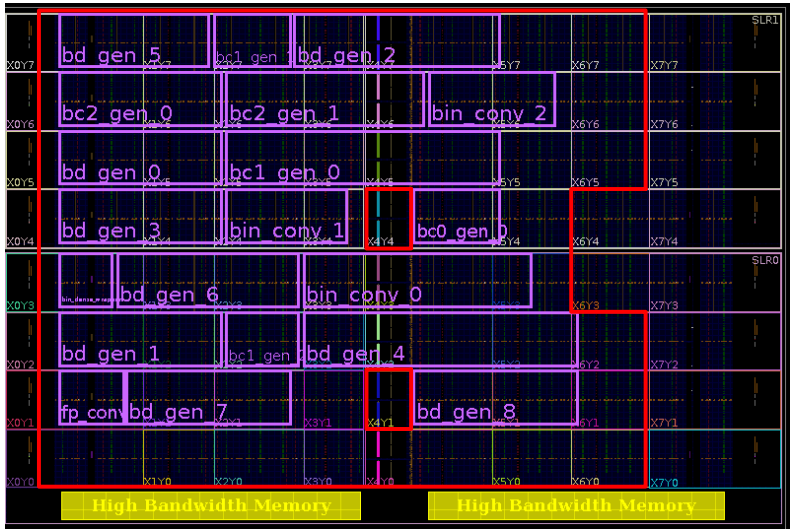


Algorithm 1 Simulated Annealing Floorplanner

```
1:  procedure FLOORPLANNER(Operator Seq)
2:      T ← T_0
3:      i ← 0
4:      Cost_min ← Update_cost()
5:      while Cost_min > 0.1 or i> ITER_MAX do
6:          Randomly swap 2 operators' order
7:          d_cost = Update_cost() - cost_min
8:          if d_cost < 0 then
9:              Cost_min ← Update_cost()
10:             Update Operator Seq
11:         else
12:             if exp(-d_cos/T)>random_possibility then
13:                 Cost_min ← Update_cost()
14:                 Update Operator Seq
15:             end if
16:         end if
17:         T ← η×T
18:         i ← i + 1
19:     end while
20:     return Operator Seq
21: end procedure
```

Evaluation

- Floorplanner Time
  - Take less than 1 second
  - Unneglectable compared with P&R time



Floorplan example for BNN

	PR#	LUTs	BRAM	DSP	Runtime
Rendering	6	5718	64	15	0.050s
Digit Rec	20	40758	320	0	0.006s
Spam Filter	15	11382	12	256	0.004s
Optical Flow	16	18489	84	330	0.086s
Face Detect	20	66654	169	100	0.050s
Binary NN	22	36950	1042	5	0.022s

Evaluation

- Initial-compile Speedup
  - Take more time to make the PR overlay
  - Increase the overhead by 15%-67%

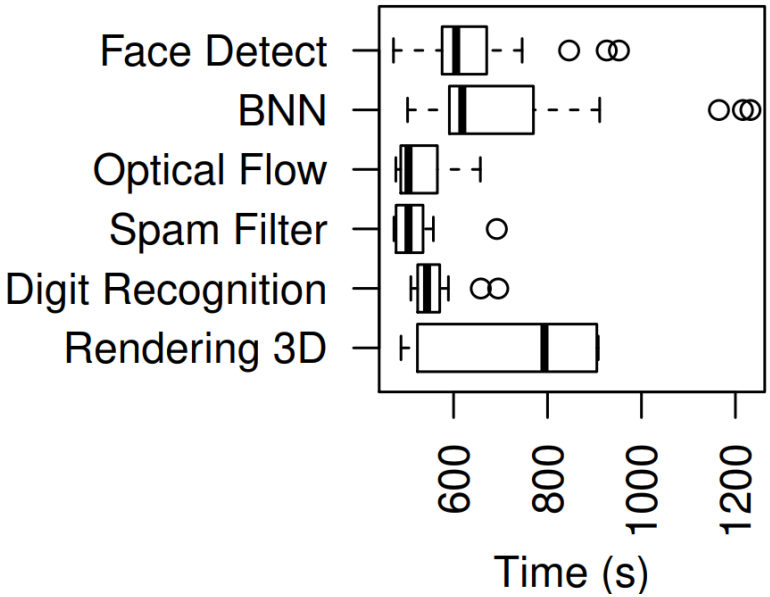
	Vitis	HiPR	
	Total	Total	Overhead
Rendering	4264	7152	67 %
Digit Rec	5173	6125	19 %
Spam Filter	3942	4541	15 %
Optical Flow	4139	6880	66 %
Face Detect	6288	8851	40 %
Binary NN	6584	9632	46 %

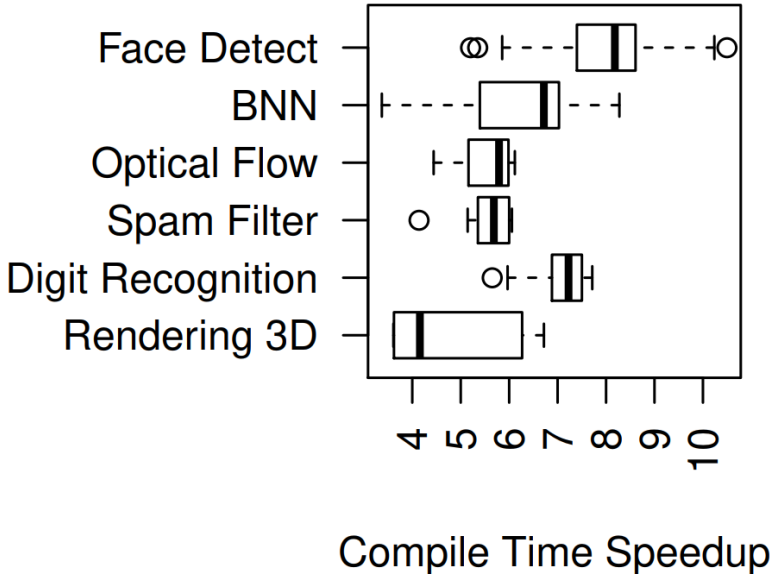
- Incremental-compile Speedup
  - Maximum compile time for all the operators
  - Speedup is 3.4-5.6X

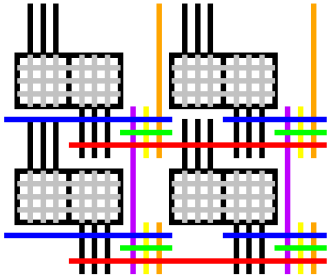
	Vitis	HiPR	
	Total	Total	speedup
Rendering	3278	908	3.6
Digit Rec	3927	695	5.6
Spam Filter	2865	692	4.1
Optical Flow	2918	657	4.4
Face Detect	4954	952	5.2
Binary NN	4154	1232	3.4

Evaluation

- Incremental-compile Speedup
  - Maximum compile time for all the operators
  - Speedup is 3.4-10.5X







Conclusion

- Define Partial Reconfigurable C-functions instead of Verilog modules
- Automate the overlay generation
- Decrease the in-compile time by **3.4-10.5X**