An Introduction to Vitis High Level Synthesis

Duc Tri Nguyen

Algorithm

1 Vector-vector Multiplication

Given vector a, b, c, calculate vector e, the result is sum all elements in e multiply with $alpha$.

$$\begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_{n-1} \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ \cdot \\ \cdot \\ \cdot \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \cdot \\ \cdot \\ \cdot \\ e_{n-1} \end{pmatrix} \tag{1}$$

$$E = (e_0 + e_1 + e_2 + \cdots + e_{n-1}) = \sum_{i=0}^{n-1} e_i \tag{2}$$

$$S = alpha * E \tag{3}$$

Vector-vector multiplication as shown in (1).
Sum of vector in (2).
Scalar multiplication shown in (3).

Algorithm

No further algorithmic optimization

1 Vector-vector Multiplication

Given vector a, b, c, calculate vector e, the result is sum all elements in e multiply with $alpha$.

$$\begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_{n-1} \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ \cdot \\ \cdot \\ \cdot \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \cdot \\ \cdot \\ \cdot \\ e_{n-1} \end{pmatrix} \tag{1}$$

$$E = (e_0 + e_1 + e_2 + \cdots + e_{n-1}) = \sum_{i=0}^{n-1} e_i \tag{2}$$

$$S = alpha * E \tag{3}$$

Vector-vector multiplication as shown in (1).
Sum of vector in (2).
Scalar multiplication shown in (3).

Algorithm

(1) Enable vertical optimization
(2) Enable horizontal optimization
(3) No optimization

1 Vector-vector Multiplication

Given vector a, b, c, calculate vector e, the result is sum all elements in e multiply with $alpha$.

$$\begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_{n-1} \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ \cdot \\ \cdot \\ \cdot \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \cdot \\ \cdot \\ \cdot \\ e_{n-1} \end{pmatrix} \tag{1}$$

$$E = (e_0 + e_1 + e_2 + \cdots + e_{n-1}) = \sum_{i=0}^{n-1} e_i \tag{2}$$

$$S = alpha * E \tag{3}$$

Vector-vector multiplication as shown in (1).
Sum of vector in (2).
Scalar multiplication shown in (3).

Algorithm

(1) Enable vertical optimization
(2) Enable horizontal optimization
(3) No optimization

Data input in flow:

A -> B -> C

1 Vector-vector Multiplication

Given vector a, b, c, calculate vector e, the result is sum all elements in e multiply with $alpha$.

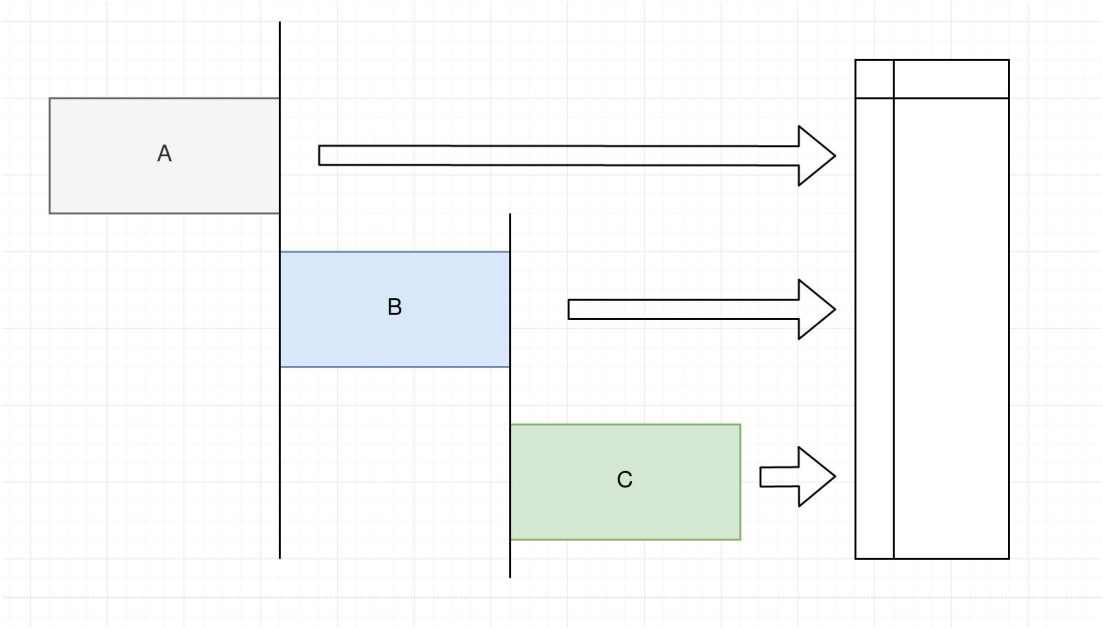
$$\begin{pmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ \cdot \\ \cdot \\ \cdot \\ b_{n-1} \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ \cdot \\ \cdot \\ \cdot \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} e_0 \\ e_1 \\ \cdot \\ \cdot \\ \cdot \\ e_{n-1} \end{pmatrix} \tag{1}$$

$$E = (e_0 + e_1 + e_2 + \cdots + e_{n-1}) = \sum_{i=0}^{n-1} e_i \tag{2}$$

$$S = alpha * E \tag{3}$$

Vector-vector multiplication as shown in (1).
Sum of vector in (2).
Scalar multiplication shown in (3).

Input Flow



Reference C

```
3   u32 hls_vector_mul(const u32 a[N], const u32 b[N], const u32 c[N])
4   {
5       const u32 mask = 0x1fffffff;
6       u32 sum = 0;
7       for (auto i = 0; i < N; i++)
8       {
9           sum += (c[i] + a[i] * b[i]) & mask;
10      }
11      sum &= mask;
12      sum *= ALPHA;
13
14      return sum;
15  }
16
```

Setup

1. Setup Testbench

Return 1 if ERROR

Return 0 if CORRECT

vector_mul_tb.cpp

```
e_gold = vector_mul(a, b, c);
e_hls = hls_vector_mul(a, b, c);

cout << "gold: " << e_gold << endl;
cout << " HLS: " << e_hls << endl;

if (e_gold != e_hls)
{
    cout << "Error" << endl;
    return 1;
}

cout << "OK" << endl;

return 0;
```

Setup

2. Setup Header

Header is used to link the testbench and the HLS code.

C vector_mul.h

```
u32 hls_vector_mul(const u32 a[N], const u32 b[N], const u32 c[N]);

void hls_vector_mul_top(hls::stream<trans_pkt> &fifo_in,
                        hls::stream<trans_pkt> &fifo_out);
```

Setup

3. HLS function

HLS function is
same as C reference code

vector_mul.cpp

```
u32 hls_vector_mul(const u32 a[N], const u32 b[N], const u32 c[N])
{
    const u32 mask = 0x1fffffff;
    u32 sum = 0;
    for (auto i = 0; i < N; i++)
    {
        sum += (c[i] + a[i] * b[i]) & mask;
    }
    sum &= mask;
    sum *= ALPHA;

    return sum;
}
```

Baseline

The baseline is from the C reference implementation

1. Does not have Hardware Interface
2. Is not accelerated
3. Is a start point

Task: Improve performance of Function in High Level Synthesis

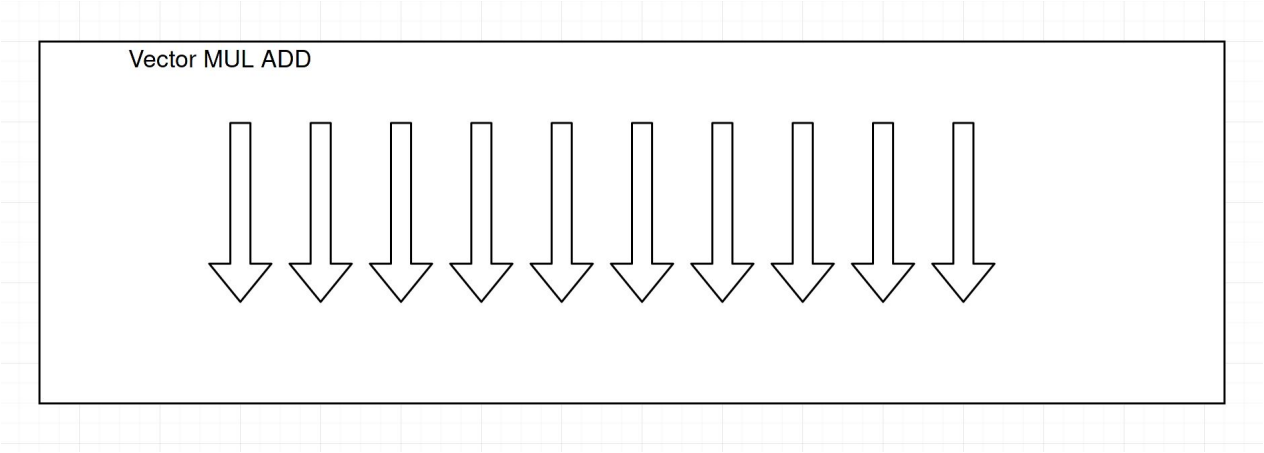
Demo Baseline

C simulation

Synthesis demo

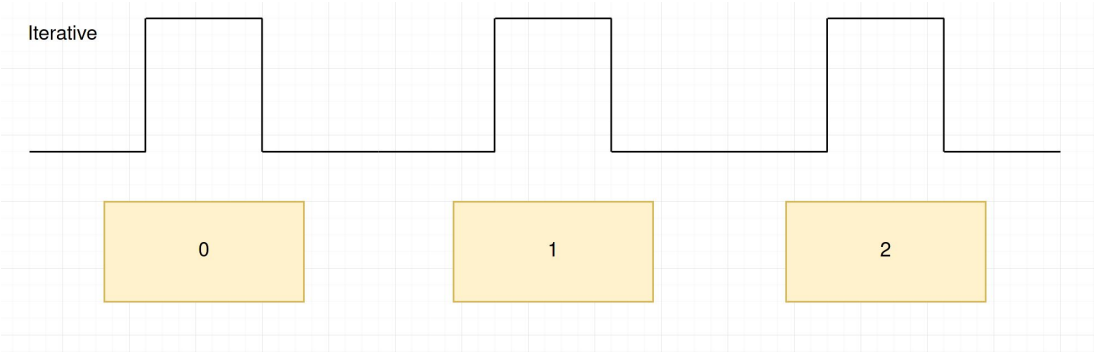
Optimization Strategy

(1) Vertical optimization



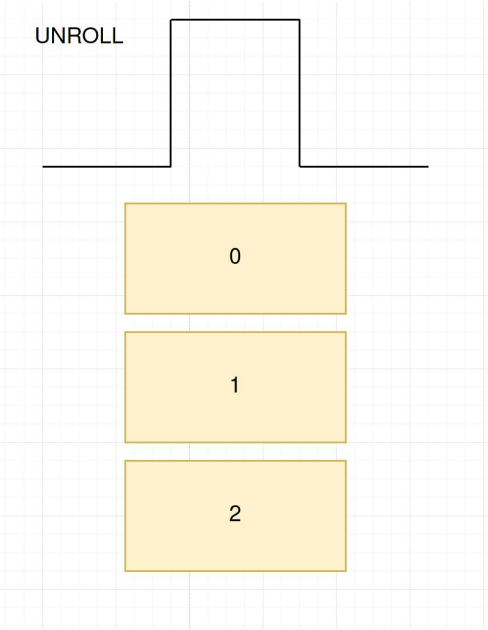
#pragma HLS UNROLL

Motivation



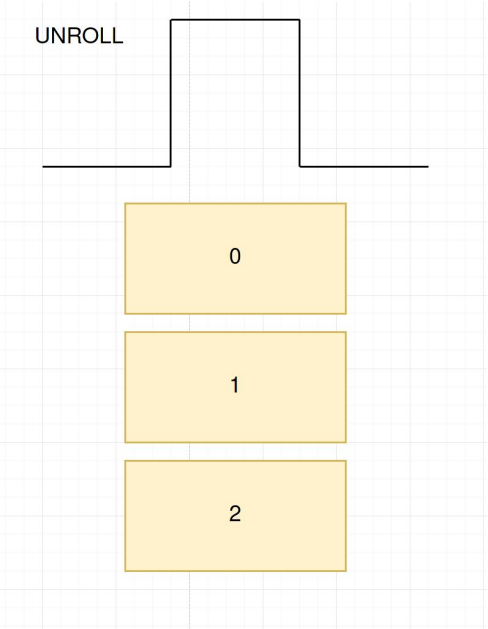
#pragma HLS UNROLL

- Parallel execution in **single** clock cycle
- Maximum performance



Problem with #pragma HLS UNROLL

- Fan-out
- Decrease frequency significantly
- Resources blow up

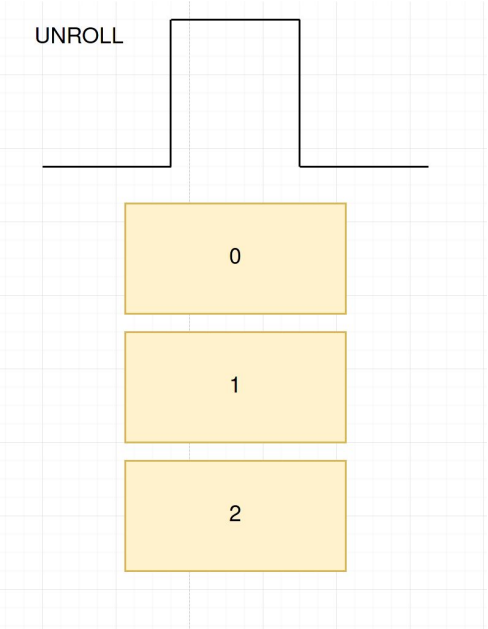


#pragma HLS UNROLL

Demo

C Simulation

Synthesis



Problem with UNROLL

- Cannot meet target frequency
- Resource utilization blow up
- Huge adder tree ← Fan-out Problem

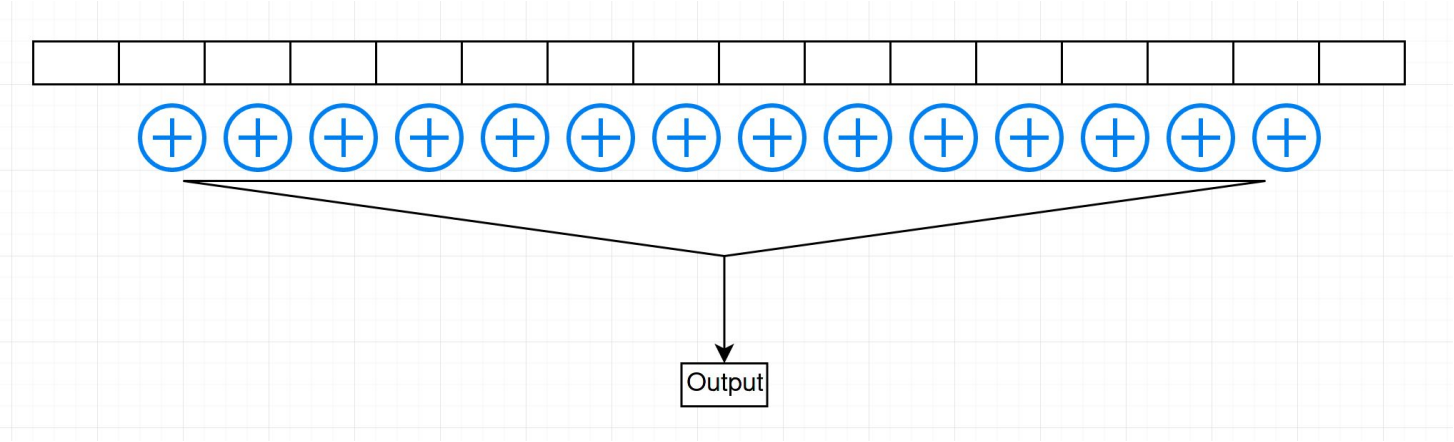
Target	Estimated	Uncertainty
100.0MHz	109.84MHz	370.37MHz

Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul		-1.80	111	1.110E3	-	112	-	no	0	0	54	91	0

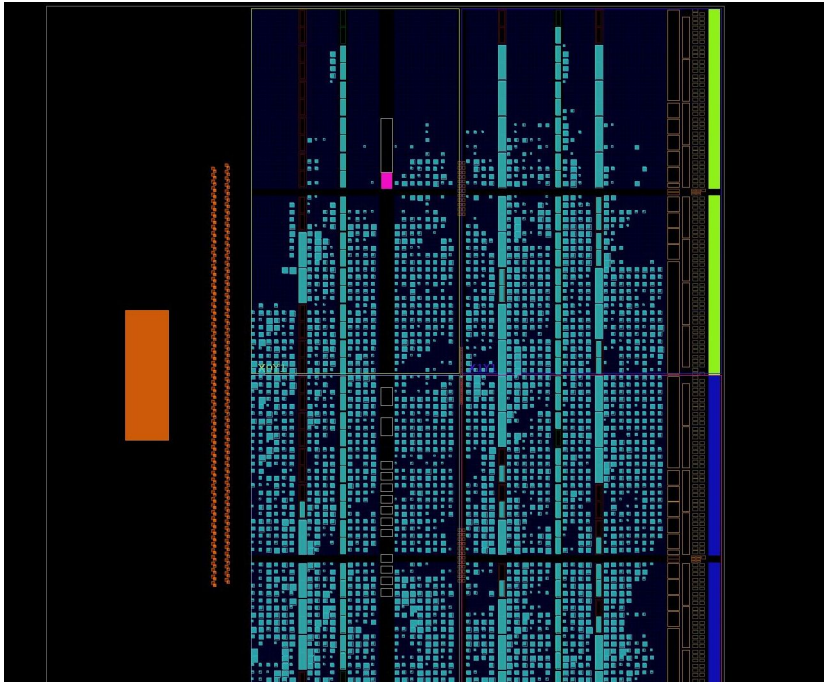
Fan-out Adder Tree



Huge adder tree decrease performance

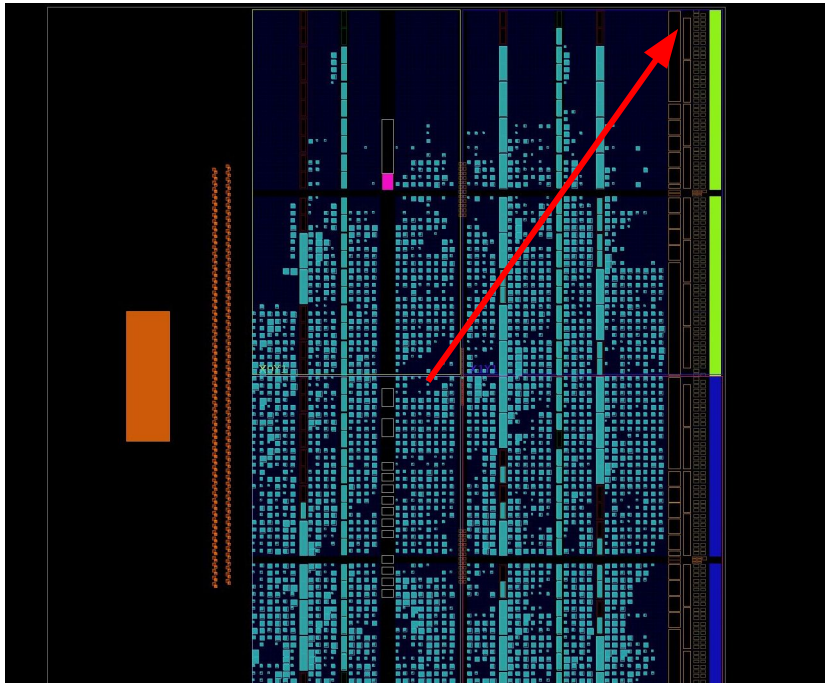
Fanout Problem

- Long critical path



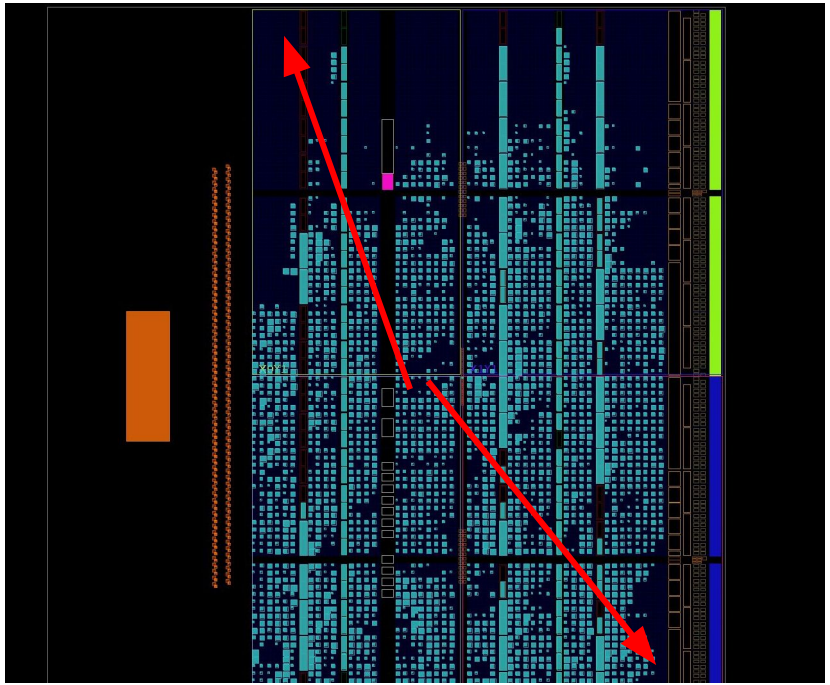
Fanout Problem

- Long critical path



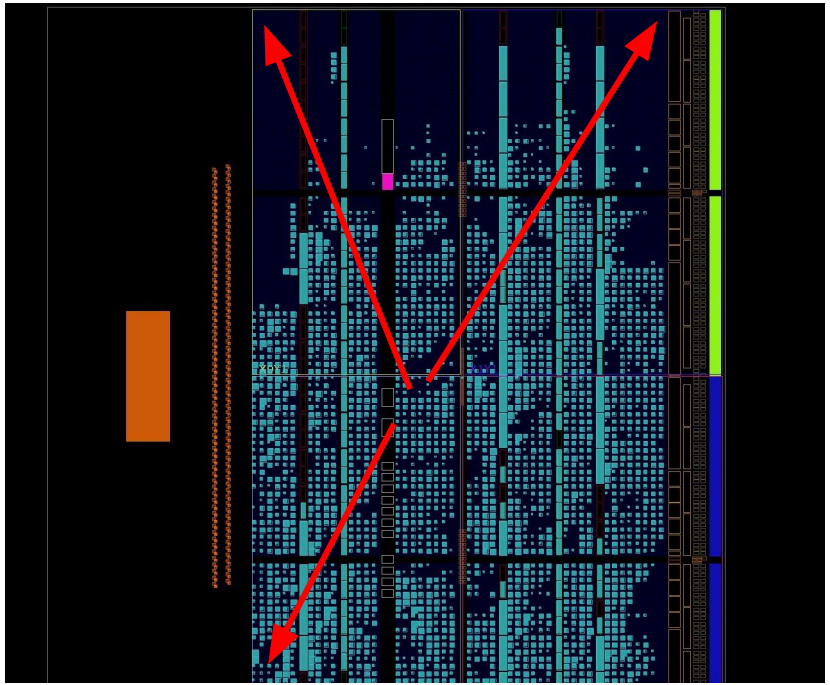
Fanout Problem

- Long critical path



Fanout Problem

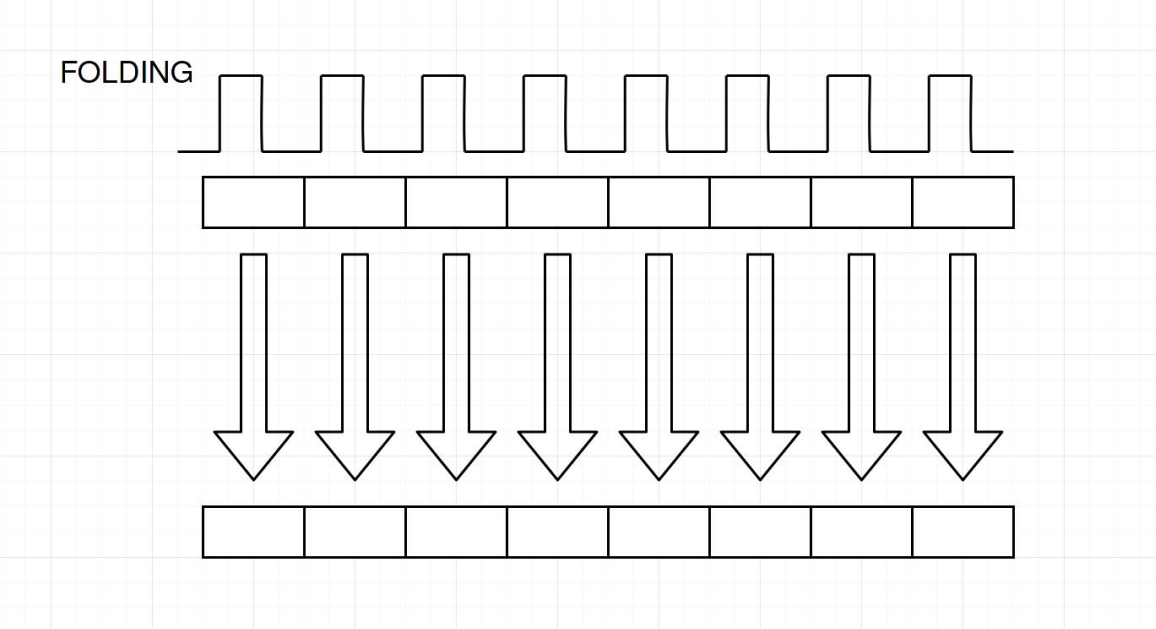
- Long critical path



Partial UNROLL or “Folding”

Simple explanation:

1. Chop big loop to smaller loop



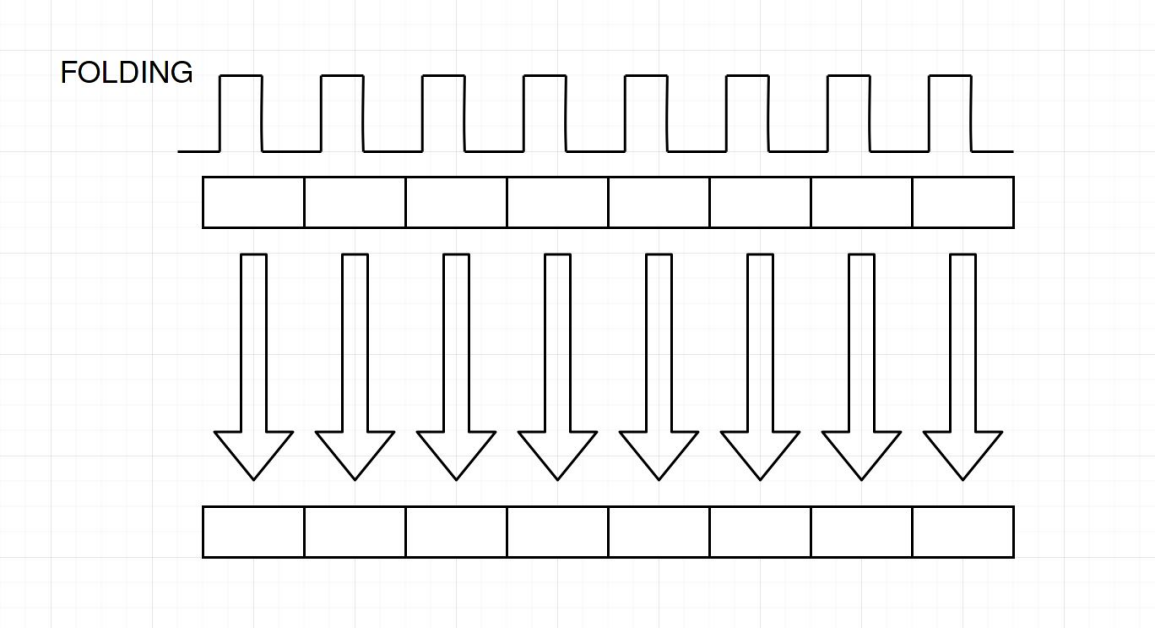
Partial UNROLL or “Folding”

Simple explanation:

1. Chop big loop to smaller loop

eh…?

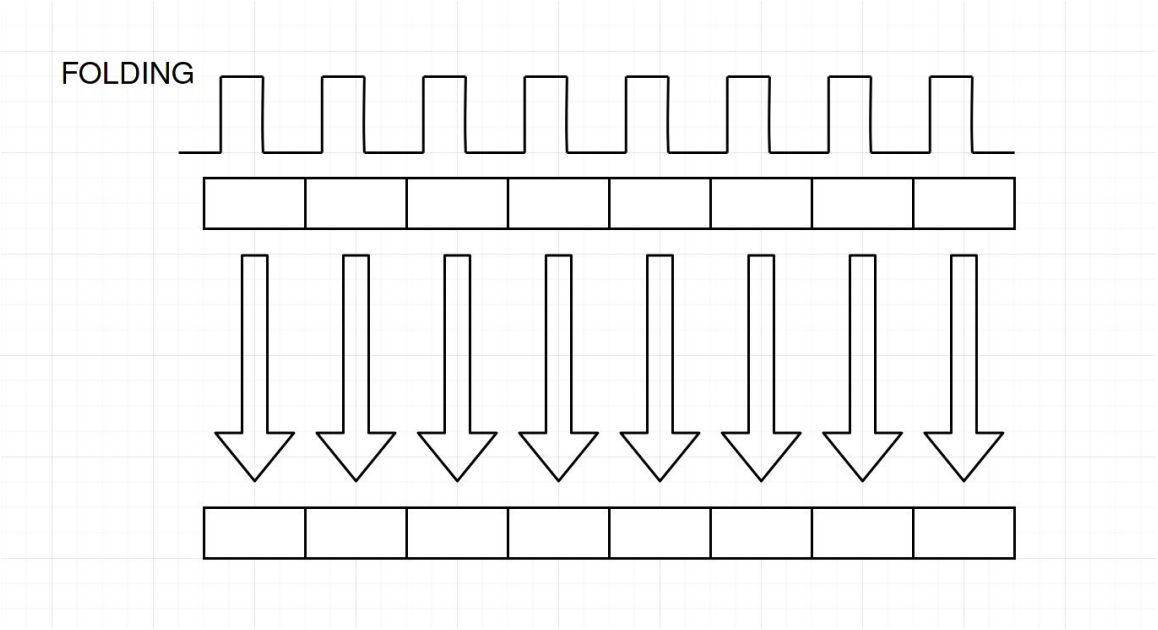
2. Chop Chop Chop



Folding Demo

Demo

- C Simulation
- Synthesis



Problem with UNROLL or “Folding”

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul	II Violation	-0.56	167	1.670E3	-	168	-	no	1	0	16	31	0
calc	II Violation	-	103	1.030E3	24	20	5	yes	-	-	-	-	-
adder_tree		-	40	400.000	2	1	40	yes	-	-	-	-	-

HW Interfaces

Problem with UNROLL or “Folding”

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul	II Violation	-0.56	167	1.670E3	-	168	-	no	1	0	16	31	0
calc	II Violation	-	103	1.030E3	24	20	5	yes	-	-	-	-	-
adder_tree		-	40	400.000	2	1	40	yes	-	-	-	-	-

HW Interfaces

- Violations
- Save dozen of cycles
- Do not meet timing

Loop merge

```
init_sum:
    for (auto i = 0; i < BUFFER; i++)
    {
#pragma HLS UNROLL
        sum[i] = 0;
    }

calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

Loop merge

We spent cycles to initialize array

Can we do better?

```
init_sum:
    for (auto i = 0; i < BUFFER; i++)
    {
#pragma HLS UNROLL
        sum[i] = 0;
    }

calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

Loop merge

Demo

- C Simulation

```
init_sum:
    for (auto i = 0; i < BUFFER; i++)
    {
#pragma HLS UNROLL
        sum[i] = 0;
    }

calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

Loop merge

The loop philosophy of HLS is simple:

One. Big. Loop

You will realize this, after reading ~600 pages Xilinx guide.

```
calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            if (i == 0)
            {
                sum[j] = static_cast<u32>(0);
            }
            else{
                sum[j] = sum[j];
            }
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

Violation

```
calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            if (i == 0)
            {
                sum[j] = static_cast<u32>(0);
            }
            else{
                sum[j] = sum[j];
            }
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

What the heck is this?

Violation

```
calc:
    for (auto i = 0; i < N; i+=BUFFER)
    {
        for (auto j = 0; j < BUFFER; j++)
        {
#pragma HLS UNROLL
            if (i == 0)
            {
                sum[j] = static_cast<u32>(0);
            }
            else{
                sum[j] = sum[j];
            }
            sum[j] += (c[i + j] + a[i + j] * b[i + j]) & mask;
        }
    }
```

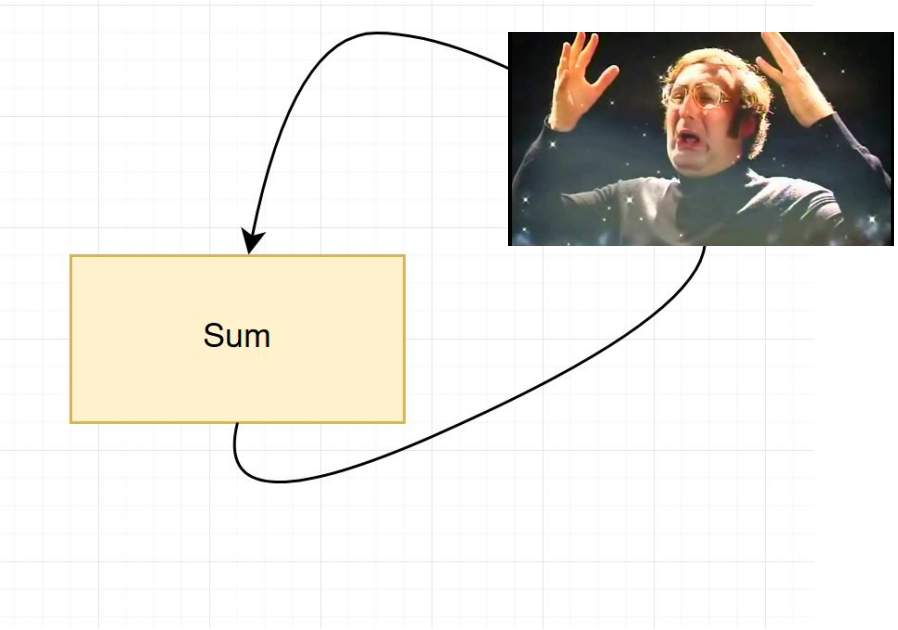
What the heck is this?

Violation

Resolve Violation

Demo

- C simulation
- Synthesis



#pragma HLS ARRAY_RESHAPE

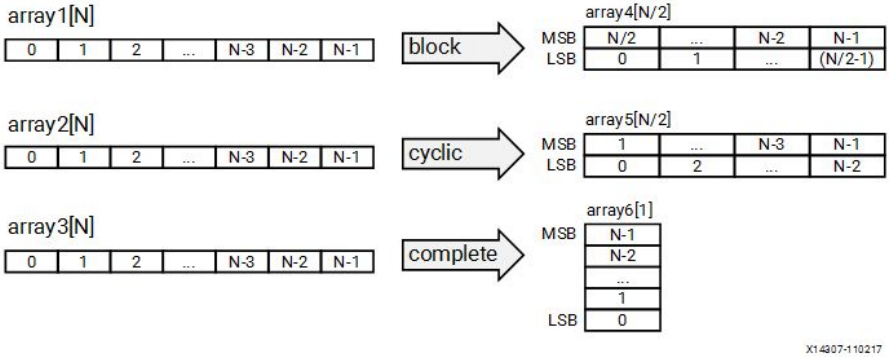
factor=2

Left side:

- Spent N clock cycles to read N elements in array

Right side:

- **Block**: 1 clock cycle read 2 elements **distance by N/2**



#pragma HLS ARRAY_RESHAPE

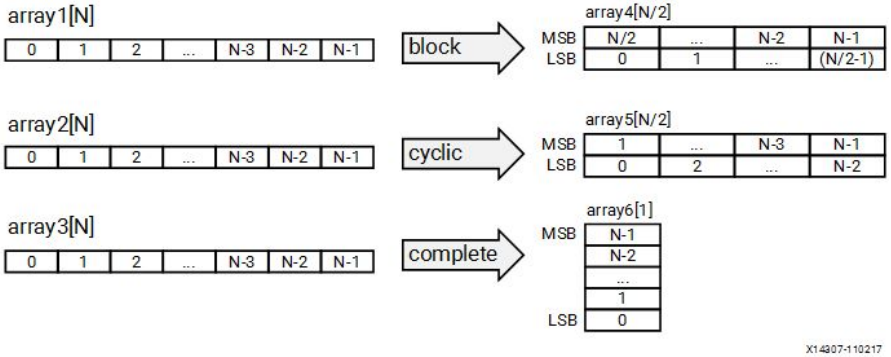
factor=2

Left side:

- Spent N clock cycles to read N elements in array

Right side:

- **Cyclic**: 1 clock cycle read 2 elements **distance by 1**



#pragma HLS ARRAY_RESHAPE

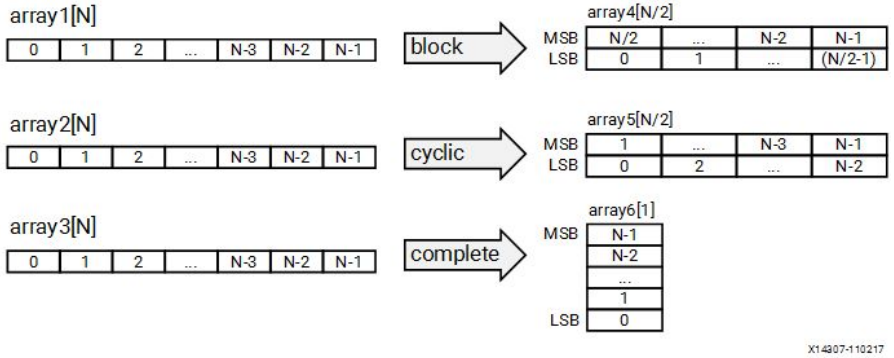
factor=2

Left side:

- Spent N clock cycles to read N elements in array

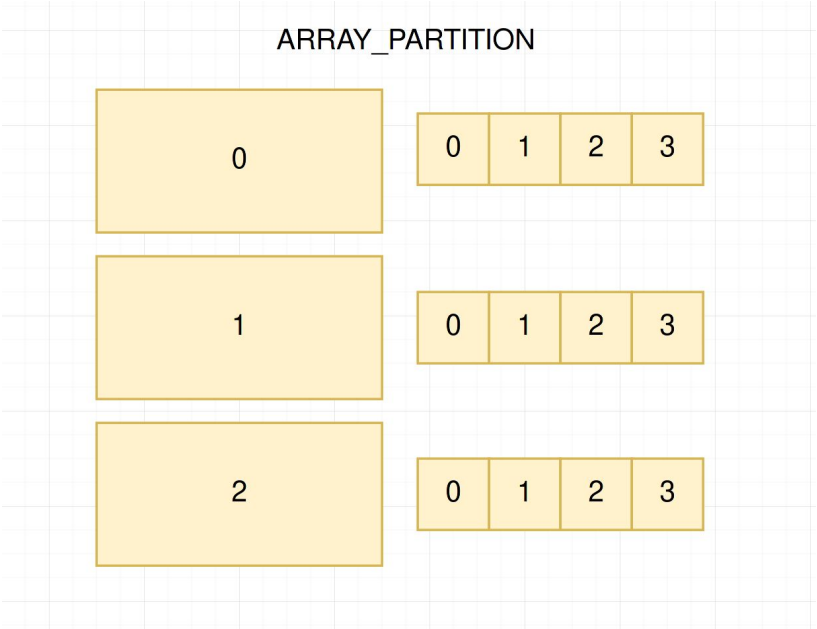
Right side:

- **Complete**: 1 clock cycle read **all** elements



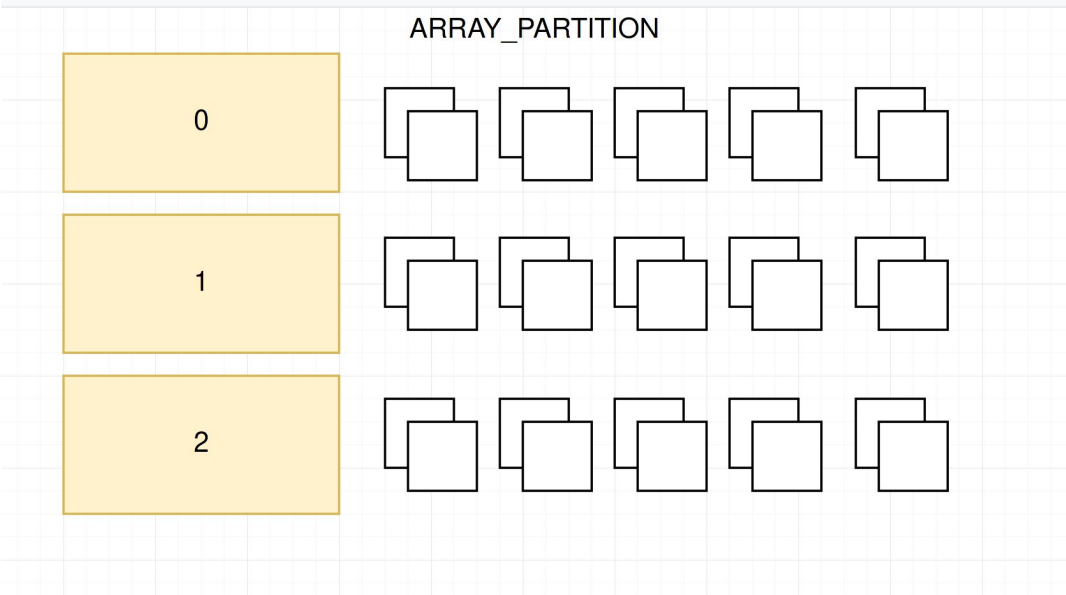
#pragma HLS ARRAY_PARITION

- Similar to ARRAY_RESHAPE
- Easier to control multi-dimensional array



#pragma HLS ARRAY_PARITION

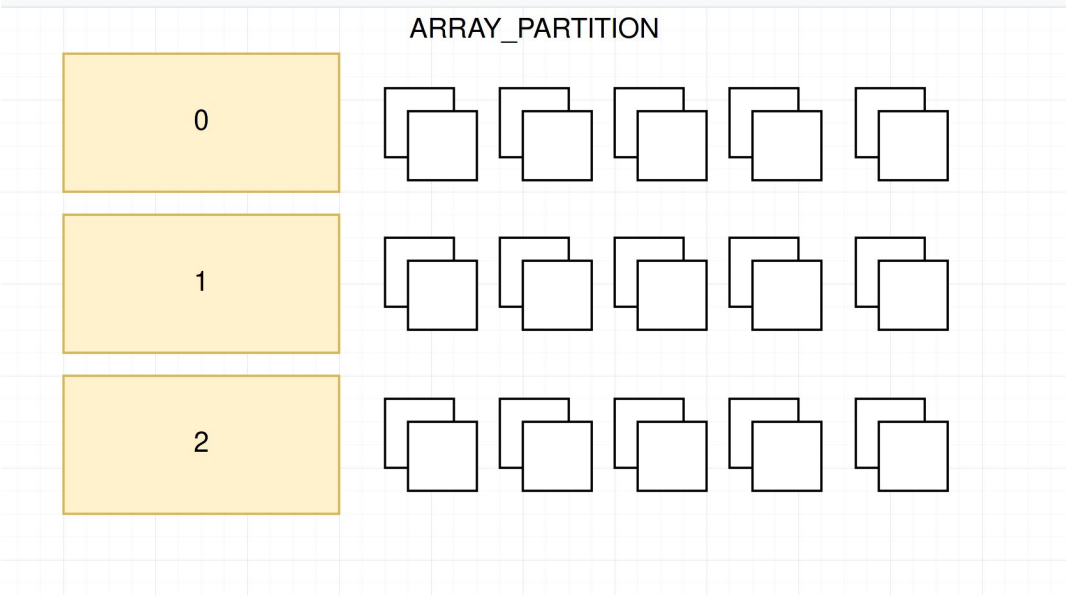
- Similar to ARRAY_RESHAPE
- Easier to control multi-dimensional array



ARRAY_PARITION vs ARRAY_RESHAPE

Two pragma are very similar, when to use what ?

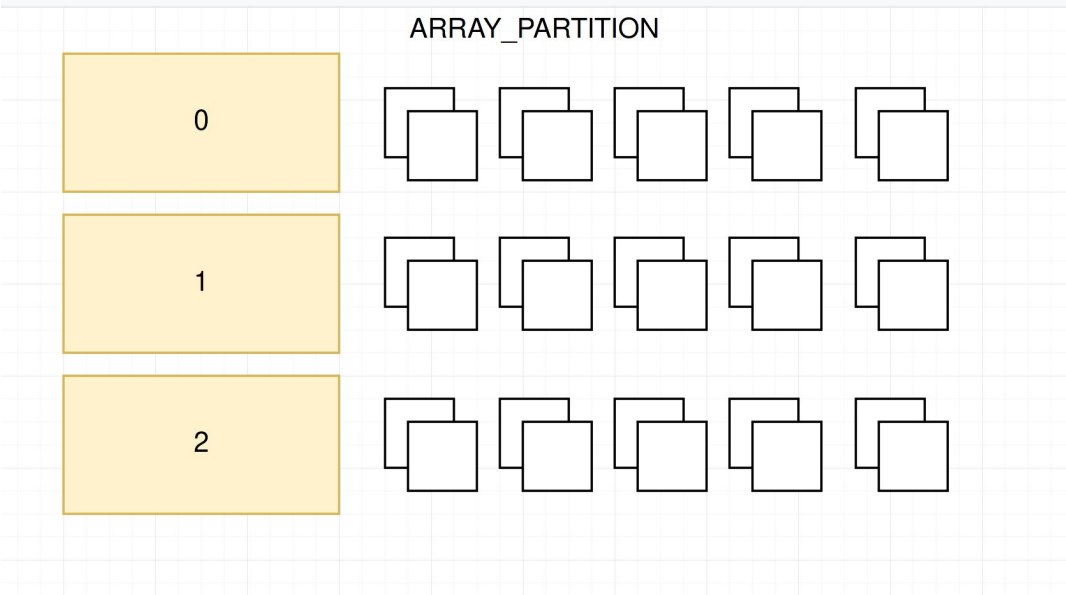
- ARRAY_RESHAPE for **single** dimensional array



ARRAY_PARITION vs ARRAY_RESHAPE

Two pragma are very similar, when to use what ?

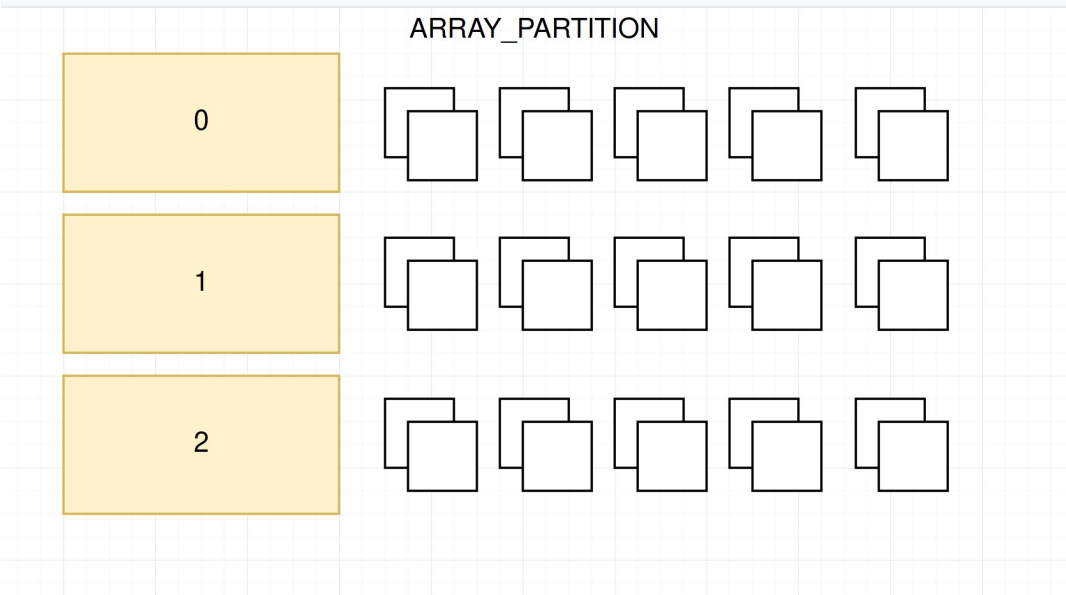
- ARRAY_RESHAPE for **single** dimensional array
- ARRAY_PARITION for **multi**-dimensional array



ARRAY_PARITION vs ARRAY_RESHAPE

Demo

- C simulation
- C synthesis



ARRAY_PARITION vs ARRAY_RESHAPE

Modules Loops

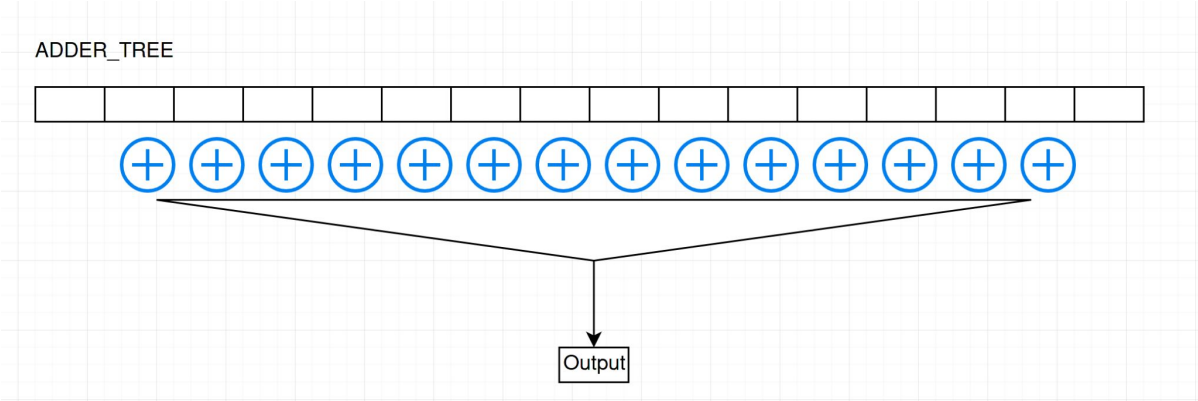
Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul		-	229	2.290E3	-	230	-	no	0	0	34	194	0
hls_vector_mul_part		-	25	250.000	-	25	-	no	0	0	24	181	0
calc		-	13	130.000	5	1	10	yes	-	-	-	-	-
adder_tree		-	5	50.000	3	1	4	yes	-	-	-	-	-
buffering_VITIS_LOOP_84_1		-	201	2.010E3	3	1	200	yes	-	-	-	-	-

HW Interfaces

Resolve all violations
Next problems: resources blow up

Adder Tree

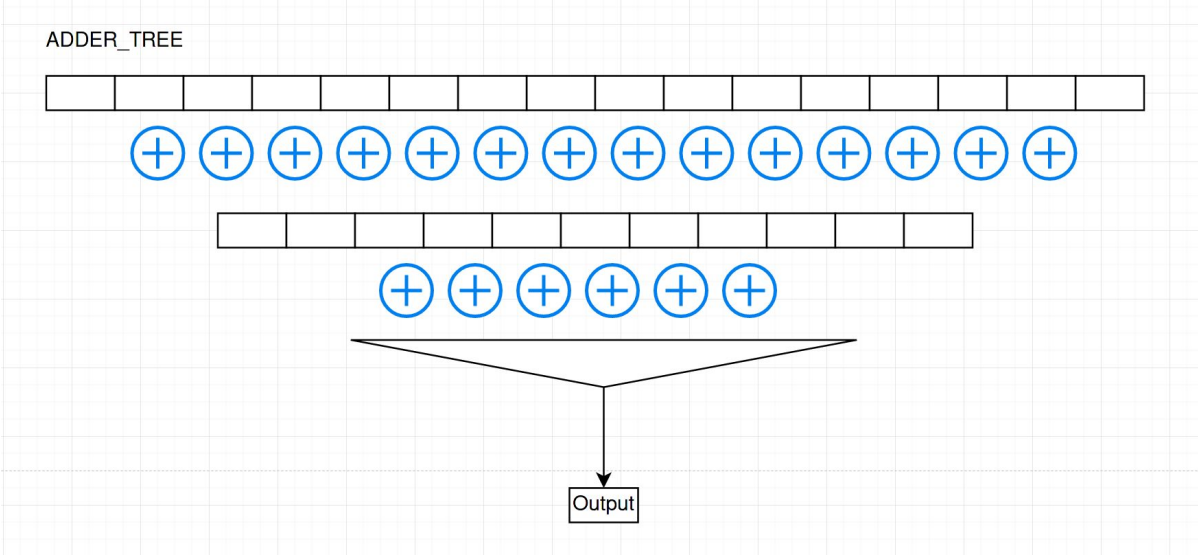
What we have



Adder Tree

What we want:

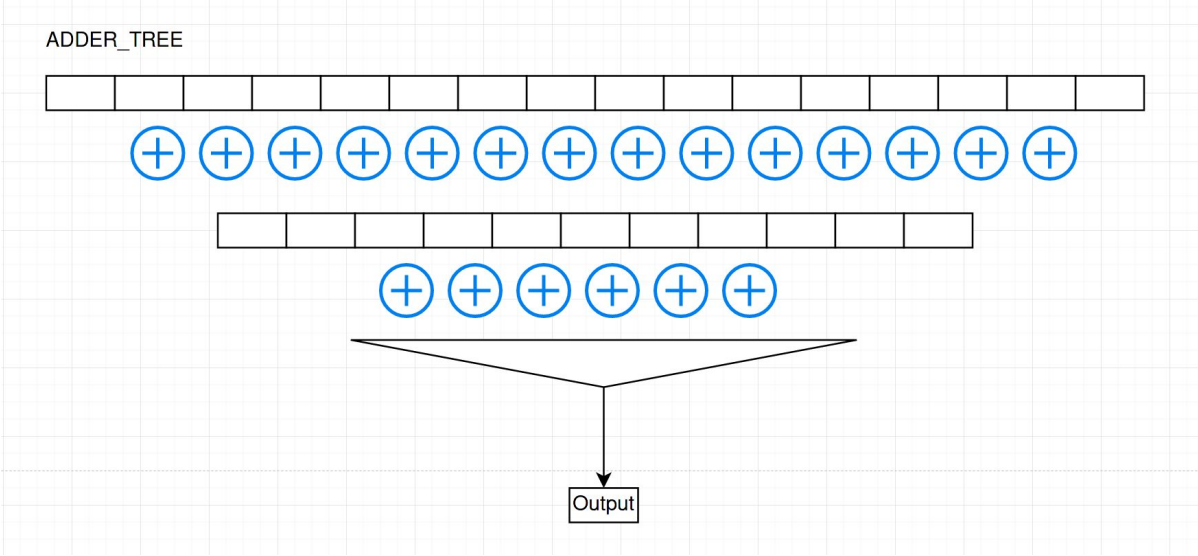
Multiple layer of adder tree



Adder Tree

Demo

- C simulation
- C synthesis



New Adder Tree

Again, we spent additional loop to initialize array

Guess what we need to do?

```
13
14      for (auto i = 0; i < 5; i++) middle[i] = 0;
15
16  adder_tree:
17      for (auto j = 0; j < BUFFER; j+=5)
18      {
19          for (auto i = 0; i < 5; i++)
20          {
21  #pragma HLS UNROLL
22              middle[i] += sum[j + i];
23          }
24      }
25
26  reduce:
27      for (auto i = 0; i < 5; i++)
28      {
29  #pragma HLS UNROLL
30          final_sum += middle[i];
31      }
```

New Adder Tree

Again, we spent additional loop to initialize array

Guess what we need to do?

- One.Big.Loop

```
    for (auto i = 0; i < 5; i++) middle[i] = 0;

adder_tree:
    for (auto j = 0; j < BUFFER; j+=5)
    {
        for (auto i = 0; i < 5; i++)
        {
#pragma HLS UNROLL
            middle[i] += sum[j + i];
        }
    }

reduce:
    for (auto i = 0; i < 5; i++)
    {
#pragma HLS UNROLL
        final_sum += middle[i];
    }
```

New Adder Tree

Again, we spent additional loop to initialize array

Guess what we need to do?

- One.Big.Loop
- Solve Violation

```
    for (auto i = 0; i < 5; i++) middle[i] = 0;

adder_tree:
    for (auto j = 0; j < BUFFER; j+=5)
    {
        for (auto i = 0; i < 5; i++)
        {
#pragma HLS UNROLL
            middle[i] += sum[j + i];
        }
    }

reduce:
    for (auto i = 0; i < 5; i++)
    {
#pragma HLS UNROLL
        final_sum += middle[i];
    }
```

New Adder Tree

Again, we spent additional loop to initialize array

Guess what we need to do?

- One.Big.Loop
- Solve Violation

Demo

- C simulation
- C synthesis

```
    for (auto i = 0; i < 5; i++) middle[i] = 0;

adder_tree:
    for (auto j = 0; j < BUFFER; j+=5)
    {
        for (auto i = 0; i < 5; i++)
        {
#pragma HLS UNROLL
            middle[i] += sum[j + i];
        }
    }

reduce:
    for (auto i = 0; i < 5; i++)
    {
#pragma HLS UNROLL
        final_sum += middle[i];
    }
```

Resources Blow up

Modules Loops

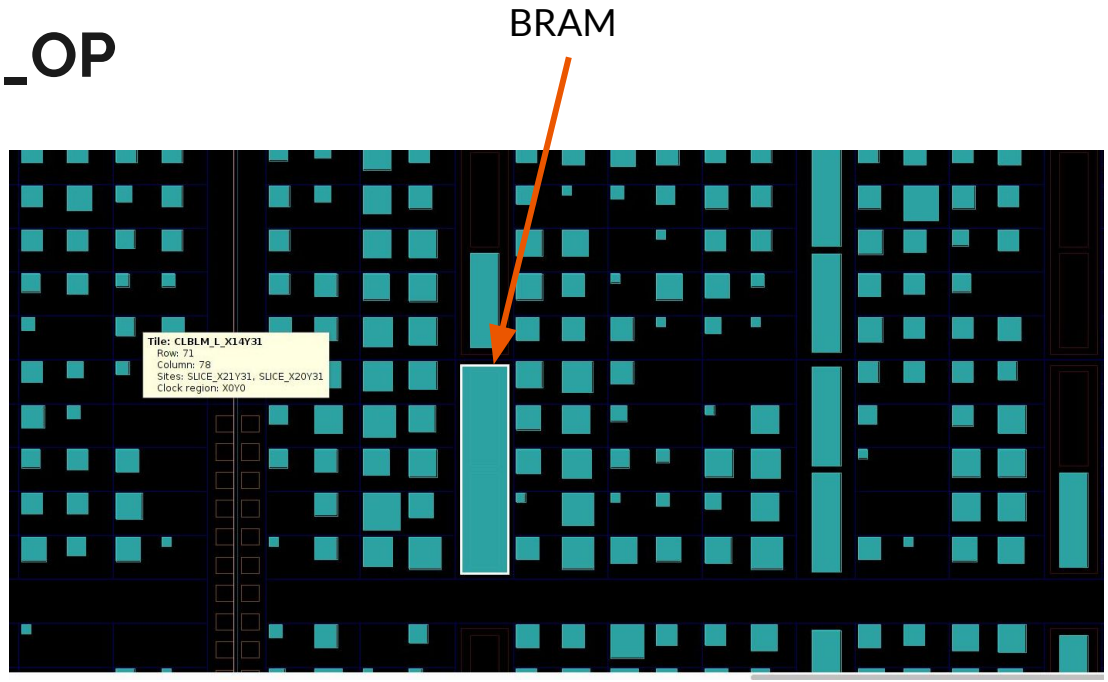
Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul		-	229	2.290E3	-	230	-	no	0	0	35	195	0
hls_vector_mul_part		-	25	250.000	-	25	-	no	0	0	25	182	0
buffering_VITIS_LOOP_86_1		-	201	2.010E3	3	1	200	yes	-	-	-	-	-

HW Interfaces

LUT utilization is above 100%
No DSP usage?

#pragma HLS BIND_OP

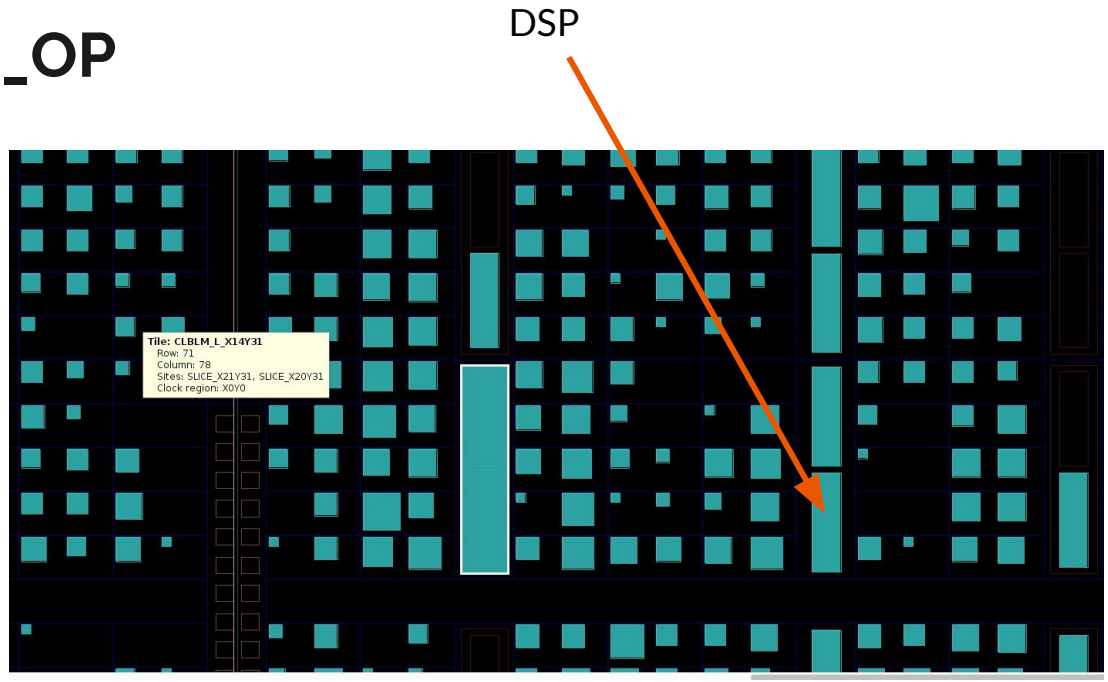
Pin specified operation to premade fabric: DSP, BRAM, URAM, etc…



#pragma HLS BIND_OP

Pin specified operation to premade fabric: DSP, BRAM, URAM, etc…

Force Pipeline inside fabric with specified latency (auto is good enough)



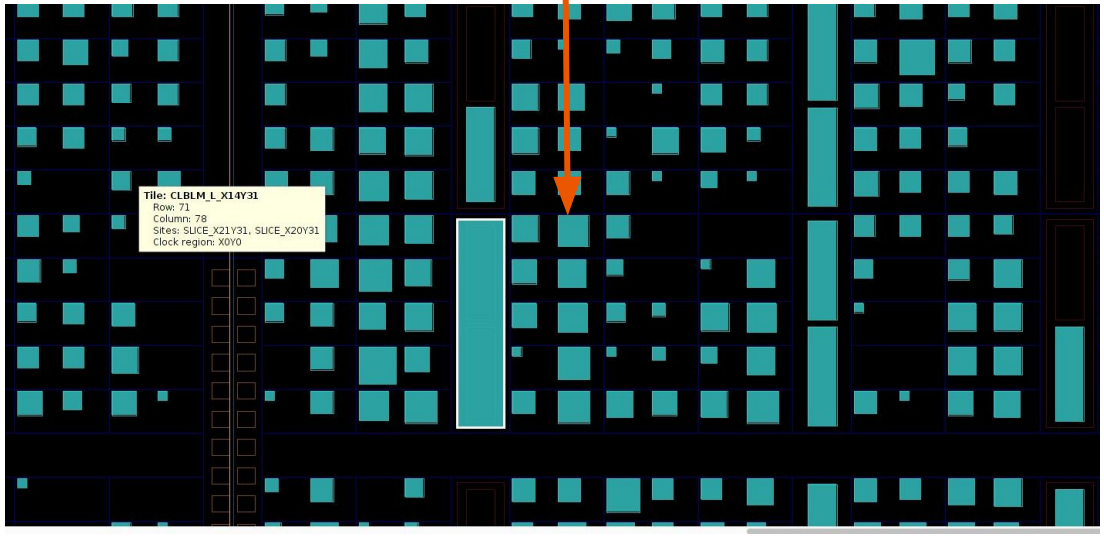
#pragma HLS BIND_OP

Pin specified operation to premade fabric: DSP, BRAM, URAM, etc…

Force Pipeline inside fabric with specified latency (auto is good enough)

Boost maximum frequency

CLB ? Who cares



#pragma HLS BIND_OP

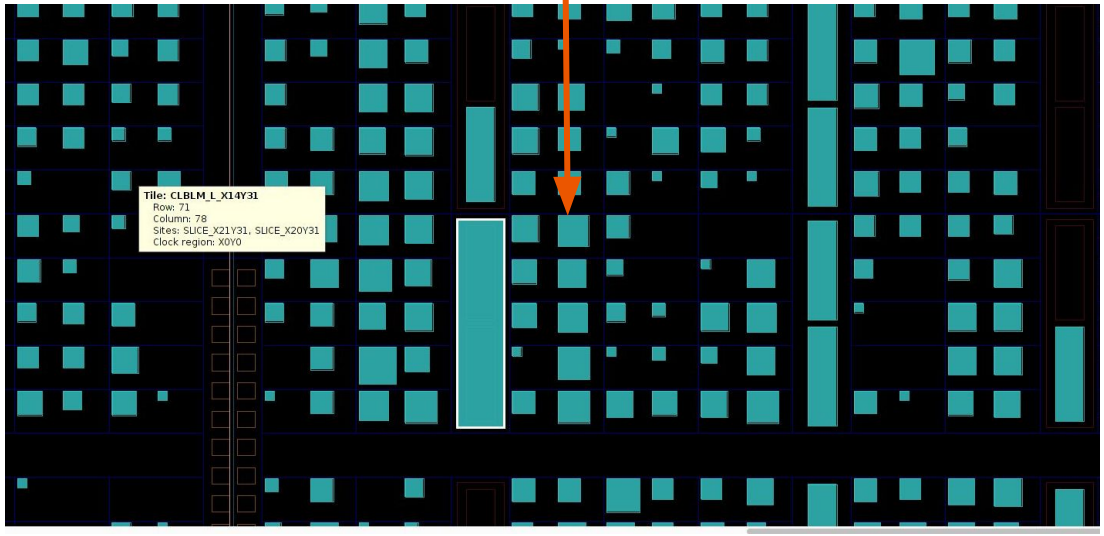
Pin specified operation to premade fabric: DSP, BRAM, URAM, etc…

Force Pipeline inside fabric with specified latency (auto is good enough)

Boost maximum frequency

Demo
- C synthesis

CLB ? Who cares



#pragma HLS BIND_OP

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul		-	237	2.370E3	-	238	-	no	0	0	18	112	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	13	104	0
calc		-	23	230.000	5	1	20	yes	-	-	-	-	-
adder_tree		-	3	30.000	3	1	2	yes	-	-	-	-	-
buffering_VITIS_LOOP_89_1		-	201	2.010E3	3	1	200	yes	-	-	-	-	-

HW Interfaces

%LUT stay the same ???

#pragma HLS BIND_OP

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul		-	237	2.370E3	-	238	-	no	0	0	18	112	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	13	104	0
calc		-	23	230.000	5	1	20	yes	-	-	-	-	-
adder_tree		-	3	30.000	3	1	2	yes	-	-	-	-	-
buffering_VITIS_LOOP_89_1		-	201	2.010E3	3	1	200	yes	-	-	-	-	-

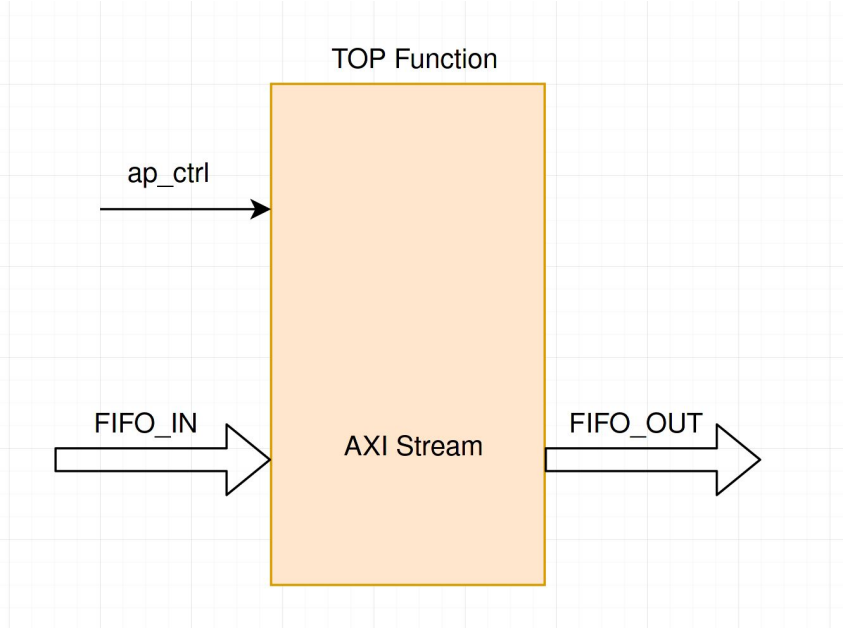
HW Interfaces

%LUT stay the same ???

Don’t worry, it will synthesize to DSP. We want to utilize as many as DSPs as we can.

Block Level I/O

It’s time to write top function

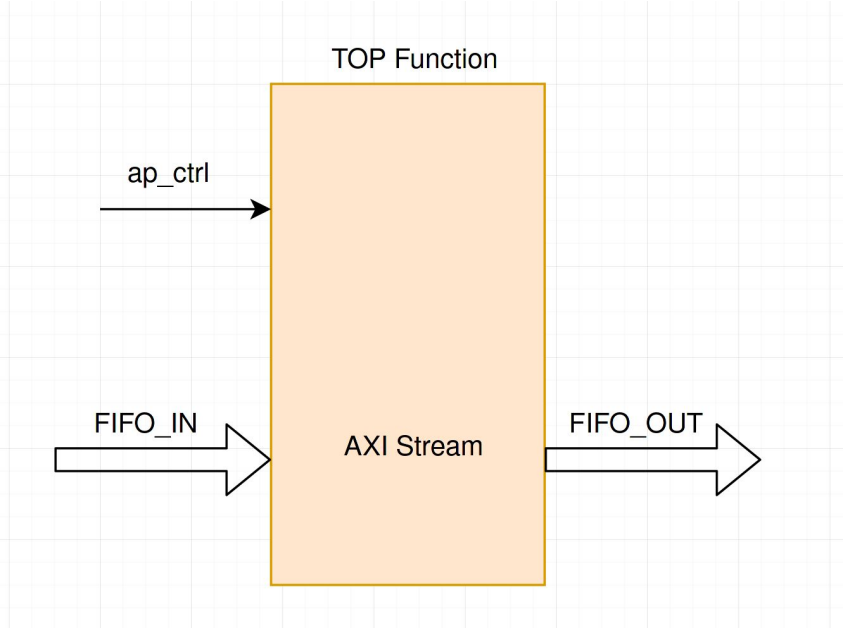


Block Level I/O

It's time to write top function

Demo

- Write AXIS TOP
- Write testbench
- C Simulation
- Synthesis



Block Level I/O

With simple #pragma INTERFACE, HLS can easily infer Block Level protocol

S_AXILITE

Interface	Data Width	Address Width
s_axi_control	32	4

AXIS

Interface	Register Mode	TDATA	TKEEP	TLAST	TREADY	TSTRB	TVALID
fifo_in	both	32	4	1	1	4	1
fifo_out	both	32	4	1	1	4	1

TOP LEVEL CONTROL

Interface	Type	Ports
ap_clk	clock	ap_clk
ap_rst_n	reset	ap_rst_n
interrupt	interrupt	interrupt
ap_ctrl	ap_ctrl_hs	

SW I/O Information

Top Function Arguments

Argument	Direction	Datatype
fifo_in	in	stream<hls::axis<ap_uint<32> 0 0 0> 0>&
fifo_out	out	stream<hls::axis<ap_uint<32> 0 0 0> 0>&

SW-to-HW Mapping

Argument	HW Name	HW Type
fifo_in	fifo_in	interface
fifo_out	fifo_out	interface

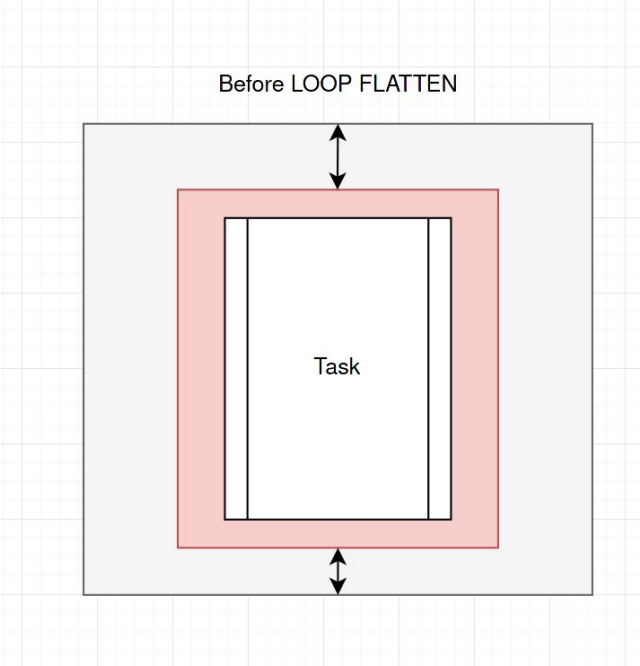
#pragma HLS LOOP_FLATTEN

Either:

- In every iteration of the **outer loop**, it spends 1 cycles to jump into **inner loop**

Or

- After **inner loop** finished, it spends 1 cycles to jump to **outer loop**



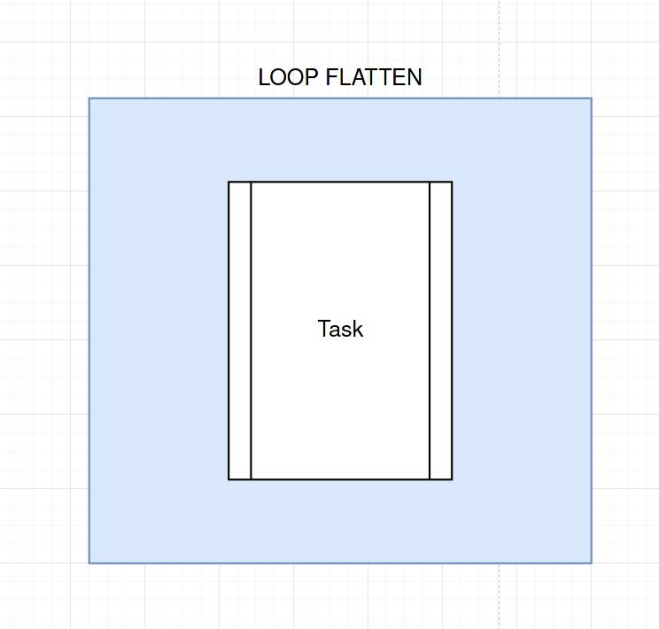
#pragma HLS LOOP_FLATTEN

Either:

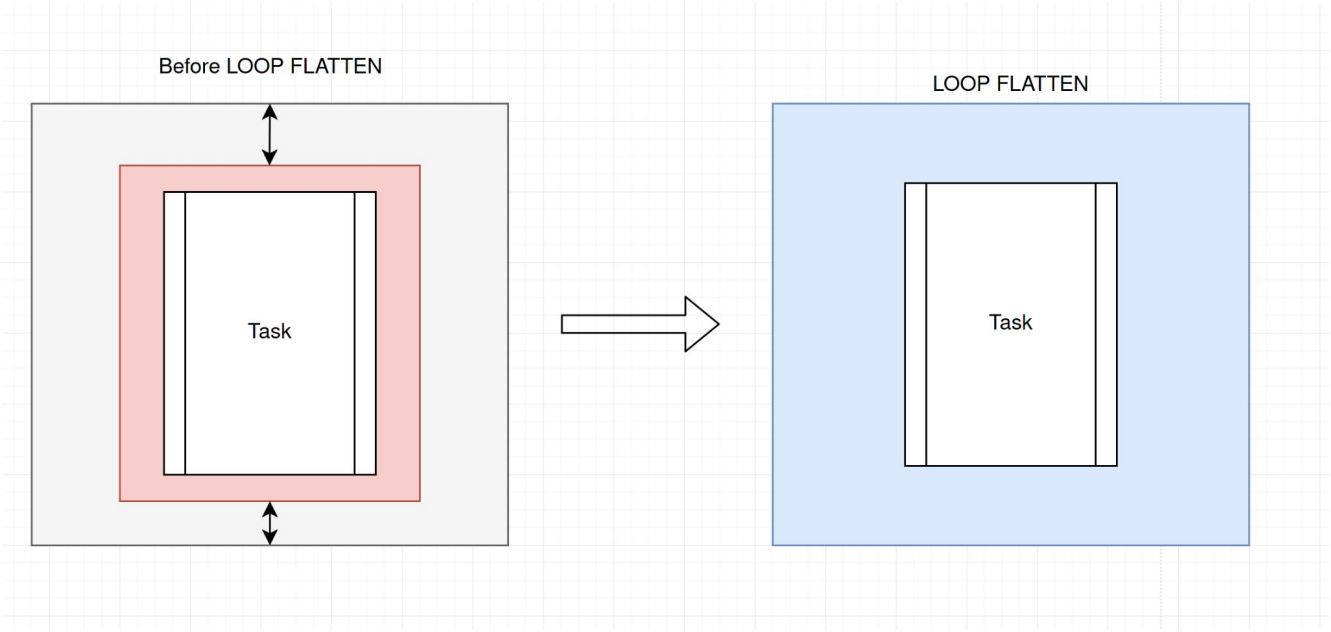
- In every iteration of the **outer loop**, it spends 1 cycles to jump into **inner loop**

Or

- After **inner loop** finished, it spends 1 cycles to jump to **outer loop**



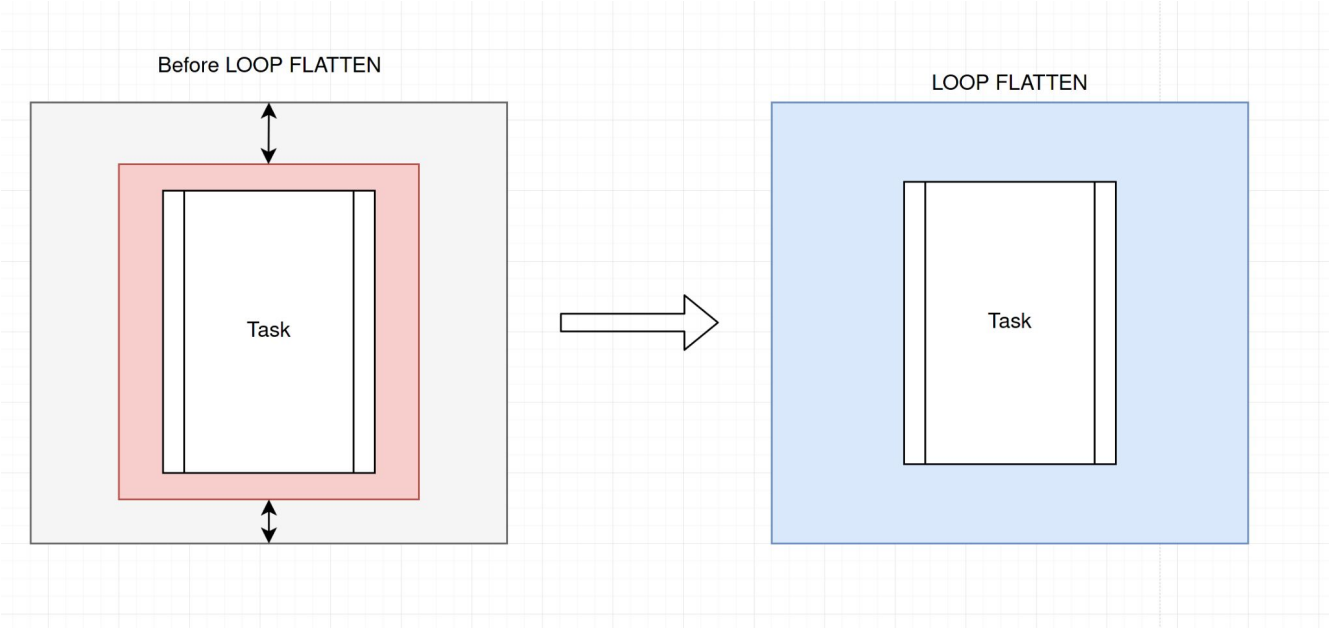
#pragma HLS LOOP_FLATTEN



#pragma HLS LOOP_FLATTEN

Demo:

- C Synthesis



#pragma HLS PIPELINE

Demo:

- C Synthesis

#pragma HLS PIPELINE

Demo:

- C Synthesis

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	20	114	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	13	104	0
buffering_VITIS_LOOP_104_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

#pragma HLS PIPELINE

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	20	114	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	13	104	0
buffering_VITIS_LOOP_104_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

%LUT > 100%.
It’s time for final optimization

Bit Precision

Apply bit precision instead of **u32** to reduce resources

```
54  calc:
55      for (auto i = 0; i < N; i+=BUFFER)
56      {
57  #pragma HLS PIPELINE II=1
58          for (auto j = 0; j < BUFFER; j++)
59          {
60  #pragma HLS UNROLL
61  #pragma BIND_OP variable=a op=mul impl=dsp
62  #pragma BIND_OP variable=b op=mul impl=dsp
63  #pragma BIND_OP variable=c op=add impl=dsp
64              auto prev = (i == 0) ? static_cast<u32>(0) : sum[j];
65              sum[j] = prev + (c[i/BUFFER][j] + a[i/BUFFER][j] * b[i/BUFFER][j]) & mask;
66          }
67      }
```

Bit Precision

Apply bit precision instead of **u32** to reduce resources

Remove all mask operations

```
49  calc:
50      for (auto i = 0; i < N; i+=BUFFER)
51      {
52  #pragma HLS PIPELINE II=1
53          for (auto j = 0; j < BUFFER; j++)
54          {
55  #pragma HLS UNROLL
56  #pragma BIND_OP variable=a op=mul impl=dsp
57  #pragma BIND_OP variable=b op=mul impl=dsp
58  #pragma BIND_OP variable=c op=add impl=dsp
59              auto prev = (i == 0) ? static_cast<u29>(0) : sum[j];
60              sum[j] = prev + (c[i/BUFFER][j] + a[i/BUFFER][j] * b[i/BUFFER][j]);
61          }
62      }
```

Bit Precision

```
u32 hls_vector_mul_part(const u32 a[N/BUFFER][BUFFER],
                        const u32 b[N/BUFFER][BUFFER],
                        const u32 c[N/BUFFER][BUFFER])
{
    const u32 mask = 0x1fffffff;

    u32 final_sum;
    u32 sum[BUFFER];
#pragma HLS ARRAY_RESHAPE variable=sum complete dim=1
```

Apply bit precision instead of **u32** to reduce resources

Remove all mask operations

Use exact bit

```
/* Bit precision
 * Remove masking, after synthesis, we see huge resources usage drop
 */
u32 hls_vector_mul_part(const u29 a[N/BUFFER][BUFFER],
                        const u29 b[N/BUFFER][BUFFER],
                        const u29 c[N/BUFFER][BUFFER])
{
    u32 final_sum;
    u29 sum[BUFFER];
#pragma HLS ARRAY_RESHAPE variable=sum complete dim=1
```

Bit Precision

```
u32 hls_vector_mul_part(const u32 a[N/BUFFER][BUFFER],
                        const u32 b[N/BUFFER][BUFFER],
                        const u32 c[N/BUFFER][BUFFER])
{
    const u32 mask = 0x1fffffff;

    u32 final_sum;
    u32 sum[BUFFER];
#pragma HLS ARRAY_RESHAPE variable=sum complete dim=1
```

Apply bit precision instead of **u32** to reduce resources

Remove all mask operations

Use exact bit

Demo

- C simulation
- Synthesis

```
/* Bit precision
 * Remove masking, after synthesis, we see huge resources usage drop
 */
u32 hls_vector_mul_part(const u29 a[N/BUFFER][BUFFER],
                        const u29 b[N/BUFFER][BUFFER],
                        const u29 c[N/BUFFER][BUFFER])
{
    u32 final_sum;
    u29 sum[BUFFER];
#pragma HLS ARRAY_RESHAPE variable=sum complete dim=1
```

Bit Precision

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	16	81	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	9	71	0
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

Huge resources drop

Bit Precision

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	16	81	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	9	71	0
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

Huge resources drop

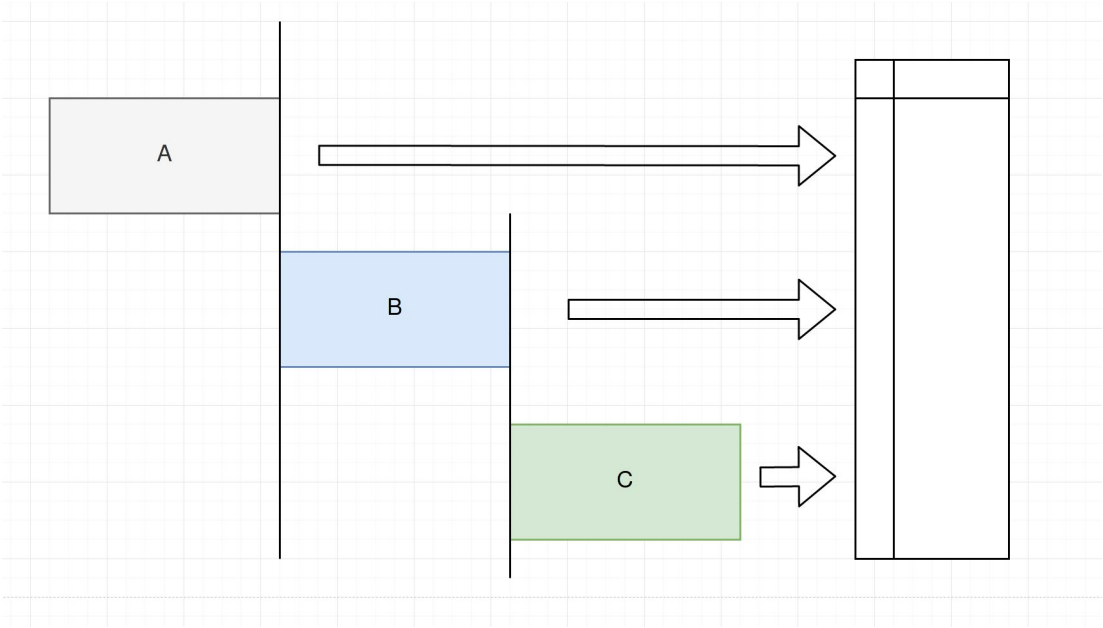
From 200 cycles at baseline, we reduce to 33 cycles.
Speed up: **6.06x**

Ideal situation

We spend huge resources to reduce latency from **200** cycles to **33** cycles.

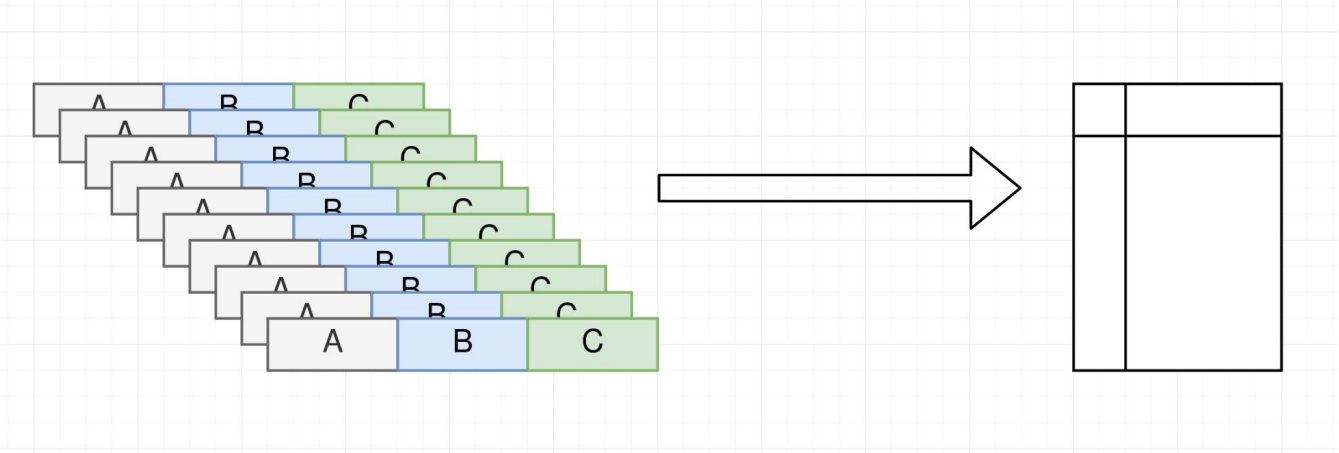
What is the ideal cycles we can achieve?

Recall Input Flow



Perfect input flow

A,B,C interleave



HLS Streaming Computation

Demo

- Update test bench
- Update code
- C Simulation
- Synthesis
- Co-simulation Verilog
- **Co-simulation VHDL ←- Note**

HLS Streaming Computation

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	604	6.040E3	-	605	-	no	0	0	1	1	0
VITIS_LOOP_43_1		-	600	6.000E3	4	3	200	yes	-	-	-	-	-

HW Interfaces

Ideal latency: 4 cycles
Ideal resources: 1%

Compare two styles of input flow

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	16	81	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	9	71	0
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM(%)	DSP(%)	FF(%)	LUT(%)	URAM(%)
hls_vector_mul_top		-	604	6.040E3	-	605	-	no	0	0	1	1	0
VITIS_LOOP_43_1		-	600	6.000E3	4	3	200	yes	-	-	-	-	-

HW Interfaces

Compare two styles of input flow

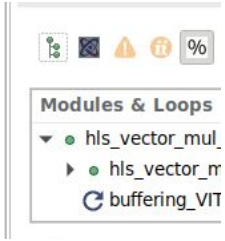
% Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	5796	14389	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	3469	12604	0
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

Resources utilization after Synthesis in Vivado

Co




```
Implementation tool: Xilinx Vivado v.2020.2
Project:             Introduction_HLS
Solution:            92_bitprecision
Device target:       xc7z010-clg400-1
Report date:         Tue Apr 13 20:43:24 UTC 2021


#=== Post-Synthesis Resource usage ===
SLICE:            0
LUT:           5549
FF:            2146
DSP:             36
BRAM:             0
SRL:              0
#=== Final timing ===
CP required:    10.000
CP achieved post-synthesis:    7.722
Timing met
```

.UT(%)	URAM(%)
81	0
71	0
-	-

Compare two styles of input flow

% Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	5796	14389	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	3469	12604	0
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

Resources utilization after Synthesis in Vivado

```
Project:            Introduction_HLS
Solution:           92_bitprecision
Device target:      xc7z010-clg400-1
Report date:        Tue Apr 13 20:43:24 UTC 2021


#=== Post-Synthesis Resource usage ===
SLICE:            0
LUT:           5549
FF:            2146
DSP:             36
BRAM:             0
SRL:              0
#=== Final timing ===
CP required:    10.000
CP achieved post-synthesis:    7.722
Timing met
```

Go back, increase BUFFER, FASTERRRRRRRRRRR!!!!

Compare two styles of input flow

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	5796	14389	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	3469	12604	0
calc		-	23	230.000	5	1	20	yes	-	-	-	-	-
adder_tree		-	3	30.000	3	1	2	yes	-	-	-	-	-
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

With 2x more resources, we reduce 23 down to 13

Compare two styles of input flow

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
hls_vector_mul_top		-	640	6.400E3	-	641	-	no	0	3	5796	14389	0
hls_vector_mul_part		-	33	330.000	-	33	-	no	0	0	3469	12604	0
calc		-	23	230.000	5	1	20	yes	-	-	-	-	-
adder_tree		-	3	30.000	3	1	2	yes	-	-	-	-	-
buffering_VITIS_LOOP_99_1		-	603	6.030E3	5	1	600	yes	-	-	-	-	-

HW Interfaces

With 2x more resources, we reduce 23 down to 13

Compare with ideal latency: 13/4 = 3.25x
Not bad. heh?

Coding Conclusion

- If you are looking at lightweight implementation: choose HDL
- HLS is an abstract layer over HDL
- HLS can perform complex operation as HDL

- Protocol with HLS is easy
- Testing with HLS is quick
- Co-simulation with HLS is quick

Performance Conclusion

- HLS can achieve good performance.
- CERG GMU demonstrates HLS can achieve similar latency, frequency as RTL.
  But HLS utilizes more resources compare to RTL.
- The problem with FPGA in HPC nowadays is not to minimize LUT, FF usage.
  It is to achieve **maximum performance**, utilize all available resources on FPGA.
- Remember to take the transfer size into account.

Performance Conclusion

- HLS can achieve good performance.
- CERG GMU demonstrates HLS can achieve similar latency, frequency as RTL. HLS utilize more resources compare to RTL.
- The problem with FPGA in HPC nowadays is not to minimize LUT, FF usage. It is to achieve maximum performance, utilize all available resources on FPGA.
- Remember to take the transfer size into account.

- **Prepare input data can give ultimate solution!!!**

What the point of Hardware accelerator anyway?

- Not mentioned in this talk: **#pragma HLS DATAFLOW**

- Not mentioned in this talk: NEON vs HLS
- Load/Store architecture and Hardware Accelerator

- The code for NEON is ready. It's up to you to figure it out which one is faster in this case. And let the class know!!!

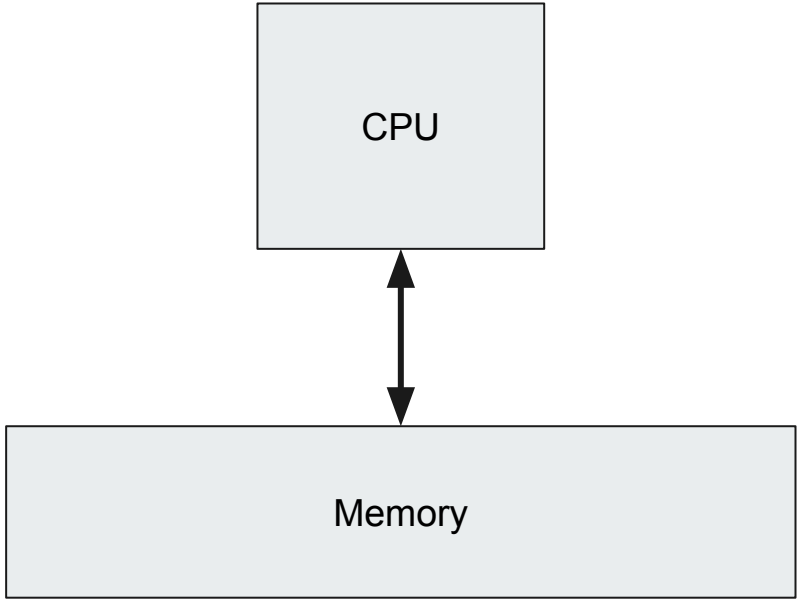
NEON Implementation

```
    0x100003c7c    03e4006f    movi v3.2d, 0000000000000000
┌─> 0x100003c80    2429df4c    ld1 {v4.4s, v5.4s, v6.4s, v7.4s}, [x9], 0x40
¦   0x100003c84    5029df4c    ld1 {v16.4s, v17.4s, v18.4s, v19.4s}, [x10], 0x40
¦   0x100003c88    7429df4c    ld1 {v20.4s, v21.4s, v22.4s, v23.4s}, [x11], 0x40
¦   0x100003c8c    1496a44e    mla v20.4s, v16.4s, v4.4s
¦   0x100003c90    3596a54e    mla v21.4s, v17.4s, v5.4s
¦   0x100003c94    5696a64e    mla v22.4s, v18.4s, v6.4s
¦   0x100003c98    7796a74e    mla v23.4s, v19.4s, v7.4s
¦   0x100003c9c    1474076f    bic v20.4s, 0xe0, lsl 24
¦   0x100003ca0    1574076f    bic v21.4s, 0xe0, lsl 24
¦   0x100003ca4    1674076f    bic v22.4s, 0xe0, lsl 24
¦   0x100003ca8    1774076f    bic v23.4s, 0xe0, lsl 24
¦   0x100003cac    8386a34e    add v3.4s, v20.4s, v3.4s
¦   0x100003cb0    a286a24e    add v2.4s, v21.4s, v2.4s
¦   0x100003cb4    c186a14e    add v1.4s, v22.4s, v1.4s
¦   0x100003cb8    e086a04e    add v0.4s, v23.4s, v0.4s
¦   0x100003cbc    08410091    add x8, x8, 0x10
¦   0x100003cc0    1fc102f1    cmp x8, 0xb0
└─< 0x100003cc4    e3fdff54    b.lo 0x100003c80
```

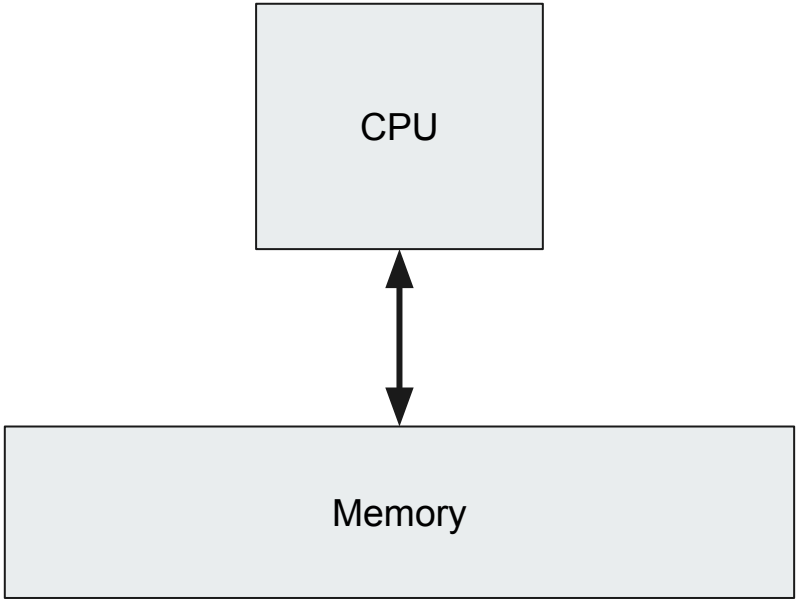
NEON Implementation

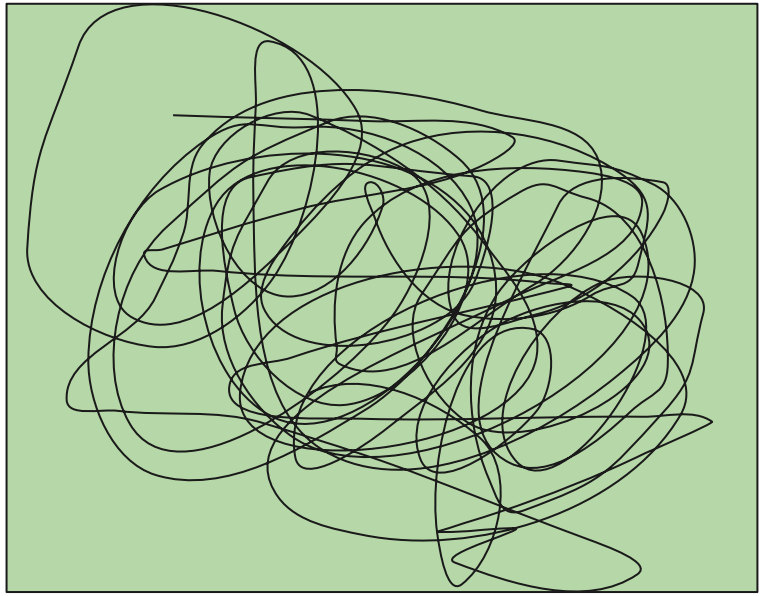
```
45
46      for (int i = 0; i < N - 8; i += 16)
47      {
48          vload(neon_a, &a[i]);
49          vload(neon_b, &b[i]);
50          vload(neon_c, &c[i]);
51
52          // tmp = c + a*b
53          vmla(neon_tmp.val[0], neon_c.val[0], neon_a.val[0], neon_b.val[0]);
54          vmla(neon_tmp.val[1], neon_c.val[1], neon_a.val[1], neon_b.val[1]);
55          vmla(neon_tmp.val[2], neon_c.val[2], neon_a.val[2], neon_b.val[2]);
56          vmla(neon_tmp.val[3], neon_c.val[3], neon_a.val[3], neon_b.val[3]);
57
58          // tmp &= mask
59          vand(neon_tmp.val[0], neon_tmp.val[0], neon_mask);
60          vand(neon_tmp.val[1], neon_tmp.val[1], neon_mask);
61          vand(neon_tmp.val[2], neon_tmp.val[2], neon_mask);
62          vand(neon_tmp.val[3], neon_tmp.val[3], neon_mask);
63
64          // e += tmp
65          vadd(neon_e.val[0], neon_e.val[0], neon_tmp.val[0]);
66          vadd(neon_e.val[1], neon_e.val[1], neon_tmp.val[1]);
67          vadd(neon_e.val[2], neon_e.val[2], neon_tmp.val[2]);
68          vadd(neon_e.val[3], neon_e.val[3], neon_tmp.val[3]);
69      }
70      // e += tmp
```

Load/Store Architecture vs Hardware Accelerator



Load/Store Architecture vs Hardware Accelerator



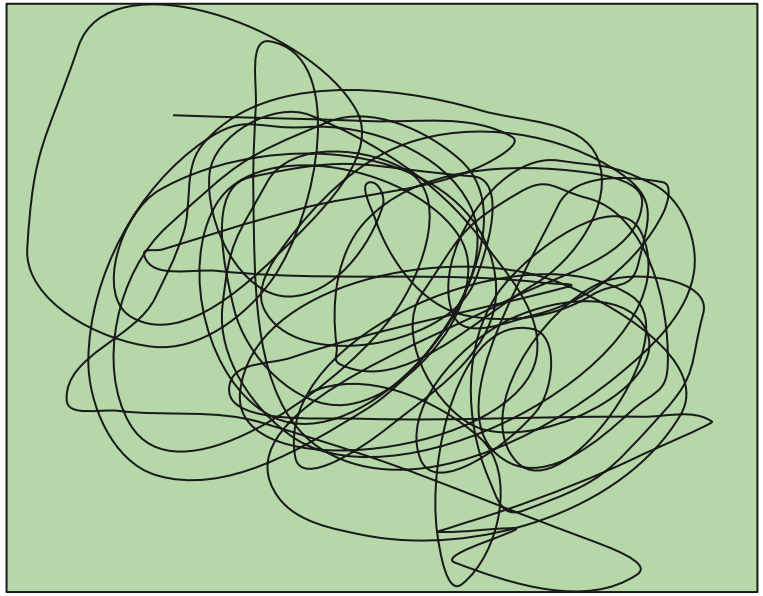


Load/Store Architecture vs Hardware Accelerator

- No pressure on CPU
- No Store and Load

Real world example:

- Video decoder: There are plenty optimized SW implementations. If you don't enable HW, it's give low framerate and high CPU usage.



Thank you

Question ???

Reference

Code used in this talk:

- https://github.com/cothan/Vitis_High_Level_Synthesis_Training

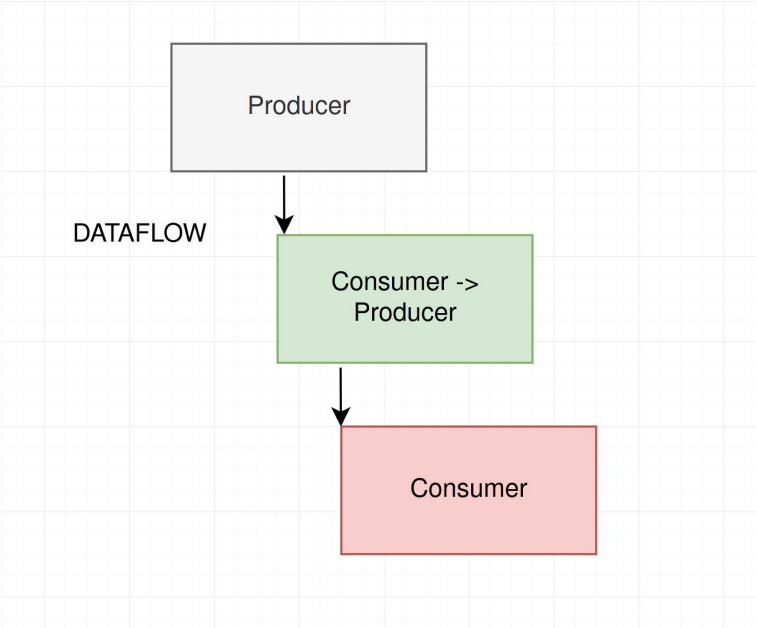
Helpful documents which are not from Xilinx:

- Transformations of High-Level Synthesis Codes for High-Performance Computing

  https://arxiv.org/abs/1805.08288

#pragma HLS DATAFLOW

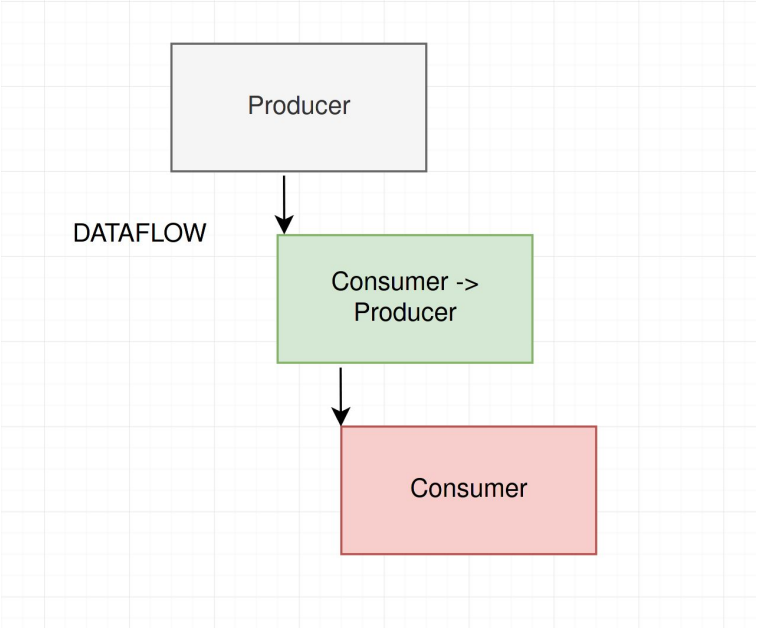
DATAFLOW need to be written in producer-consumer fashion



#pragma HLS DATAFLOW

DATAFLOW need to be written in producer-consumer fashion

May increase or decrease the resources utilization.



#pragma HLS DATAFLOW

% Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
hls_vector_mul_top		-	610	6.100E3	-	611	-	no	0	0	3421	2436	0
VITIS_LOOP_117_1		-	600	6.000E3	31	30	20	yes	-	-	-	-	-
adder_tree		-	2	20.000	1	1	2	yes	-	-	-	-	-

HW Interfaces

Ideal input flow, A,B,C interleaving.