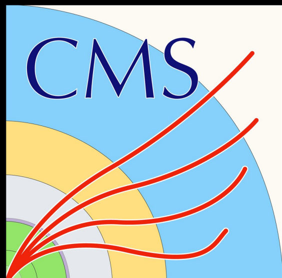


FPGA Design with High Level Synthesis
Methodology, gains, and pitfalls

Michalis Bachtis

University of California, Los Angeles
On behalf of the CMS Collaboration

TWEPP 2023

Introduction

- High-level synthesis (HLS) enables building FPGA firmware in C
- A “compiler” is generating HDL code based on a model that includes information about the FPGA resources and speed
- Since C code is serial, several directives are needed to write parallel code
  - Directives are defined by the user
- Fixed point precision is included in C as additional libraries
- Output product: IP block or HDL code
- Several gains but also pitfalls

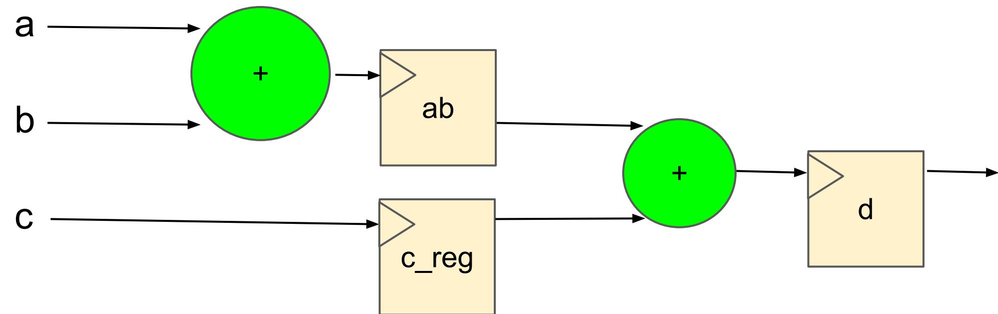
<u>Outline</u>

- Introduction to HLS with some representative examples
  - Using the AMD/Xilinx tools and FPGAs but similar tools exist in other architectures
- Real design implementations from CMS experiment and beyond
- Proposed design methodology
- Discussion

Introduction to High Level Synthesis

Example: HDL adder of three 32 bit numbers

```
module adder(
   input wire clk,
   input wire [31:0] a,
   input wire [31:0] b,
   input wire [31:0] c,
   output reg [33:0] d
);
reg [31:0] c_reg;
reg [32:0] ab;
always @ (posedge clk) begin
    ab<=a+b;
    c_reg<=c;
    d<=ab+c_reg;
end
endmodule
```



- Instantiating a pipeline of two steps
  - Step 1: add a+b → store the output in ab register. register c to c_reg so that the data are aligned in the next clock
  - Step 2: add ab+c_reg to create the output that is stored in a register
- Two clock cycles, latency of 1 cycle
- Output is registered

Example: HLS adder of three 32 bit numbers

Fixed point integer library

32 bit unsigned integer

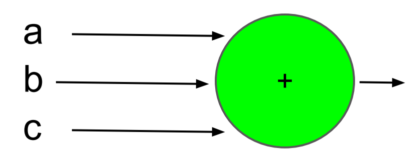
Pragma directives.

Adder logic

```
#include <ap_int.h>
ap_uint<34> adder(const ap_uint<32>& a,
          const ap_uint<32>&b,
          const ap_uint<32>& c) {
#pragma HLS pipeline II=1
#pragma HLS interface ap_none port=a
#pragma HLS interface ap_none port=b
#pragma HLS interface ap_none port=c
#pragma HLS interface ap_ctrl_none port=return
ap_uint<33> ab=a+b;
ap_uint<34> d = ab+c;
return d;
}
```

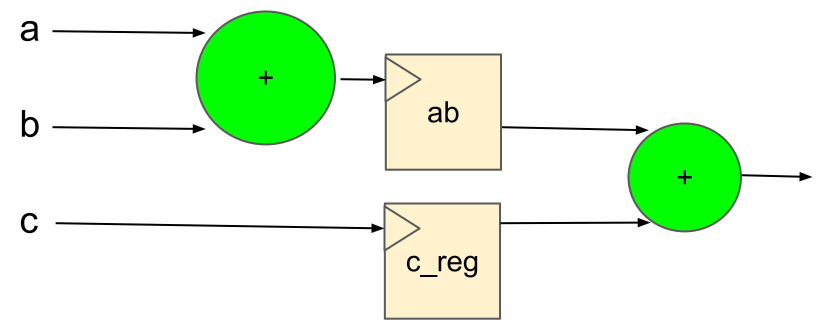
- C is written with fixed point numbers
- No specification of register stages
  - only that the design should be pipelined
- Implementation actually depends on the compiler that needs target clock speed

Clock: 100 MHz



Combinational logic: compiler knows FPGA can do it in 1 clock, latency of 0 cycles

Clock: 500 MHz



Two step pipeline. Latency of 1 cycle The difference with HDL is that it did not register the output. One can add a compiler directive to do this

What do we learn from this trivial example?

```
module adder(
    input wire clk,
    input wire [31:0] a,
    input wire [31:0] b,
    input wire [31:0] c,
    output reg [33:0] d
);
reg [31:0] c_reg;
reg [32:0] ab;
always @ (posedge clk) begin
     ab<=a+b;
     c_reg<=c;
     d<=ab+c_reg;
end
endmodule
```

```
#include <ap_int.h>
ap_uint<34> adder(const ap_uint<32>& a,
           const ap_uint<32>&b,
           const ap_uint<32>& c) {
#pragma HLS pipeline II=1
#pragma HLS interface ap_none port=a
#pragma HLS interface ap_none port=b
#pragma HLS interface ap_none port=c
#pragma HLS interface ap_ctrl_none port=return
ap_uint<33> ab=a+b;
ap_uint<34> d = ab+c;
return d;
}
```

- HLS will make “optimal” logic based on the intended performance
  - But the developer relies on the compiler. Not much personal freedom
- Not that in HDL the developer needs to keep track of aligning the data in the pipeline
  - Painful in complex designs to keep track of everything
  - **HLS automatically registers the data when it creates pipelines**

DSP cores

- DSP cores are silicon cores in AMD/Xilinx FPGAs
  - Perform fast multiplication and other operations
    - Saves many LUTs that would be used for multipliers
  - Are usually instantiated in HDL with xilinx internal libraries

48-Bit Accumulator/Logic Unit
B
A
D
C
+/−
Pre-adder
27 x 18 Multiplier
Pattern Detector
XOR
P
Pattern Detect
X16750-082917

HLS: a+b*c

```
ap_uint<33> dsp(const ap_uint<16>& a,
        const ap_uint<16>&b,
        const ap_uint<16>& c) {
#pragma HLS pipeline II=1
#pragma HLS interface ap_none port=a
#pragma HLS interface ap_none port=b
#pragma HLS interface ap_none port=c
#pragma HLS interface ap_ctrl_none port=return
    ap_uint<32> bc = b*c;
    return a+bc;
}
```

HDL (a+b*c)

- HLS very effective in configuring and instantiating DSPs
- DSPs have variable latency
  - HLS tunes it

Lookup tables

```
const ap_uint<72>  lookup[512] = {
        0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
        21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,
        39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,
        57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,
        75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,
        93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,
        108,109,110,111,112,113,114,115,116,117,118,119,120,
        121,122,123,124,125,126,127,128,129,130,131,132,133,
        134,135,136,137,138,139,140,141,142,143,144,145,146,
        147,148,149,150,151,152,153,154,155,156,157,158,159,
        160,161,162,163,164,165,166,167,168,169,170,171,172,
        173,174,175,176,177,178,179,180,181,182,183,184,185,
        186,187,188,189,190,191,192,193,194,195,196,197,198,
        199,200,201,202,203,204,205,206,207,208,209,210,211,
        212,213,214,215,216,217,218,219,220,221,222,223,224,
        225,226,227,228,229,230,231,232,233,234,235,236,237,
        238,239,240,241,242,243,244,245,246,247,248,249,250,
        251,252,253,254,255,256,257,258,259,260,261,262,263,
        264,265,266,267,268,269,270,271,272,273,274,275,276,
        277,278,279,280,281,282,283,284,285,286,287,288,289,
        290,291,292,293,294,295,296,297,298,299,300,301,302,
        303,304,305,306,307,308,309,310,311,312,313,314,315,
        316,317,318,319,320,321,322,323,324,325,326,327,328,
        329,330,331,332,333,334,335,336,337,338,339,340,341,
        342,343,344,345,346,347,348,349,350,351,352,353,354,
        355,356,357,358,359,360,361,362,363,364,365,366,367,
        368,369,370,371,372,373,374,375,376,377,378,379,380,
        381,382,383,384,385,386,387,388,389,390,391,392,393,
        394,395,396,397,398,399,400,401,402,403,404,405,406,
        407,408,409,410,411,412,413,414,415,416,417,418,419,
        420,421,422,423,424,425,426,427,428,429,430,431,432,
        433,434,435,436,437,438,439,440,441,442,443,444,445,
        446,447,448,449,450,451,452,453,454,455,456,457,458,
        459,460,461,462,463,464,465,466,467,468,469,470,471,
        472,473,474,475,476,477,478,479,480,481,482,483,484,
        485,486,487,488,489,490,491,492,493,494,495,496,497,
        498,499,500,501,502,503,504,505,506,507,508,509,510,511};

ap_uint<73> lookupAndAdd(const ap_uint<32>& a,const ap_uint<9>& addr) {
    return a+lookup[addr];
}
```

- An adder that reads a LUT and and adds a constant
- A LUT of 512x72 bits instantiated
  - In Ultrascale architecture = 1 BRAM
- At 100 MHz
  - Latency of 1 cycle [to read the ROM]
- At 500 MHz
  - Latency of 2 cycles [automatic register in the output of ROM]
- When increasing the width of more than 72 or if we add more than 52 entries
  - Automatically instantiates more BRAMs

Memory

```
static ap_uint<64> memory[256];

ap_uint<64> memory_handshake(const ap_uint<8>& addr,const ap_uint<64>& data, const ap_uint<1>& wen  ) {

#pragma HLS pipeline II=1
    if (wen) {
        memory[addr]=data;
        return 0;
    }
    else {
        return memory[addr];
    }

}
```

- Memory instantiated as global array instantiated outside the method
- Simple handshake, write and read from memory
- Instantiates 2 BRAMs (true dual port)
- Latency of 1 cycle

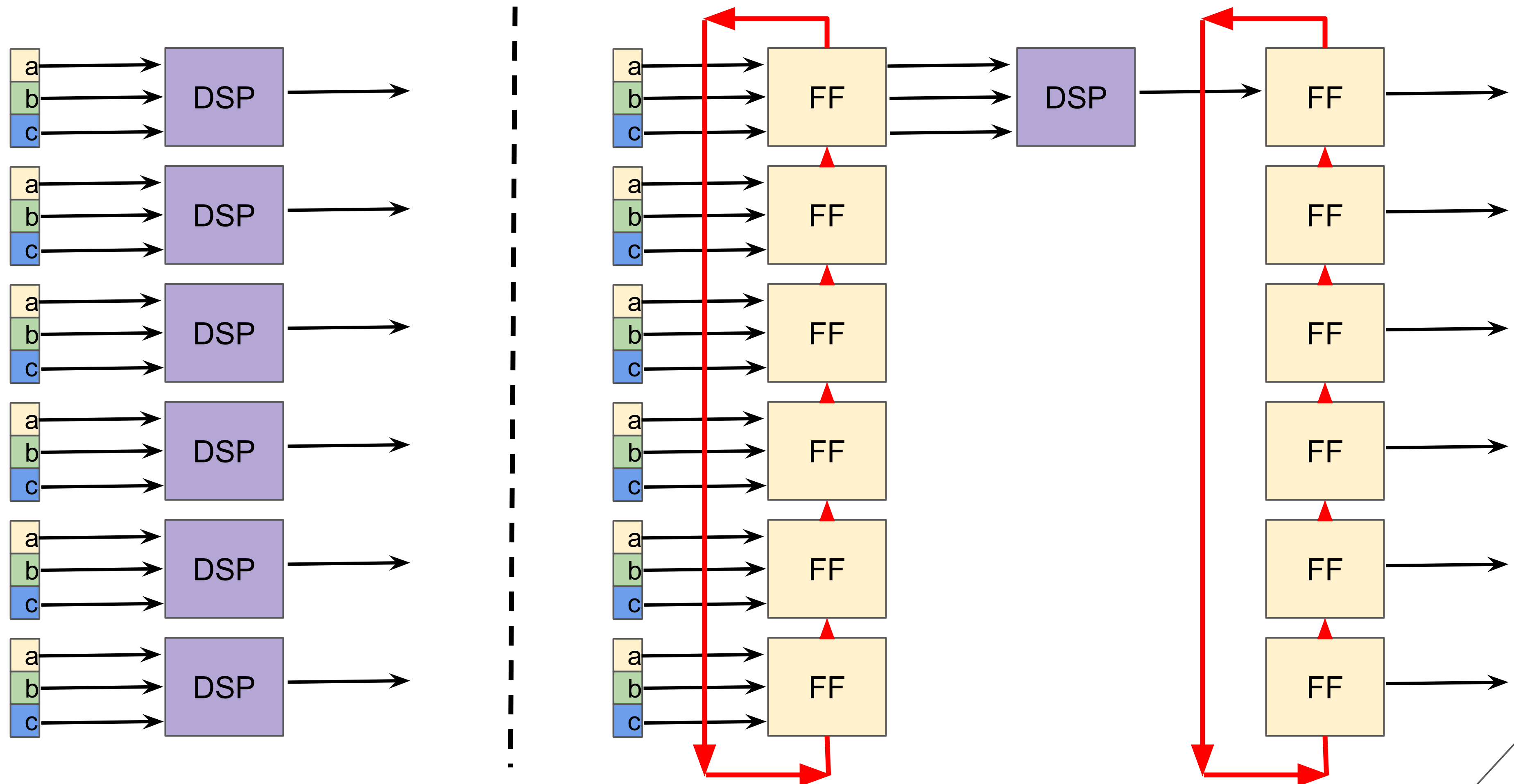
Registers (Flip flops)

```
static ap_uint<64> memory[256];

ap_uint<64> memory_handshake(const ap_uint<8>& addr,const ap_uint<64>& data, const ap_uint<1>& wen  ) {
#pragma HLS array_partition variable=memory complete  dim=1
#pragma HLS pipeline II=1
    if (wen) {
        memory[addr]=data;
        return 0;
    }
    else {
        return memory[addr];
    }

}
```

- The #pragma command breaks the array into individual registers
- Instead of 2 BRAMs it instantiates 16k Flip-flops
- With this pragma, multiple array entries are accessible at any cycle [since it is a register] but without it only one (two) entries can be accessed because it is a BRAM
- From C perspective, the array is defined. Actual implementation depends on #pragma entries

Playing with the pipeline, reusing logic

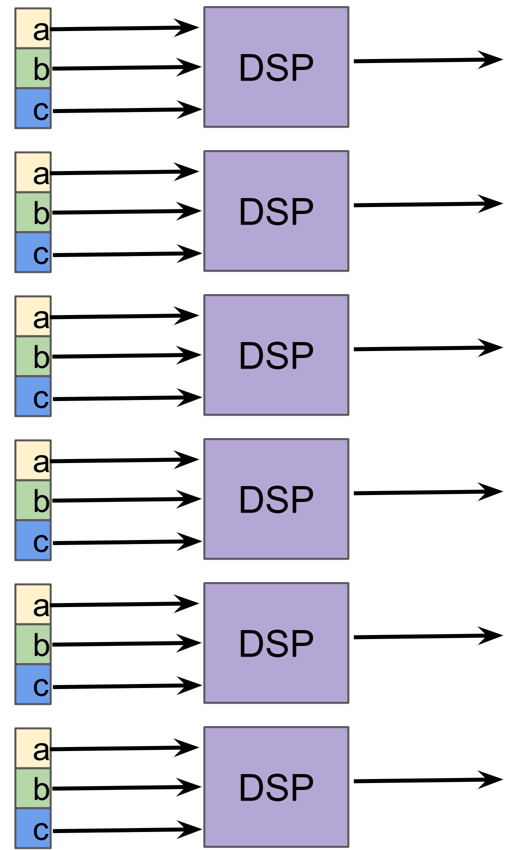
- In Time Multiplexed designs: implement one algorithm core and feed several chunks of data. As an example let's assume:
  - 6 sets of three 16 bit numbers (a,b,c) are arriving in the system
  - For each set we need to calculate a+b*c [with a DSP]
- We have a fully pipelined option and an option to reuse logic



Fully pipelined design

```
typedef struct {
    ap_uint<16> a0; ap_uint<16> b0; ap_uint<16> c0;
    ap_uint<16> a1; ap_uint<16> b1; ap_uint<16> c1;
    ap_uint<16> a2; ap_uint<16> b2; ap_uint<16> c2;
    ap_uint<16> a3; ap_uint<16> b3; ap_uint<16> c3;
    ap_uint<16> a4; ap_uint<16> b4; ap_uint<16> c4;
    ap_uint<16> a5; ap_uint<16> b5; ap_uint<16> c5;
} data_in_t;

out_t fully_pipelined(const data_in_t& in) {

#pragma HLS pipeline II=1

//put the data in arrays as registers for better C++ code
ap_uint<33> a[6];
ap_uint<33> b[6];
ap_uint<33> c[6];
ap_uint<33> d[6];
#pragma HLS array_partition variable=a complete  dim=1
#pragma HLS array_partition variable=b complete  dim=1
#pragma HLS array_partition variable=c complete  dim=1
#pragma HLS array_partition variable=d complete  dim=1

a[0]=in.a0;b[0]=in.b0;c[0]=in.c0;
a[1]=in.a1;b[1]=in.b1;c[1]=in.c1;
a[2]=in.a2;b[2]=in.b2;c[2]=in.c2;
a[3]=in.a3;b[3]=in.b3;c[3]=in.c3;
a[4]=in.a4;b[4]=in.b4;c[4]=in.c4;
a[5]=in.a5;b[5]=in.b5;c[5]=in.c5;


for (unsigned int i=0;i<6;++i) {
#pragma HLS unroll
    d[i] = a[i]+b[i]*c[i];
}

out_t out;
out.d0=d[0];
out.d1=d[1];
out.d2=d[2];
out.d3=d[3];
out.d4=d[4];
out.d5=d[5];


return out;
```

Unroll the loop [like a generate statement]



BRAM	DSP	FF	LUT	URAM
0	6	196	2	0

Latency of 3 cycles @ 500 MHz:
takes new data every cycle
6 DSPs instantiated

DSP reuse design

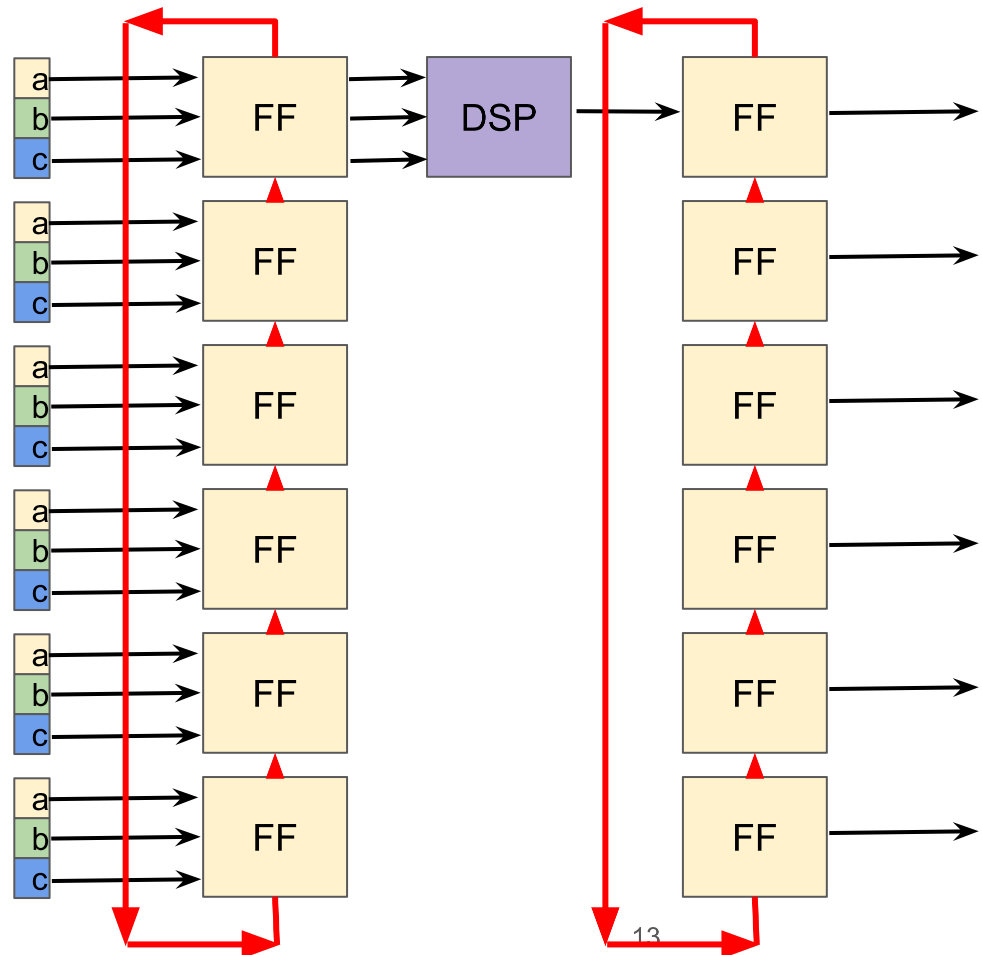
```
typedef struct {
    ap_uint<16> a0; ap_uint<16> b0; ap_uint<16> c0;
    ap_uint<16> a1; ap_uint<16> b1; ap_uint<16> c1;
    ap_uint<16> a2; ap_uint<16> b2; ap_uint<16> c2;
    ap_uint<16> a3; ap_uint<16> b3; ap_uint<16> c3;
    ap_uint<16> a4; ap_uint<16> b4; ap_uint<16> c4;
    ap_uint<16> a5; ap_uint<16> b5; ap_uint<16> c5;
} data_in_t;

out_t dsp_reuse(const data_in_t& in) {
#pragma HLS pipeline II=6
//put the data in arrays as registers for better C++ code
ap_uint<33> a[6];
ap_uint<33> b[6];
ap_uint<33> c[6];
ap_uint<33> d[6];
#pragma HLS array_partition variable=a complete  dim=1
#pragma HLS array_partition variable=b complete  dim=1
#pragma HLS array_partition variable=c complete  dim=1
#pragma HLS array_partition variable=d complete  dim=1

a[0]=in.a0;b[0]=in.b0;c[0]=in.c0;
a[1]=in.a1;b[1]=in.b1;c[1]=in.c1;
a[2]=in.a2;b[2]=in.b2;c[2]=in.c2;
a[3]=in.a3;b[3]=in.b3;c[3]=in.c3;
a[4]=in.a4;b[4]=in.b4;c[4]=in.c4;
a[5]=in.a5;b[5]=in.b5;c[5]=in.c5;

#pragma HLS allocation operation instances=mul limit=1

for (unsigned int i=0;i<6;++i) {
#pragma HLS pipeline II=1
    d[i] = a[i]+b[i]*c[i];
}

out_t out;
out.d0=d[0];
out.d1=d[1];
out.d2=d[2];
out.d3=d[3];
out.d4=d[4];
out.d5=d[5];

return out;
}
```

Allow only one multiplier

Pipeline the loop

BRAM	DSP	FF	LUT	URAM
0	1	539	366	0

- Latency of 7 cycles, one DSP
- New data every 6 cycles



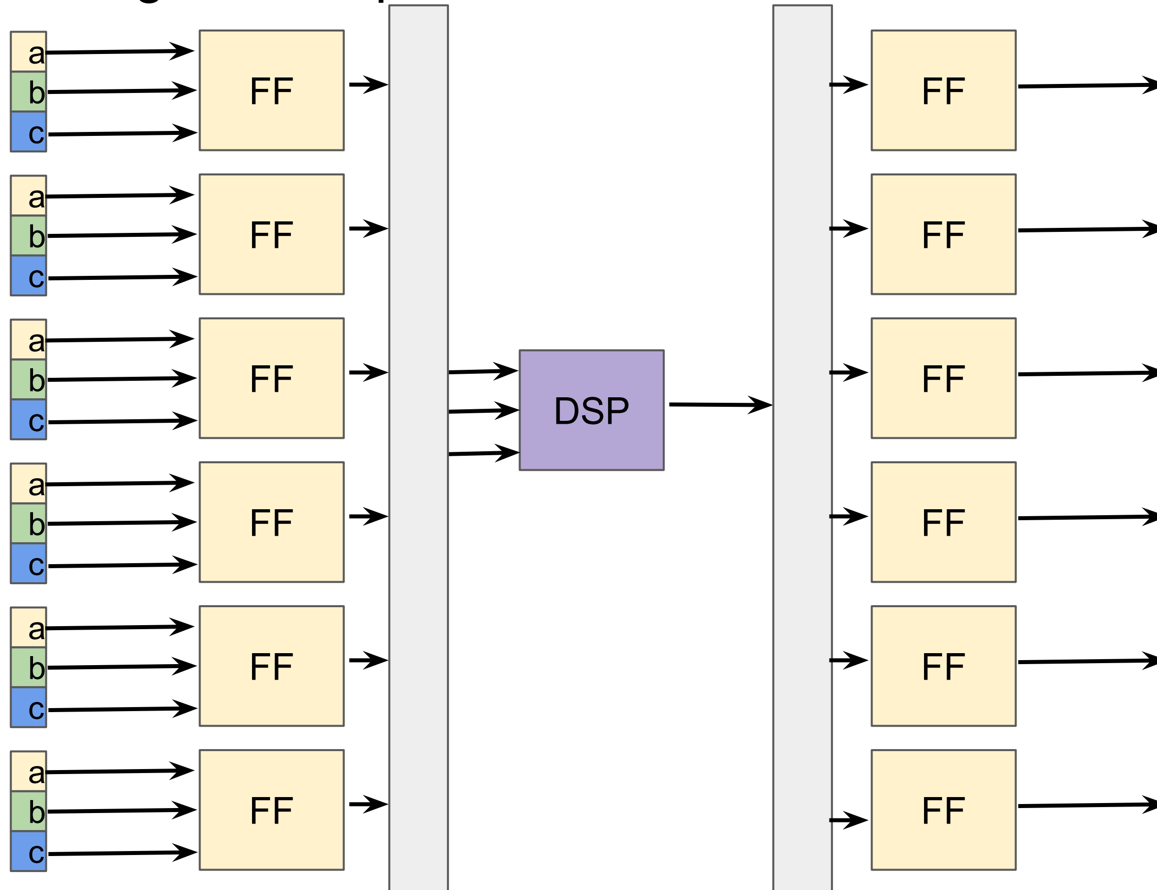
Comparison of the code

```
typedef struct {
    ap_uint<16> a0; ap_uint<16> b0; ap_uint<16> c0;
    ap_uint<16> a1; ap_uint<16> b1; ap_uint<16> c1;
    ap_uint<16> a2; ap_uint<16> b2; ap_uint<16> c2;
    ap_uint<16> a3; ap_uint<16> b3; ap_uint<16> c3;
    ap_uint<16> a4; ap_uint<16> b4; ap_uint<16> c4;
    ap_uint<16> a5; ap_uint<16> b5; ap_uint<16> c5;
} data_in_t;

out_t fully_pipelined(const data_in_t& in) {

#pragma HLS pipeline II=1

//put the data in arrays as registers for better C++ code
ap_uint<33> a[6];
ap_uint<33> b[6];
ap_uint<33> c[6];
ap_uint<33> d[6];
#pragma HLS array_partition variable=a complete  dim=1
#pragma HLS array_partition variable=b complete  dim=1
#pragma HLS array_partition variable=c complete  dim=1
#pragma HLS array_partition variable=d complete  dim=1

a[0]=in.a0;b[0]=in.b0;c[0]=in.c0;
a[1]=in.a1;b[1]=in.b1;c[1]=in.c1;
a[2]=in.a2;b[2]=in.b2;c[2]=in.c2;
a[3]=in.a3;b[3]=in.b3;c[3]=in.c3;
a[4]=in.a4;b[4]=in.b4;c[4]=in.c4;
a[5]=in.a5;b[5]=in.b5;c[5]=in.c5;


for (unsigned int i=0;i<6;++i) {
#pragma HLS unroll
    d[i] = a[i]+b[i]*c[i];
}

out_t out;
out.d0=d[0];
out.d1=d[1];
out.d2=d[2];
out.d3=d[3];
out.d4=d[4];
out.d5=d[5];


return out;
```

```
typedef struct {
    ap_uint<16> a0; ap_uint<16> b0; ap_uint<16> c0;
    ap_uint<16> a1; ap_uint<16> b1; ap_uint<16> c1;
    ap_uint<16> a2; ap_uint<16> b2; ap_uint<16> c2;
    ap_uint<16> a3; ap_uint<16> b3; ap_uint<16> c3;
    ap_uint<16> a4; ap_uint<16> b4; ap_uint<16> c4;
    ap_uint<16> a5; ap_uint<16> b5; ap_uint<16> c5;
} data_in_t;

out_t dsp_reuse(const data_in_t& in) {
#pragma HLS pipeline II=6
//put the data in arrays as registers for better C++ code
ap_uint<33> a[6];
ap_uint<33> b[6];
ap_uint<33> c[6];
ap_uint<33> d[6];
#pragma HLS array_partition variable=a complete  dim=1
#pragma HLS array_partition variable=b complete  dim=1
#pragma HLS array_partition variable=c complete  dim=1
#pragma HLS array_partition variable=d complete  dim=1

a[0]=in.a0;b[0]=in.b0;c[0]=in.c0;
a[1]=in.a1;b[1]=in.b1;c[1]=in.c1;
a[2]=in.a2;b[2]=in.b2;c[2]=in.c2;
a[3]=in.a3;b[3]=in.b3;c[3]=in.c3;
a[4]=in.a4;b[4]=in.b4;c[4]=in.c4;
a[5]=in.a5;b[5]=in.b5;c[5]=in.c5;

#pragma HLS allocation operation instances=mul limit=1

for (unsigned int i=0;i<6;++i) {
#pragma HLS pipeline II=1
    d[i] = a[i]+b[i]*c[i];
}

out_t out;
out.d0=d[0];
out.d1=d[1];
out.d2=d[2];
out.d3=d[3];
out.d4=d[4];
out.d5=d[5];


return out;
}
```

Pitfall. What did the compiler really build?

- We did not specify anything in the code to force a shift register…
  - We could have "helped" the compiler by mimicking the array manipulations in C
- In fact the design was implemented with muxes



- Does it matter?
  - Sometimes it does because in large designs the routing congestion could get the implementation to fail..

What have we learned from the small examples?

- HLS provides the capability to instantiate complex firmware using simple C code + a set of #pragma directives
- HLS tunes the code based on the target clock speed by adding register stages and tuning the internal pipelines of silicon modules [DSPs, BRAMs]
- In complex mathematical operations, the designer focuses on the algorithm and the alignment of data within long pipelines is done by the HLS compiler
  - IMO, the biggest advantage of this approach
- HLS can be configured to manage the data in a smarter way (e.g reuse logic) but the user has to think and check what is the supporting logic that is instantiated by the compiler
  - E.g shift registers vs muxes.
  - Part of the C code can be rewritten to mimic HDL, forcing the data management firmware to what the user wants
- My personal preference is to perform the data management in HDL and connect it with firmware HLS core that implements the computationally intensive algorithms

Real-world designs from CMS

Comparison of HDL vs HLS: OMTF

- In the context of HLS evaluation we implemented the full version of the CMS Overlap Muon Track Finder (OMTF) firmware in both HDL and HLS
  - OMTF performs muon reconstruction in the L1 trigger of CMS

Virtex 7 FPGA

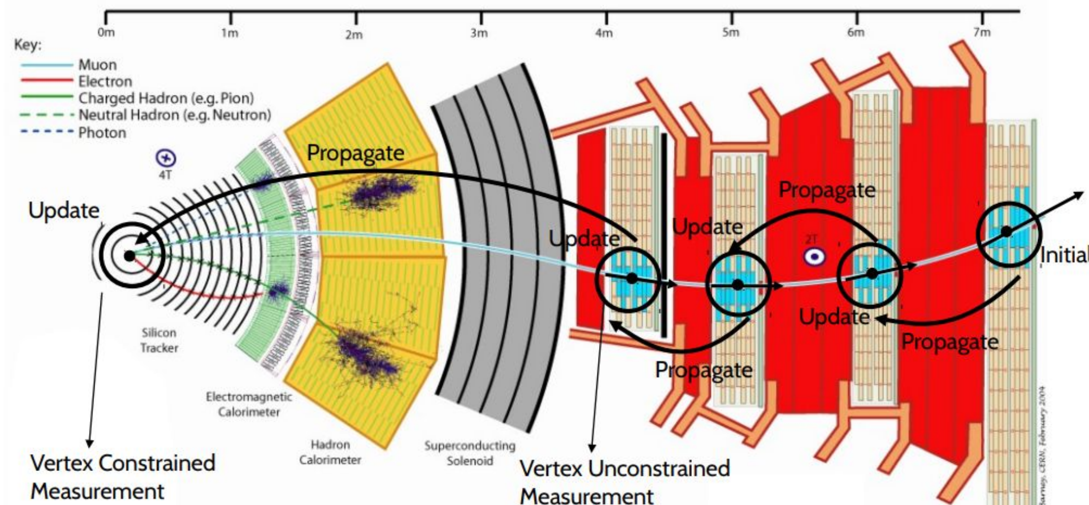
Design method	Slice LUTs	Slice Regis-ters	F7 Muxes	F8 Muxes	Slices	LUT as Logic	LUT as Memory	Block RAM Tile
HLS	123964 (28.6%)	112240 (12.9%)	437 (0.2%)	33 (0.03%)	39312 (36.3%)	106900 (24.7%)	17064 (9.8%)	360 (24.49%)
HDL	114791 (26.5%)	92845 (10.7%)	272 (0.13%)	39 (0.04%)	40607 (37.5%)	108905 (25.1%)	5886 (3.4%)	360 (24.49%)

- Very similar performance for both approaches
- Proof that one does not have to pay a big price in FPGA resources to gain from the HLS approach

An approximate Kalman Filter in CMS Run-2 (I/II)

- CMS Barrel Muon Track finder upgrade for Run-3
- Muon tracking at FPGAs
  - Used to be lookup table based
- New approaches: Real trajectory reconstruction with approximate Kalman Filter
  - Propagation of parameters
  - Fitting
- Requires many calculations
  - Exploit DSP cores

TWEPP 2018(!)



$$x_n = \begin{pmatrix} k \\ \phi \\ \phi_b \end{pmatrix}_n = \begin{pmatrix} 1 & 0 & 0 \\ a & 1 & b \\ c & 0 & d \end{pmatrix} \begin{pmatrix} k \\ \phi \\ \phi_b \end{pmatrix}_{n-1}$$

F

multiple scattering

$$P_{n+1} = F P_n F^T + Q$$

H

$$z_k = \begin{pmatrix} \phi_s \\ \phi_{bs} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} k \\ \phi \\ \phi_b \end{pmatrix}$$

$$y_n = z - H x_n$$

$$S = HPH^T + R$$

position error

$$K = PH^T S^{-1}$$

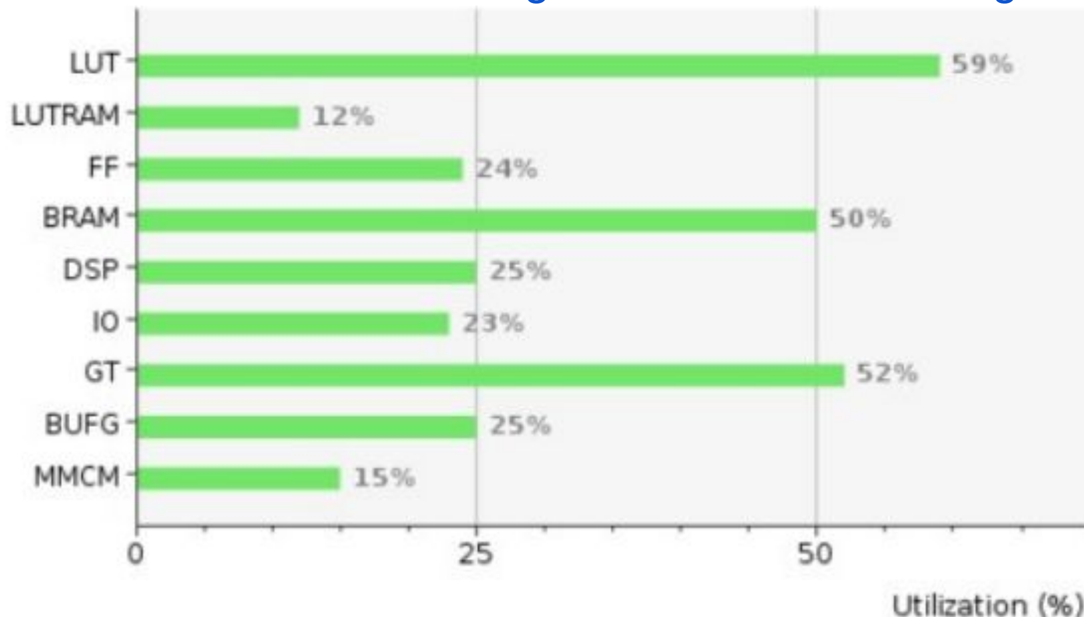
matrix inversion

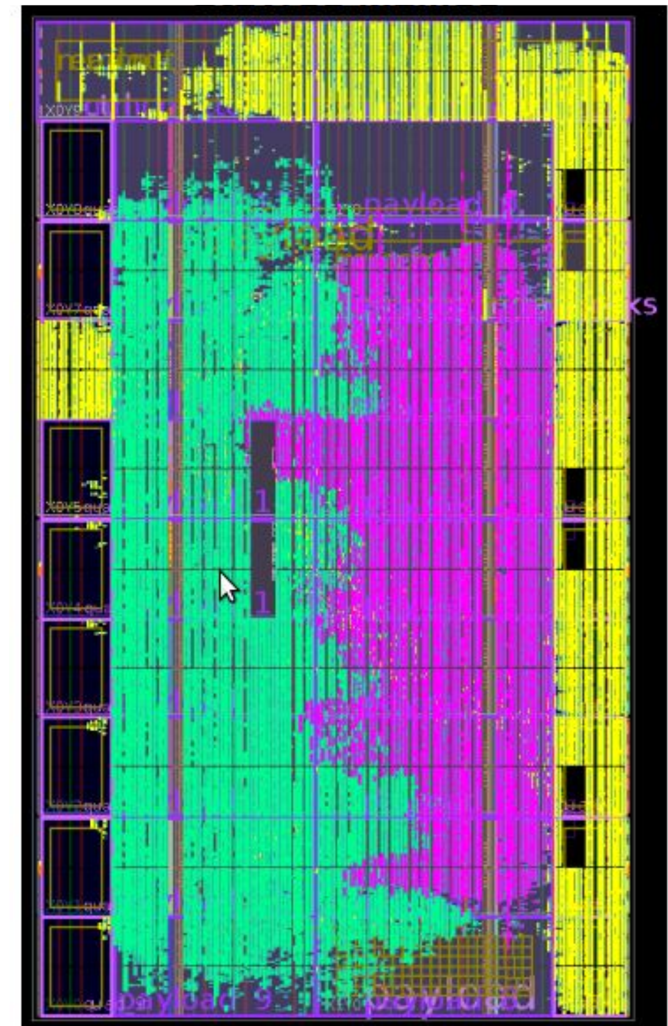
$$x = x_n + K y_n$$

Kalman Gain

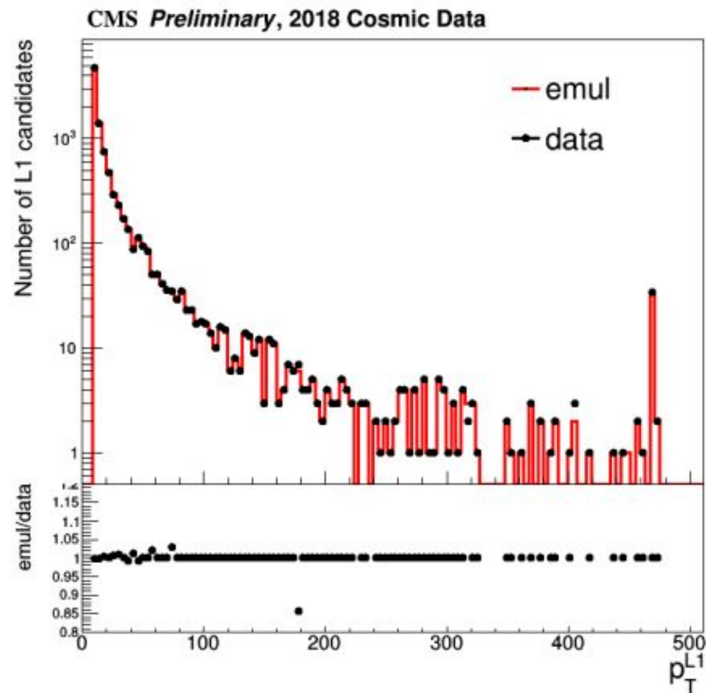
An approximate Kalman Filter in CMS Run-2 (II/II)

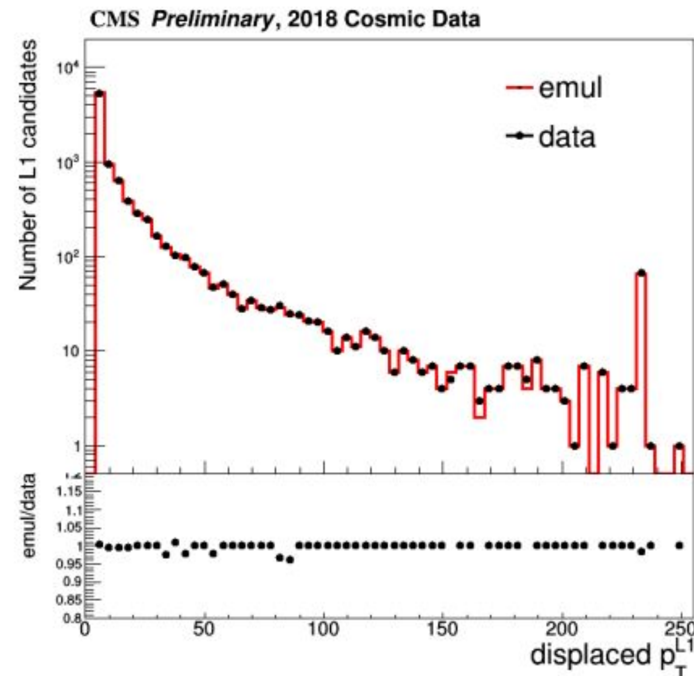
- New Kalman Filter version written in HLS
- Instantiated into the same FPGA (together with the nominal algorithm)
  - Took data in parallel
- It did fit the latency of about 250 ns!
- Nominal algorithm since the beginning of Run-3
  - First HLS running firmware in data taking





A gain of HLS: Faster data vs emulator agreement



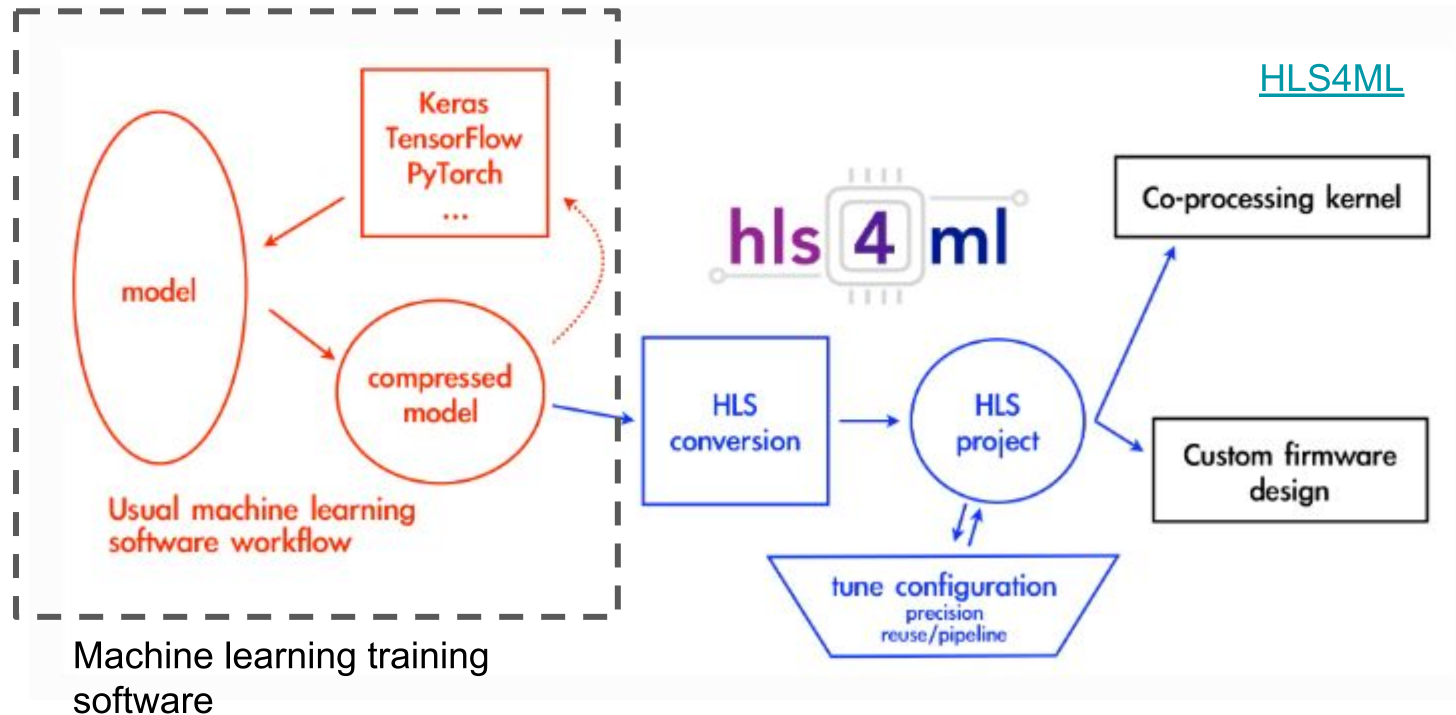


- Firmware emulation → very important for HEP experiments
- With HLS we can load the C fixed point libraries in our emulation code and make the firmware and software agree much faster
  - Usually took years !
- Example: Kalman Filter Muon Track Finder was at 100% agreement already at the beginning of Run-3

Another usage: Real Time Machine Learning - HLS4ML

1804.06913

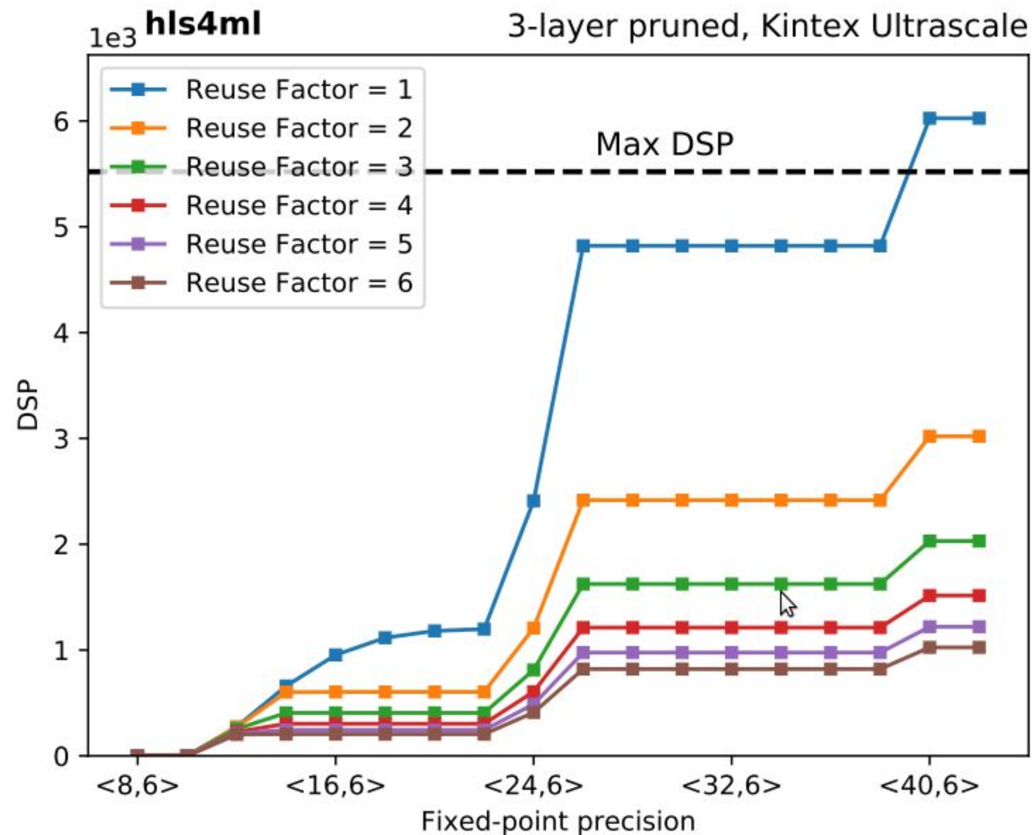
- Fast deep neural networks in FPGAs with HLS



- Translate output of ML training tools to HLS code
  - With fixed point precision
  - Prune nodes that do not have a major impact
- Optimize the design by **reusing logic** and picking the speed
  - With HLS, reuse logic for very big networks
- Feed IP core into the FPGA

Example from HLS4ML: reusing logic

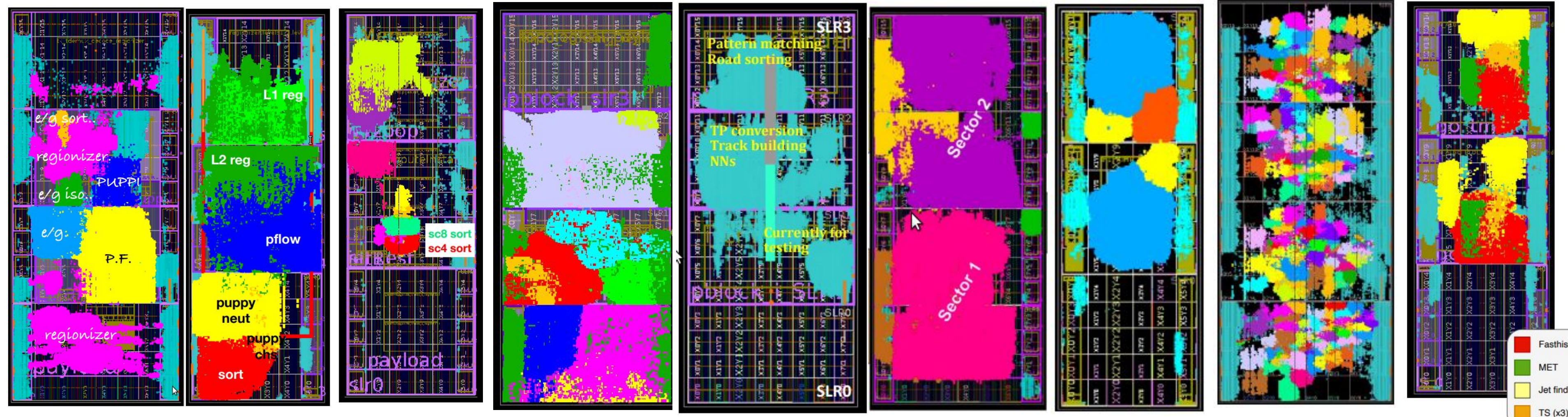
1804.06913



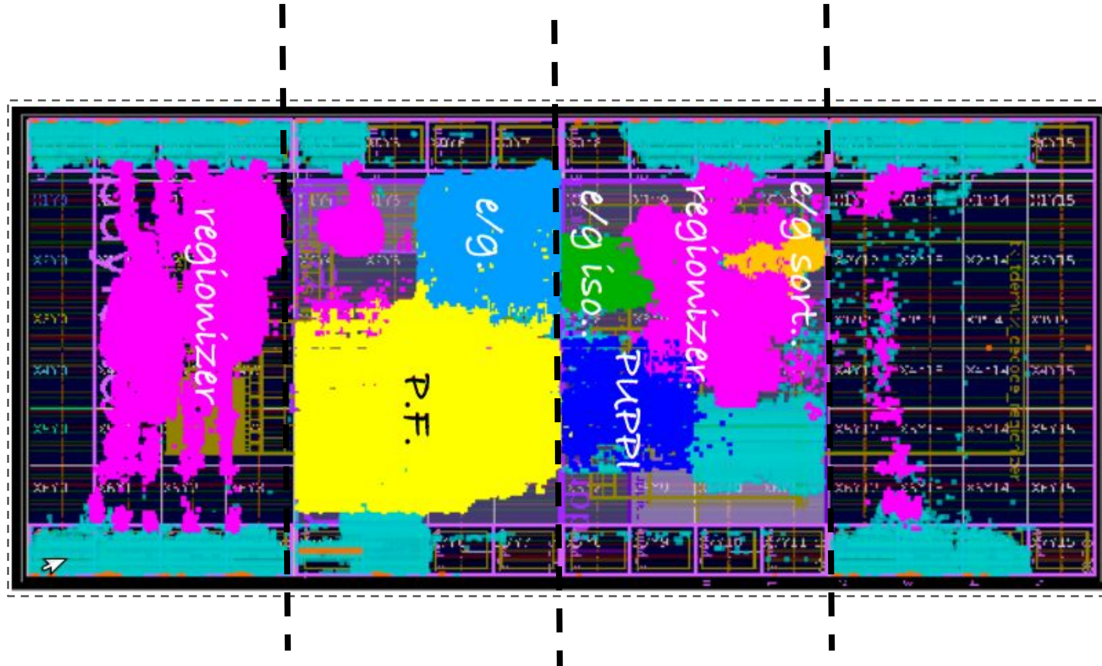
- Using the HLS reuse logic allows to sacrifice throughput for resources
  - But without having to recode the firmware !

The Phase-2 CMS Trigger

- HLS thoroughly used in the CMS Trigger Upgrade for HL-LHC
- Allowed us to train students and postdocs in firmware developments so that they can provide algorithm cores in the final system
- Allowed to have the same developer for firmware and emulator in the experiment software
- Engineering effort focused on infrastructure and of course putting things together



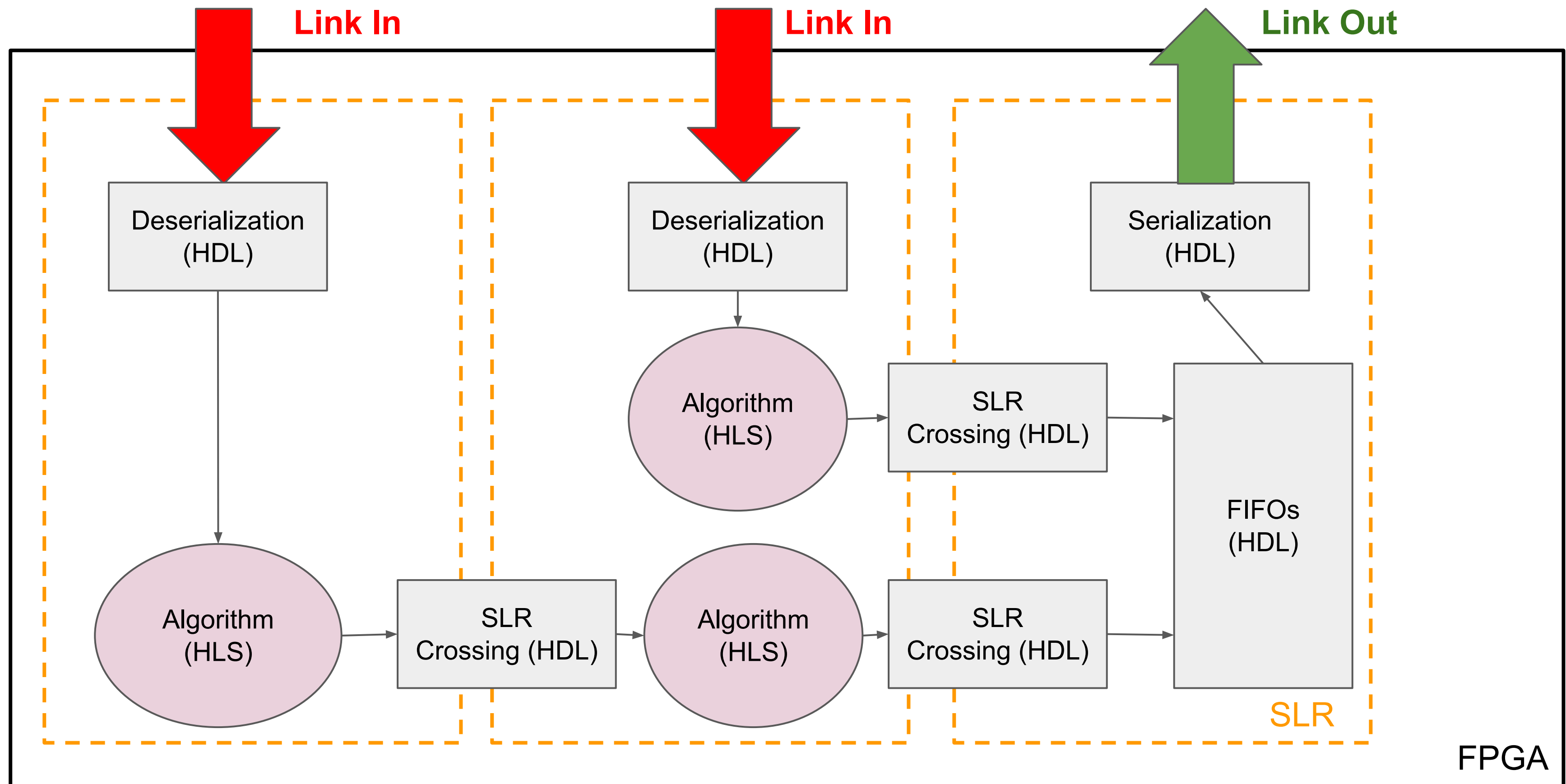
Stacked Silicon Interconnect + HLS



- Modern devices split in different regions (SLRs)
- Some limitations in crossing between them
- Standard HLS modules do not nominally cross
    - There are some tools that do
- This suggests splitting the design with several small HLS modules
- And use HDL “glue” to connect them together
- Also improves compilation time - HLS compiler is slow!

Summary of findings and Methodology

HLS algorithm + HDL data management+glue



- Optimal results obtained by both HLS cores and HDL
- Algorithms → HLS
- Data management, SLR crossing etc → HDL

Proposed guidelines for algorithm development in HLS

- Many small designs more optimal than a very big one
  - Compiler much more efficient
- Two stages of simulation
  - Simulation of C code → Check fixed point precision and algorithm output
    - Also compiles C with GCC and checks if the C code is correct
  - C/RTL co-simulation → Verify that the compiler did the right thing
    - Almost always true except in very aggressive complex designs
  - Simulate the IP core with the rest of the HDL code (e.g in AMD/Xilinx Vivado)
- Pipelining and logic reuse
  - My preference → fully pipelined HLS cores + logic to feed data in HDL
  - Compiler is very effective with fully pipelined designs and you avoid the pitfall of muxes vs shift registers etc
- If the compiler takes many hours, something is going wrong
  - Memory intensive!
  - Sometimes it happens in correct designs
    - Compiler is a black box

Summary

- High Level Synthesis is becoming the new norm not only in particle physics but also in the industry
    - The AI revolution and FPGA acceleration of software routines have pushed companies to deliver efficient and powerful tools
- In designs present in particle physics experiments where often custom link protocols deliver the data a combination of HDL and HLS is optimal
    - Data management with HDL, Algorithms in HLS avoids common pitfalls
- Independently of the technical gains, there is the human factor
    - HLS is the first step to train a non-expert in FPGA firmware
    - Non-experts can become exports in HLS and deliver complex designs
    - Eventually they also learn HDL (!)