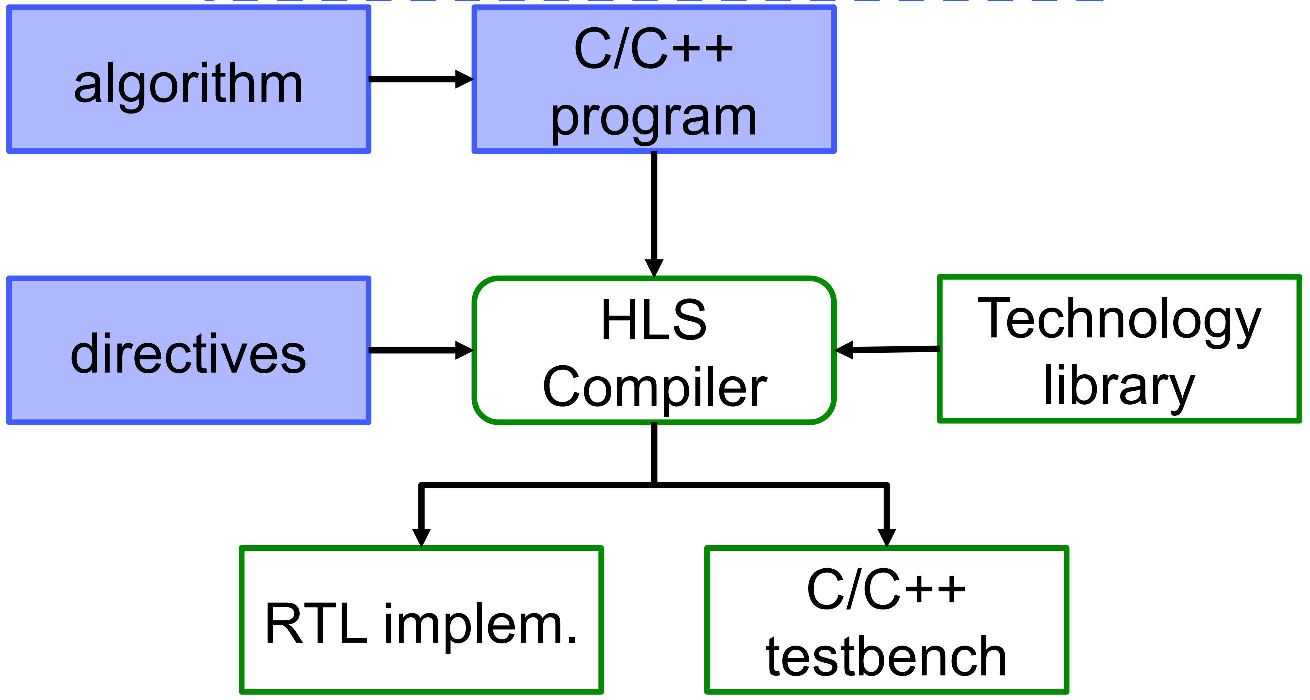
High Level Synthesis

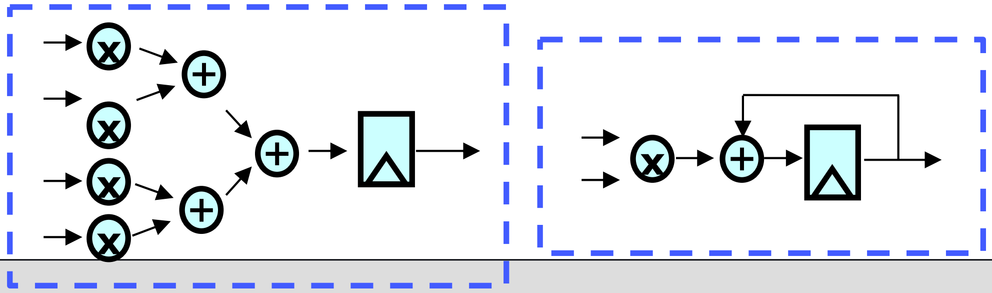
How to use it!

HLS : from algorithm to Silicon

```
for(i=0;i<N;i++)
    acc+=data[i]*coef[i];
```

$$\sum_{i=0}^{N-1} A_i . X_i$$





High Level Synthesis aims at raising the level of abstraction of hardware design

HLS : does it really work ?

- High Level Synthesis is not a new idea
  - Seminal academic research work starting in the early 1990’s
  - First commercial tools on the market around 1995
- First generation of HLS was a commercial failure
  - Worked only on small toy examples
  - QoR (area/power) way below expectations
- It took 15 years to make these tools actually work
  - Now used by worldclass chip vendors (Apple, Samsung, Nxt, …)
  - Many designers reject HLS (bad past experience or prejudice)

HLS Customers Stories

- Major industry players use HLS solutions. Qualcomm, Google, Nvidia praised Catapult-C for:
- C/C++ language
  - Single, familiar language
  - Bit exact results between model and RTL
- High level design
  - Team working on high-value problems
  - DSE : try high number of algo/archi
- Verification
  - Simpler testbench, much faster verification
  - 99% of functional bugs found in C++ before RTL simulation
  - Google video encoder verification: 7-8 orders of magnitude faster

F. Sijstermans and J. Li, "Working Smarter, Not Harder: NVIDIA Closes Design Complexity Gap With High-Level Synthesis", 2015.

HLS Customers Stories

- Fast arch. exploration, specification change, design reuse
  - Nvidia changed its decoder from 8 to 10 bits in less than 2 months,
  - Used design on mobile (510 MHz, 20nm) and desktop (800MHz, 28 HP)
  - Performance matching RTL custom design

Design	Display module 1		Display module 2		Camera module 1		Camera module 2	
	RTL	HLS	RTL	HLS	RTL	HLS	RTL	HLS
Area	3434	2876	8796	10960	2762	2838	49390	50247
Timing	0	0	-0,36	-0,33	0	0	0	0
Perf	3 pixels/3cycles		3 pixels/3cycles		2 pixels/cycles		2 pixels/cycles	
Latency	3 cycles		3 cycles		unconstrained		unconstrained	

F. Sijstermans and J. Li, "Working Smarter, Not Harder: NVIDIA Closes Design Complexity Gap With High-Level Synthesis", 2015.

Describing digital designs in C/C++

Is C/C++ suited for hardware design ?

- No, it follows a sequential execution model
  - Deriving efficient hardware involve parallel execution
- No, it has flat/linear vision of memory
  - All addresses point to a same space (virtual memory)
  - In an FPGA/ASIC memory is partitioned (memory blocks)
- No, it supports to few datatypes (char, int, float)
  - Hardware IP use customized wordlength (e.g 12 bits)

How to describe circuit with C, then ?

- Use classes to model **circuits** component and structure
  - Instantiate components and explicitly wire them together
  - Easy to generate a low level representation of the circuit
  - Use of template can ease the design of complex circuits

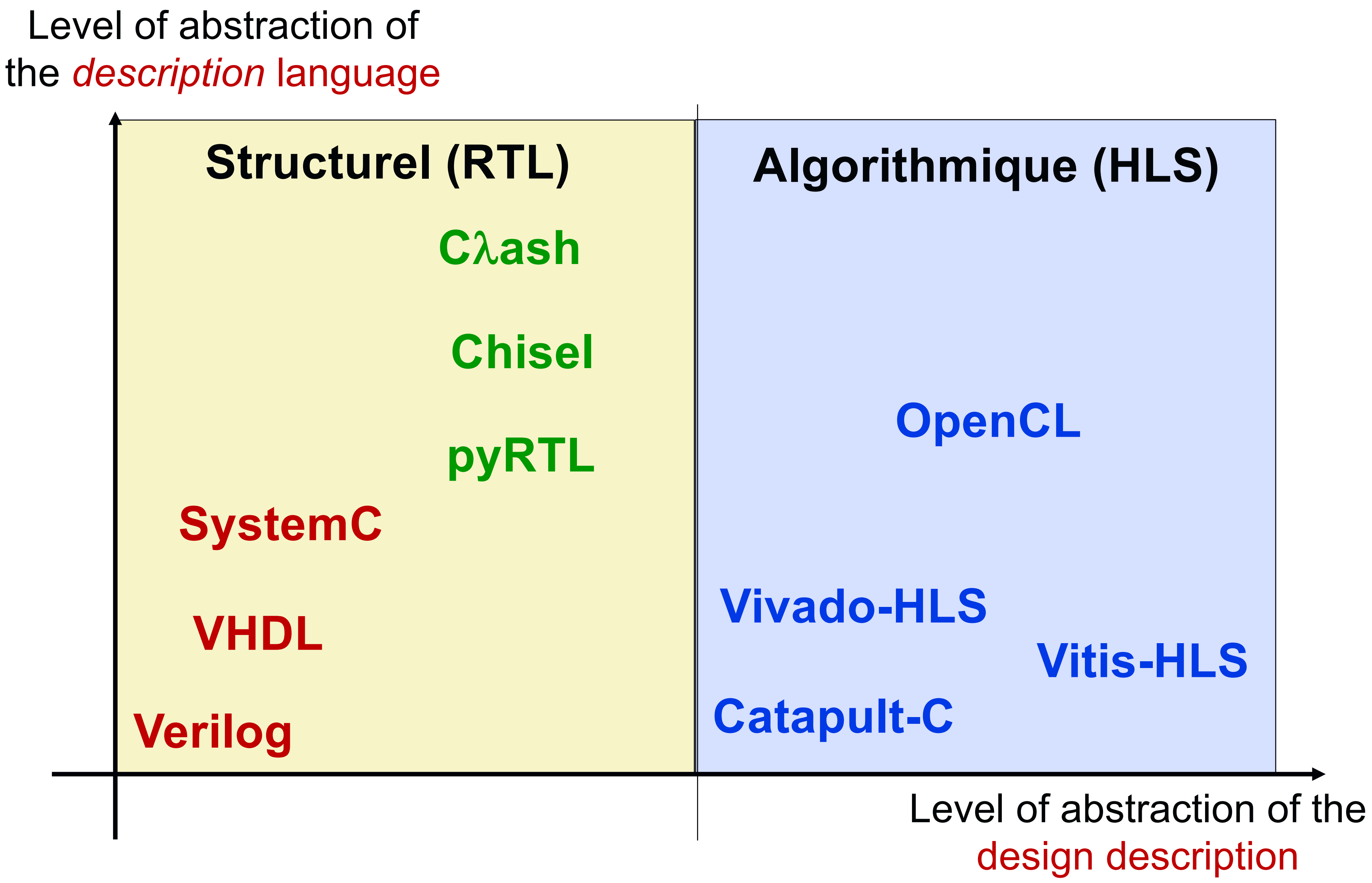
⇒ Hardware Construction Languages
  - SystemVerilog, SystemC, Chisel

Still operates at the Register to Logic Level

Does marginally raise the level of **design** abstraction

- C based HLS tools aims at being used as C compilers
  - User writes an **algorithm** in C, the tool produces a circuit.

HDL vs HCL vs HLS



The right use of HLS tools

- To use HLS, one **still** need to « *think in hardware* »
  - The designer defines the system macro-architecture
  - The HLS tools then derives the micro-architectures
  - Not full C (no printf, no malloc, no recursion, etc.)

Designers must fully understand how C/C++ language constructs map to hardware

- Need to be aware of optimizing compilers limitations
  - How smart a compiler dependency analysis can be ?
  - How to bypass compiler limitations (e.g using #pragmas)

Key things to understand

1. How the HLS tool *infers* the component interface (I/Os)
   - Key issue for integrating the component into other designs

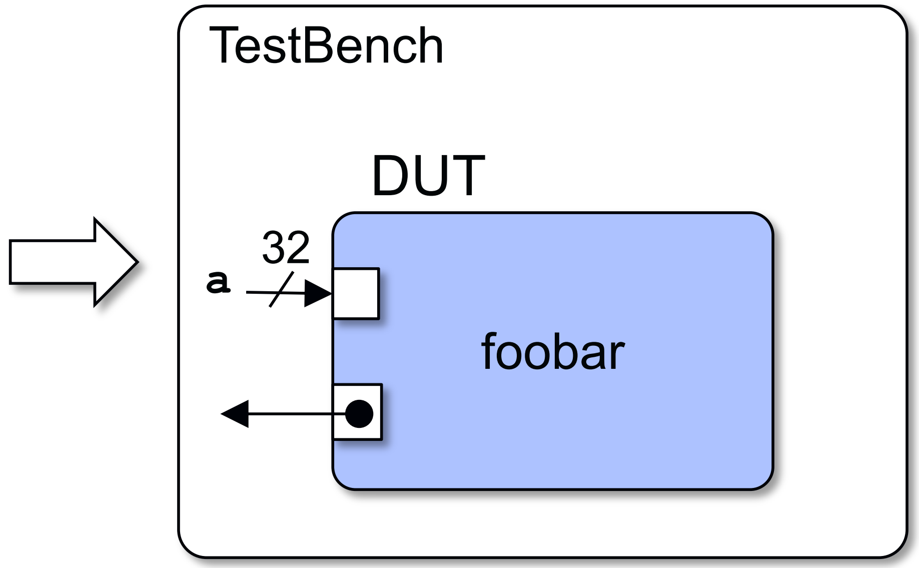
2. How the HLS tool handles data types and memory
   - Key issue when dealing with image/signal processing algorithms

3. How the HLS tool handles time (clock cycles)
   - Key issue for choosing between fast/large or slow/small IPs

Specifying a hardware IP in C ?

- A C program = **`main`** + secondary functions
  - Mapping main to hardware does not make sense …

```
int main(int a){
   …
   res = foobar(12);
   if (res!=2) error(-1)
}

int foobar(int a){


   …
}
```

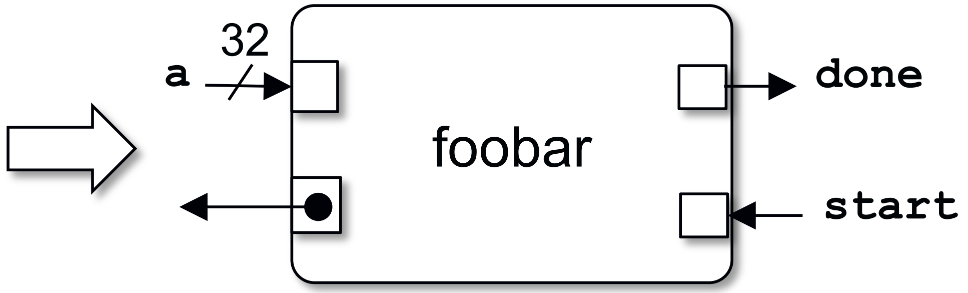


- A C-HLS description = Testbench + hardware IP
  - User specified function synthesized as a hardware IP
  - Main + others function serve as testbench for testing

Inferring a component from C code

- IP interface is inferred from the function prototype
  - Specific ports for controlling the component (start, end, etc.)
  - Simple data bus (scalar arguments of function results)

```
int foobar(int a){

   ...

}
```

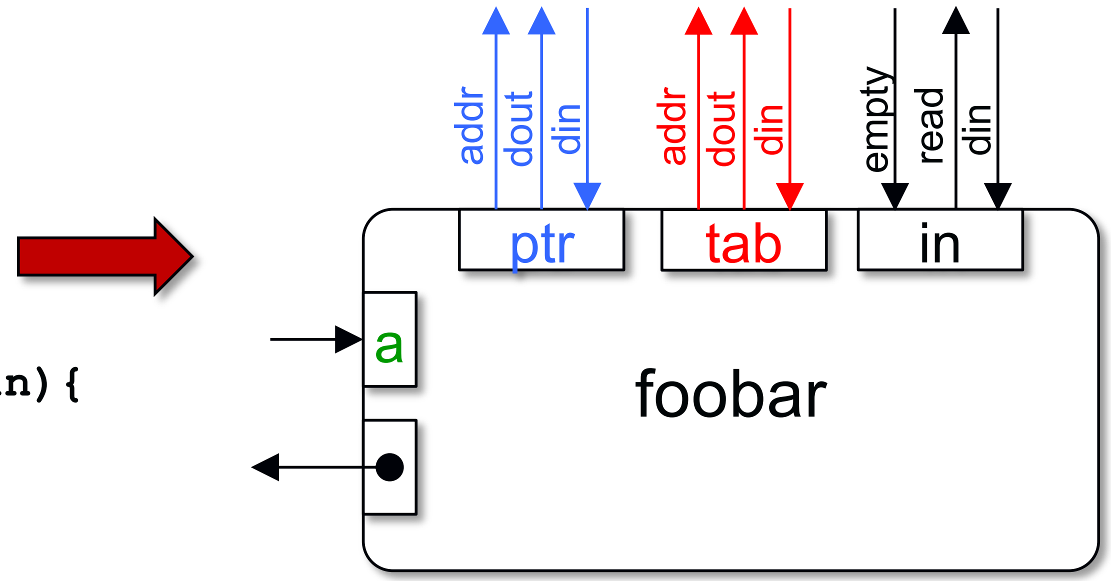


- HLS tools generates two descriptions
  - A synthesizable HDL description of the hardware IP
  - A behavioral HDL/SystemC description of the testbench
  - Testing = making sure C and HDL simulation is equivalent

Interfacing with buses, memories, etc.

- Most IPs need to handle complex interface protocols
  - To read/write from an external (e.g off chip) memory
  - To read/write to/from streams and/or FIFO buffers
  - To access a shared CPU bus (AXI, Avalon, Amba, etc).
- Special interface are inferred using argument type
  - Can be customized through annotations (pragmas)

```
int foobar(
          int a,
          int* ptr,
          int tab[],
          in_stream<int>* in){
   …
}
```



Key things to understand

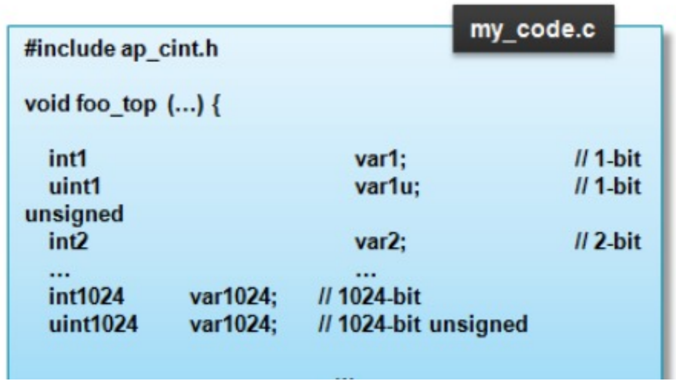
1. How the HLS tool *infers* the component interface (I/Os)
   - Key issue for integrating the component into other designs

2. How the HLS tool handles data types and memory
   - Key issue when dealing with image/signal processing algorithms

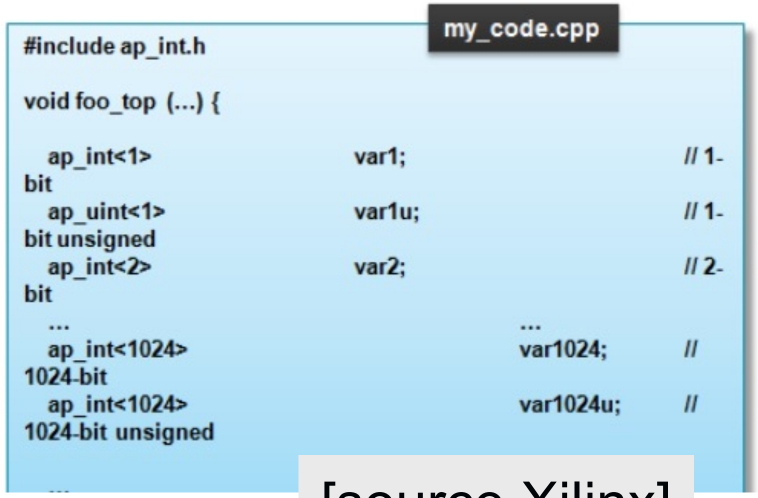
3. How the HLS tool handles time (clock cycles)
   - Key issue for choosing between fast/large or slow/small IPs

Hardware "bit accurate" datatypes

- C/C++ standard types limit hardware design choices
  - Hardware implementations use a wide range of wordlength
  - Use the smallest wordlength keeping algorithm correctness.
- HLS provides bit accurate types in C and C++
  - SystemC and vendor specific types supported to simulate custom wordlength hardware datapaths in C/C++



```
my_code.c
#include ap_cint.h

void foo_top (...) {

  int1                     var1;         // 1-bit
  uint1                    var1u;        // 1-bit
unsigned
  int2                     var2;         // 2-bit
  ...                      ...
  int1024     var1024;   // 1024-bit
  uint1024    var1024;   // 1024-bit unsigned
```

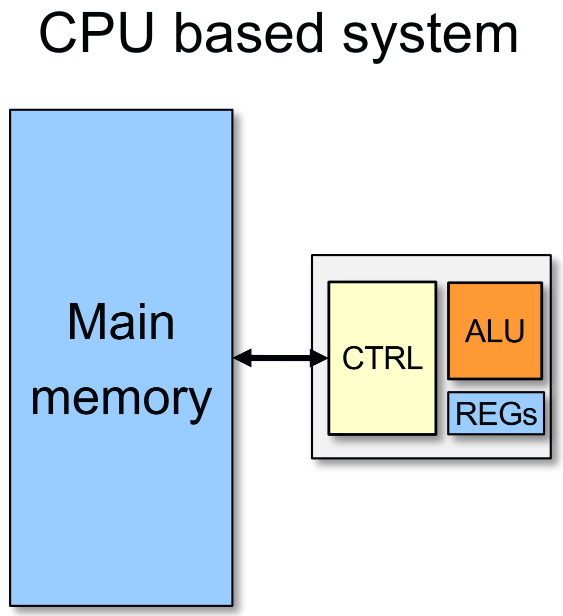


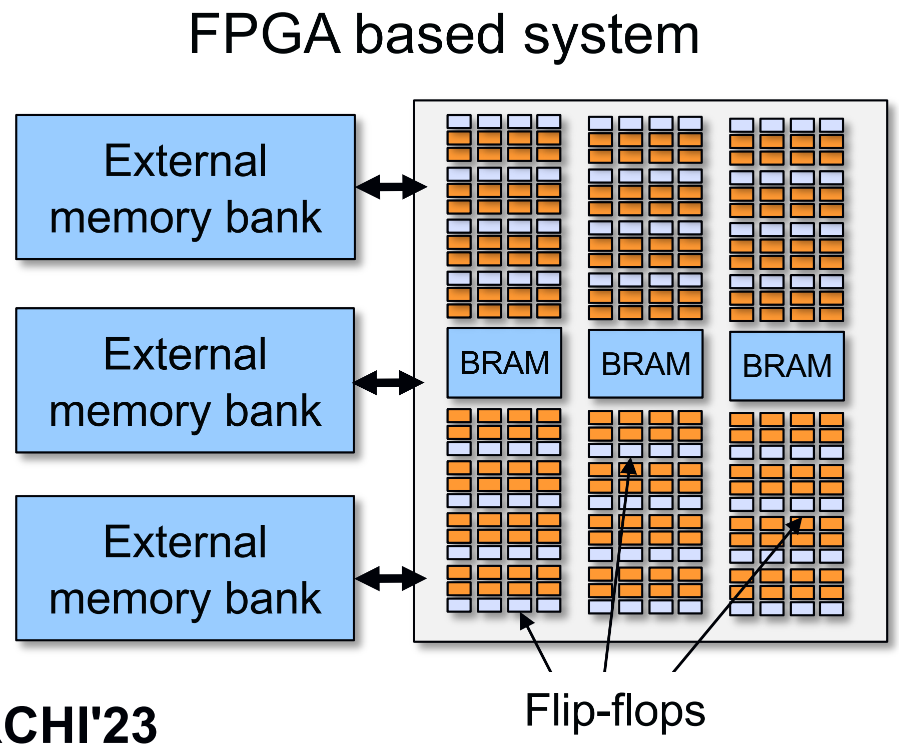
```
my_code.cpp
#include ap_int.h

void foo_top (...) {

  ap_int<1>                var1;                    // 1-
bit
  ap_uint<1>               var1u;                   // 1-
bit unsigned
  ap_int<2>                var2;                    // 2-
bit
  ...                                   ...
  ap_int<1024>                          var1024;    //
1024-bit
  ap_int<1024>                          var1024u;   //
1024-bit unsigned

  ...
```

[source Xilinx]

How to manage memory ?

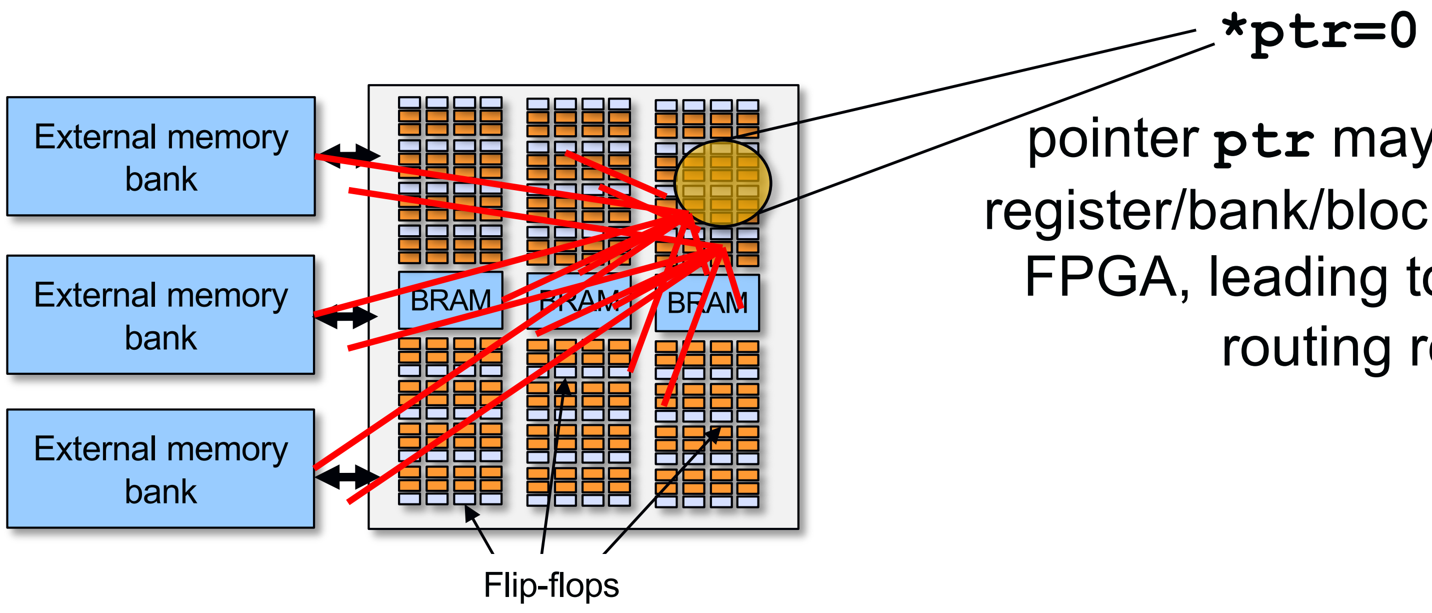
- In C/C++ memory is a large & flat address space
  - Enabling pointer arithmetic, dynamic allocation, etc.
  - Targeting a CPU based memory system
- In FPGAs or ASICs memory hierarchy is a mess ..
  - Registers, on-chip memory blocks, external memory banks,





HLS memory restrictions

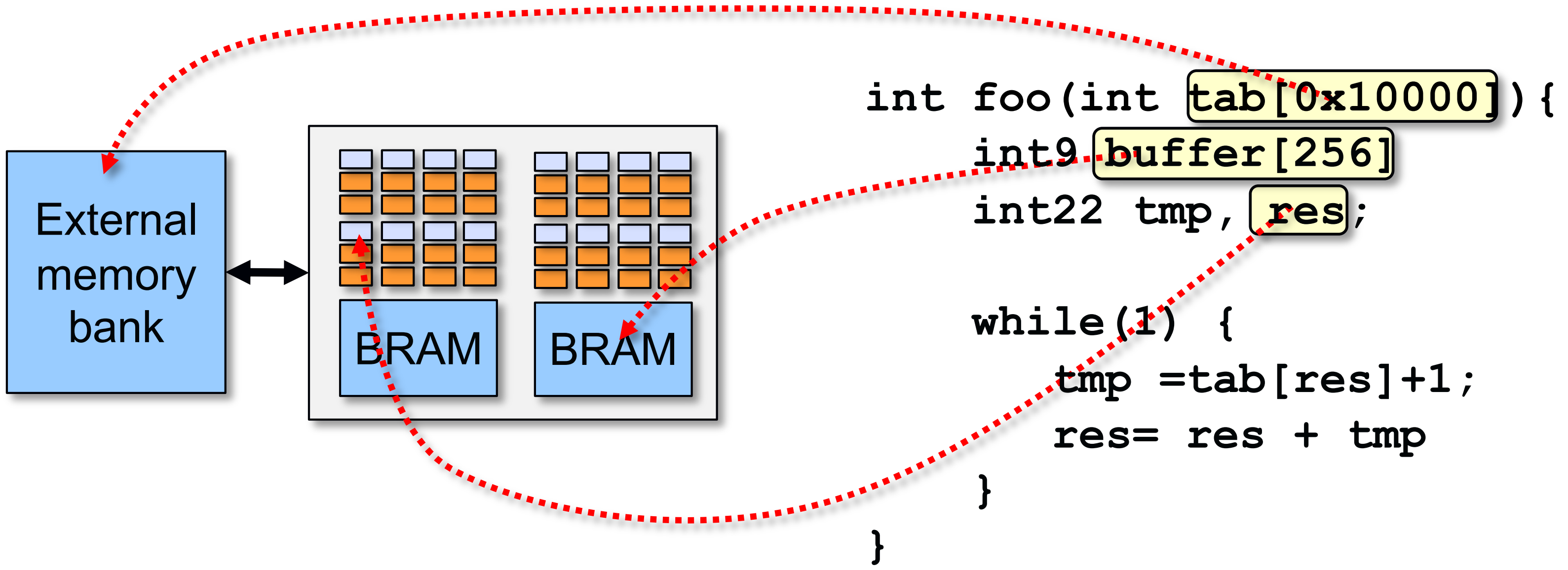
- No dynamic memory allocation (no malloc/free)
  - Cannot “create” memory space dynamically
- Use of global variables is forbidden
  - This is OK since HLS operates at the function level anyway
- Limited support for pointers because of aliasing



pointer **ptr** may refer to any register/bank/bloc in/out of the FPGA, leading to impractical routing requirements

Mapping of variable to memoy

- Mapping of program variables infered from source code
  - Scalar variables are mapped to flip-flops or wires
  - Local arrays are stored in on-chip memory blocks
  - External arrays (arguments) are stored in external memory



```
int foo(int tab[0x10000]){
    int9 buffer[256]
    int22 tmp, res;

    while(1)  {
        tmp =tab[res]+1;
        res= res + tmp
    }
}
```

Question : what happens to tmp ?

Key things to understand

1. How the HLS tool *infers* the component interface (I/Os)
   - Key issue for integrating the component into other designs

2. How the HLS tool handles data types and memory
   - Key issue when dealing with image/signal processing algorithms

3. How the HLS tool handles time (clock cycles)
   - Key issue for choosing between fast/large or slow/small IPs

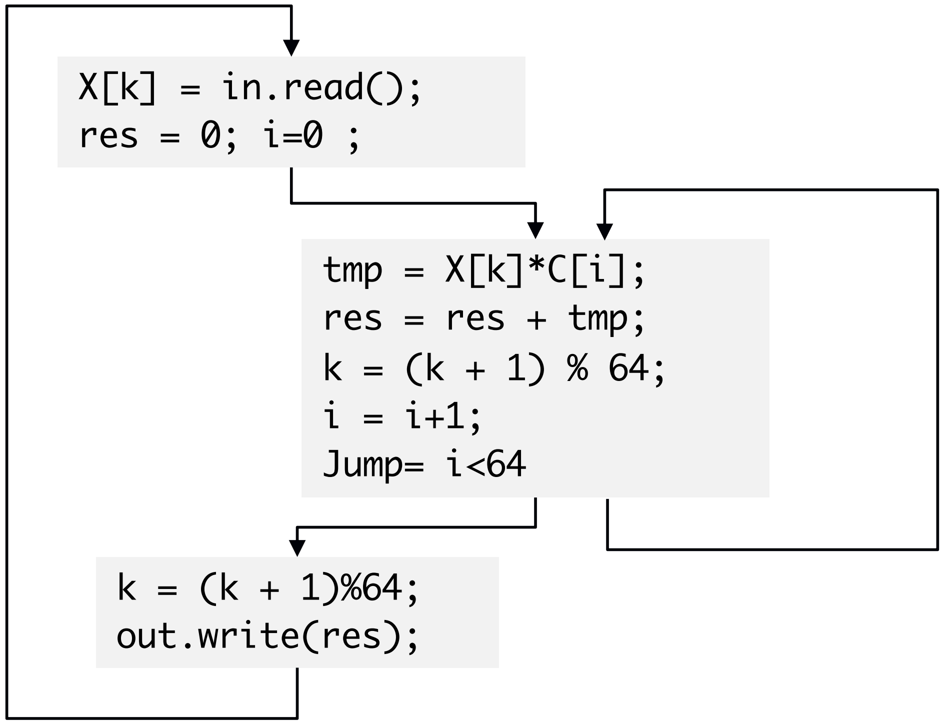
Where is time ?

- In C/C++, there is no formal notion of time
  - This is a problem : we need synchronous (clocked) hardware
- HLS tools *infer* timing using two approaches
  - Implicit semantics (meaning) of language constructs
  - User defined compiler directives (annotations)
- Extending C/C++ with ***an implicit semantics*** ???
  - They add a temporal meaning to some C/C++ constructs
  - Ex : a loop iteration takes at least one cycle to execute
- Best way to understand is through examples …

Program representation

- The tools ”sees” the C code as a state flow chart

```
while(1) {
   X[k] = in.read();
   res = 0; i=0;
   for (;i<64;i++) {
      tmp = X[k]*C[i];
      res = res + tmp;
      k = (k + 1) % 64;
   }
   k = (k + 1)%64;
   out.write(res);
}
```



```
X[k] = in.read();
res = 0; i=0 ;
```

```
tmp = X[k]*C[i];
res = res + tmp;
k = (k + 1) % 64;
i = i+1;
Jump= i<64
```

```
k = (k + 1)%64;
out.write(res);
```

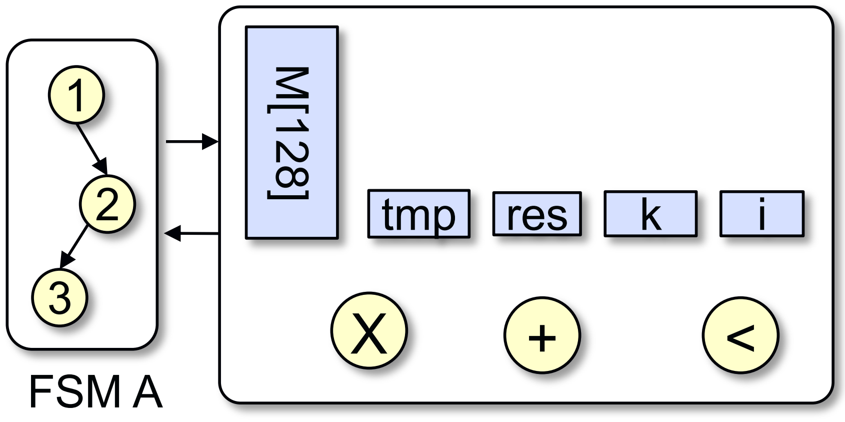
- Transitions between blocks can be handled by an FSM
  - Transition evaluated on when a block execution is “finished”
  - Need to select how/when to execute operations within the block

Scheduling/mapping in a block

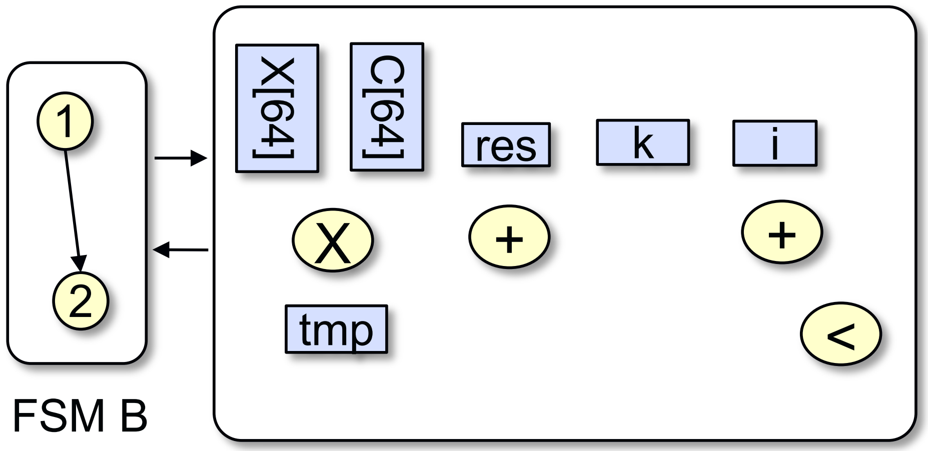
- For each block the tool creates a datapath + FSM design
  - Several implementations are possible !
  - Trade-off between # clock ticks and resource usage

```
Tick 1:    tmp = M[k+64] // X[k]
           k = (k + 1) % 64;
-----------------------------------------
Tick 2:    tmp = tmp * M[i] //C[i]
           i = i+1;
-----------------------------------------
Tick 3:    res = res + tmp
           Jump= i<64
```

```
Tick 1:    tmp = X[k]*C[i],
           k = (k + 1) % 64,
           i = i+1;
-----------------------------------------
Tick 2:    res = res + tmp,
           Jump= i<64
```



1 add, 1 mem, 1 mul, 4 regs

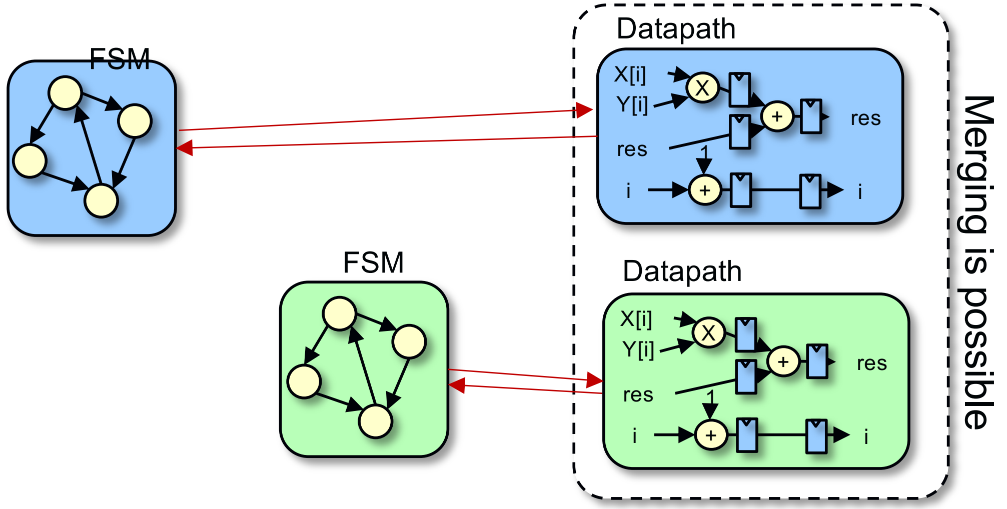


2 add, 2 mem, 1 mul, 4 regs

Scheduling conditionals

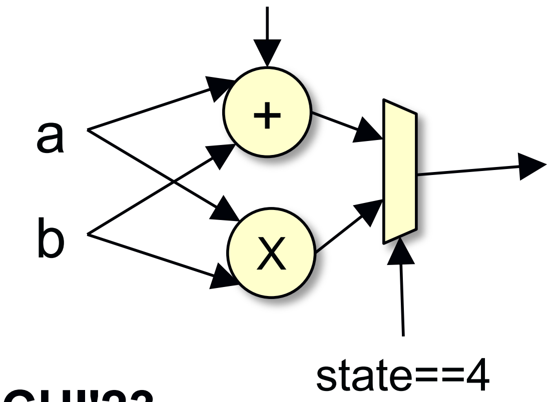
- HLS tools have two different ways of handling them
  - By considering branches as two different blocks

```
if(state==4){
  for (int i=0;i<32;i++) {
    res += X[i]/Y[i];
  }
} else {
  for (int i=0;i<32;i++) {
    res += 1/X[i];
  }
}
```



  - By executing both branches and select the correct results if then/else do not contain loops function calls.

```
if(state==4){
  res = a+b;
} else {
  res = a*b;
}
```



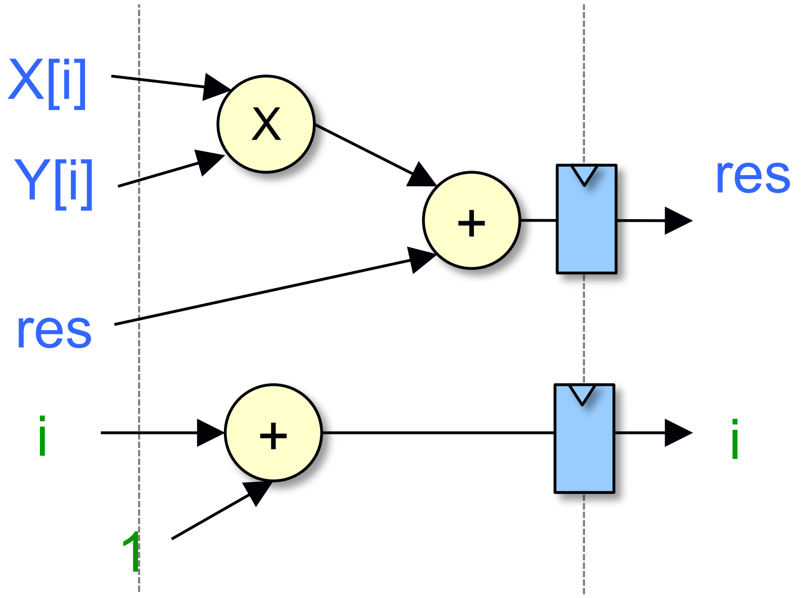
Scheduling for/while loops

- Loops are one of the most important program constructs
  - Most signal/image kernel consists of (nested) loops
  - They are often the reason for resorting to hardware IPs
- Their support in HLS tools vary a lot between vendors
  - Best in class : Vitis HLS, Synphony, Catapult-C,
- HLS users spent most on their time optimizing loops
  - Directly at the source level by tweaking the source code
  - By using compiler/script based optimization directives

Loop single-cycle execution

- HLS default behavior is to execute one iteration/cycle
  - Executing the loop with N iteration takes $N+1$ cycles
    - Extra cycle for evaluating the exit condition

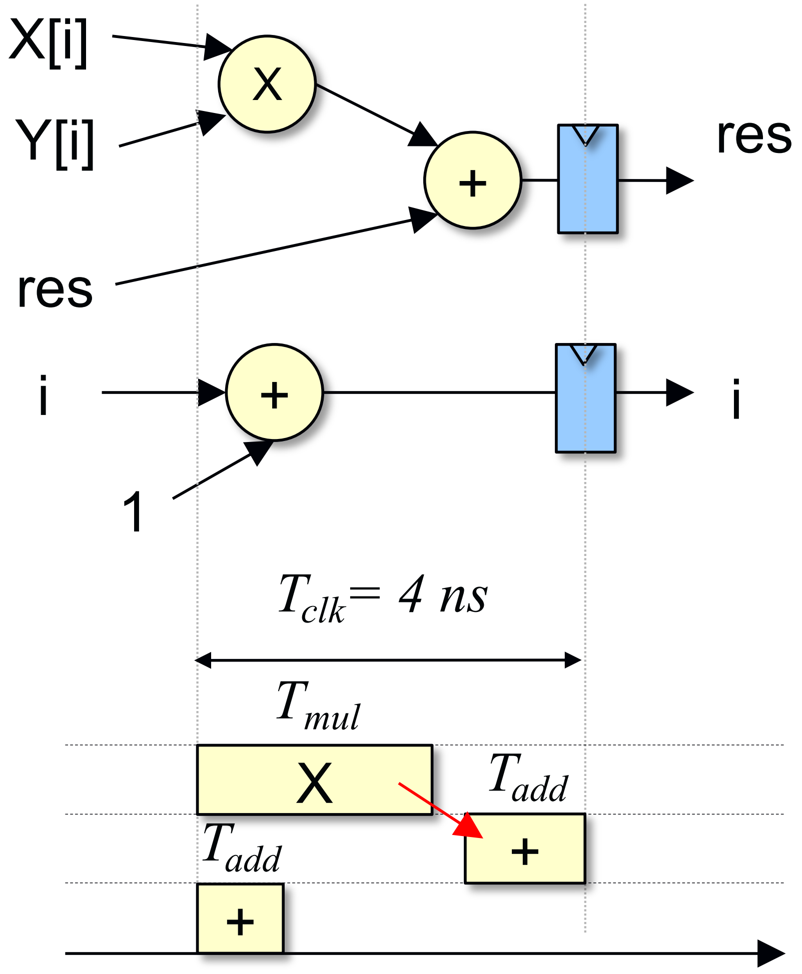
```
float X[512],Y[512];
res=0.0;
for (int i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp;
}
```



- Each operation is mapped to its own operator
  - No resource sharing/hardware reuse

Loop single-cycle execution

- Critical path (f_{max}) depends on loop body complexity
  - Let us assume that $T_{mul}=3ns$ and $T_{add}=1ns$



✘ No control on the clock speed !

```
int X[512],Y[512];
int tmp,res=0;
for (int i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp;
}
```

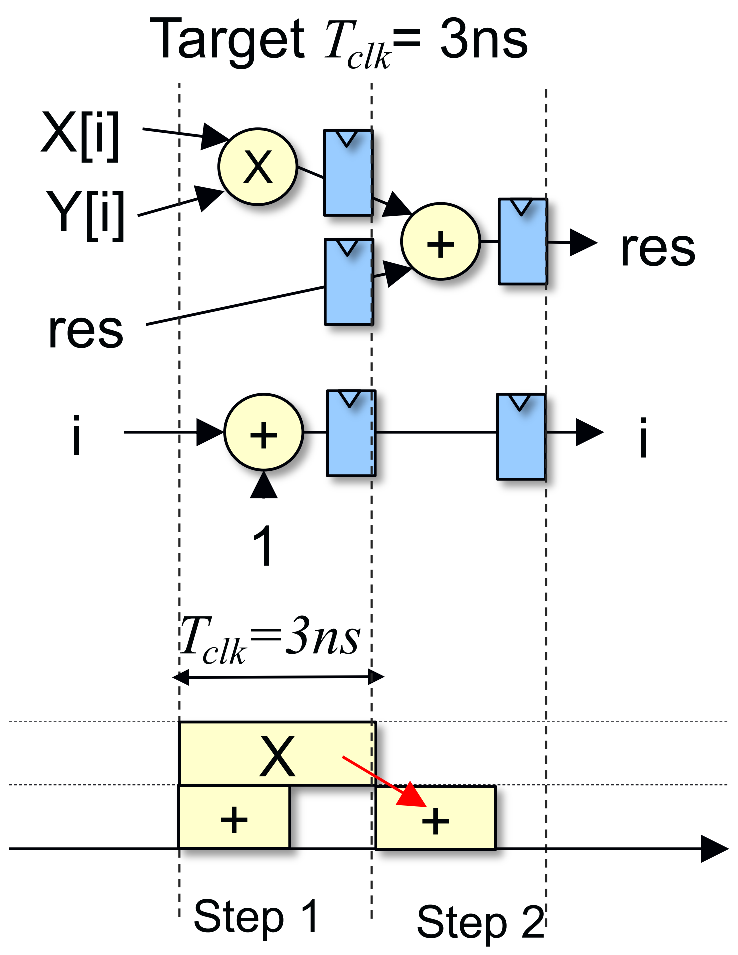
#cycles	f_{clk}	Exec time

Multi-cycle execution

- Most HLS tools handle constraints over T_{clk}
  - Body execution is split into several shorter steps (clock-ticks)
  - Needs accurate information about target technology

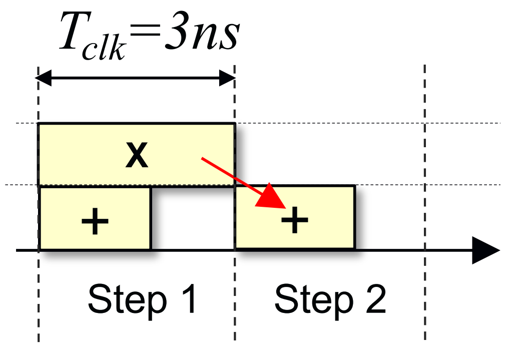
```
int X[512],Y[512];
int tmp,res=0;
for (int i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp;
}
```

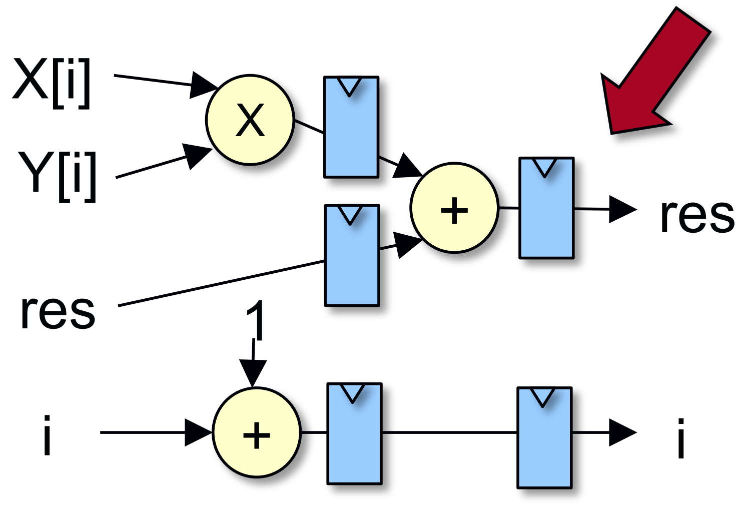
#cycles	f_{clk}	Exec time



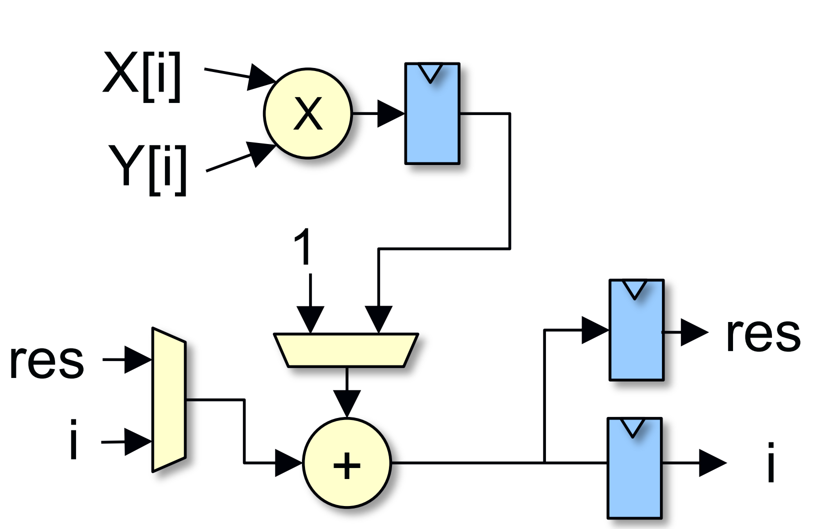
Hardware reuse in multi-cycle execution

- For multi-cycle execution, HLS enable operators reuse
  - HLS tools balance hardware operator usage of over time
  - Trade-off between operator cost and muxes overhead





1 multiplier, 2 adder

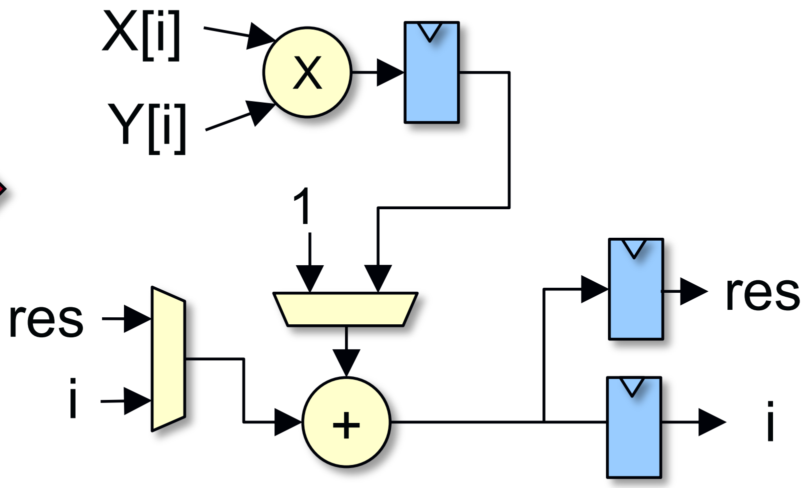


1 multiplier, 1 adder, ***2 muxes***

Hardware reuse in multi-cycle execution

- Hardware reuse rate is controlled by the user
  - Trade-off between parallelism (performance) and reuse (cost)
  - Users provides constraints on the # and type of resources
  - Mostly used for multipliers, memory banks/ports, etc.
- The HLS tools decides what and when to reuse
  - Optimizes for area or speed, following user constraints
  - Combinatorial optimization problem (exponential complexity)

```
int X[512],Y[512];
int tmp,res=0;
#pragma HLS mult=1,adder=1
for (int i=0;i<512;i++){
   tmp = X[i]*Y[i];
   res = res + tmp;
}
```



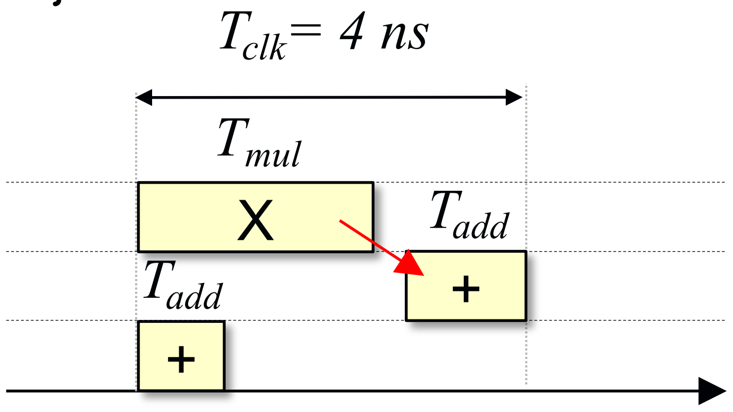
1 multiplier, 1 adder, ***2 muxes***

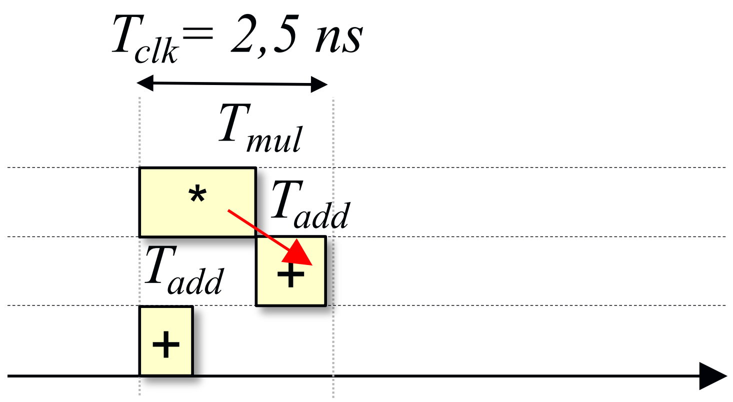
How to further improve performance ?

- Improve f_{max} with arithmetic optimizations
  - Avoid complex/costly operations as much as possible
  - Aggressively reduce data wordlength to shorten critical path

```
int X[512],Y[512];
int tmp,res=0;
for (int i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp;
}
```

```
int12 X[512],Y[512];
int16 res=0; int24 tmp;
for (int9 i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp>>8;
}
```





Very effective as it improves performance and reduce cost

High Level Synthesis

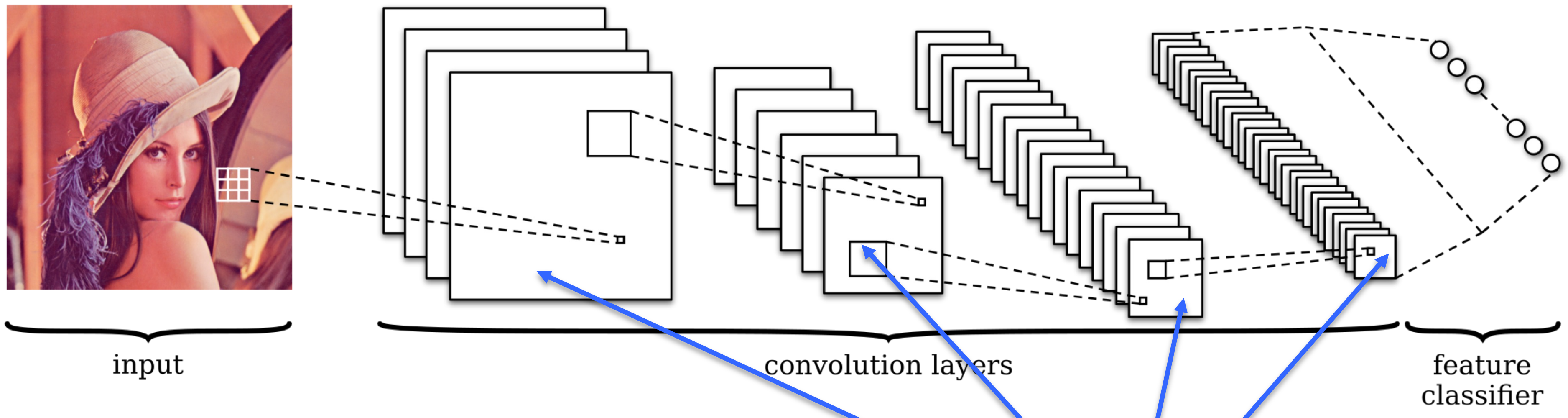
DSE for FPGA accelerators using HLS

Key things to understand

1. How to take advantage of parallelism in the C kernel
   - Loop unrolling
   - Loop pipelining
   - Loop fusion
2. How to optimize memory accesses
3. How to maximize hardware utilization rate
4. Understanding & circumventing HLS tool limitations

Example : CNN inference

- CNN organized in layers, with mainly convolution layers



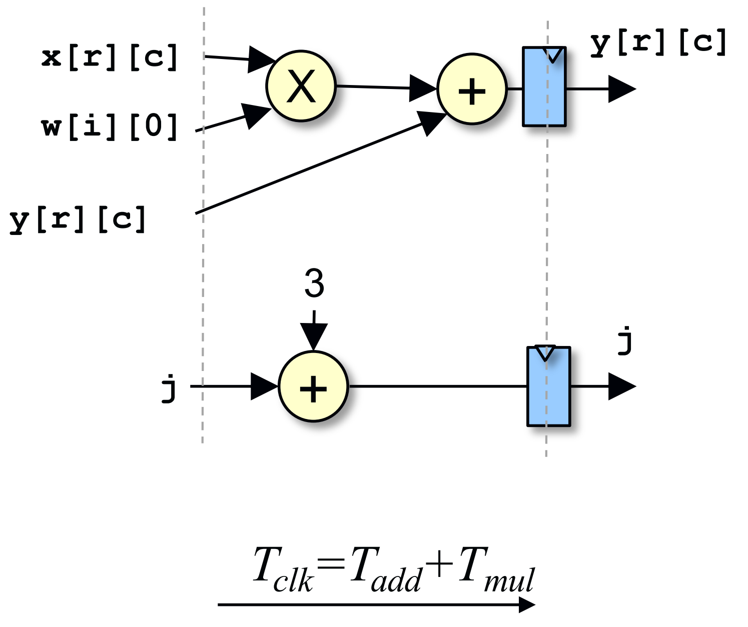
```
for(to = 0; to < M ; to++) {
  for(ti = 0; ti < N ; ti++) {
    for(row = 0; row < R ; row++) {
      for(col = 0; col < C ; col++) {
        for(i = 0; i < K ; i++) {
          for(j = 0; j < K ; j++) {
                 y[to][row][col] +=
              w[to][ti][i][j] *
              x[ti][S*row+i][S*col+j];
} } } } } }
```

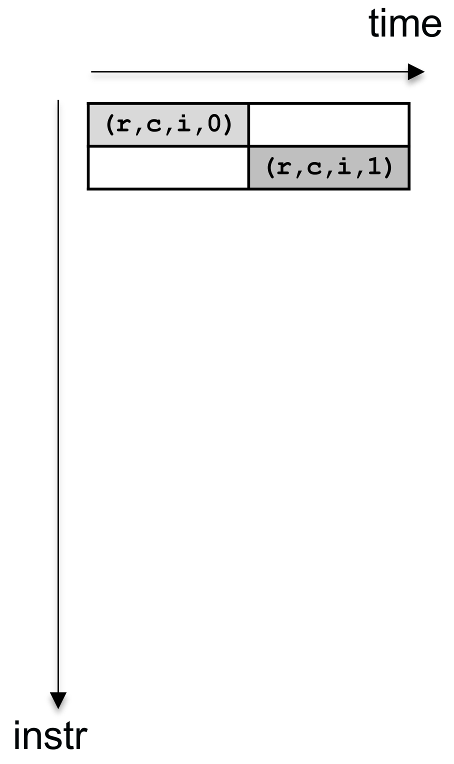
$$y_{r,c,m} = \sum_{n=0}^{N-1} \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} x_{Sr+i,Sc+j,n} \cdot w_{i,j,n,m}$$

Accelerating a convolution layer

- What do we obtain from HLS with vanilla C code ?
  - one iteration/cycle for the innermost loop
  - Inefficient combinational datapath (for FPGAs)

```
ac_int<6> X[…][…];
ac_int<3> W[…][…];
ac_int<11> y;

for(r = 0; r < ROW ; row++) {
  for(c = 0; c < COL ; col++) {
    for(i = 0; i < 3 ; i++) {
      for(j = 0; j < 3 ; j++) {
        y[r][c] +=
           w[i][j] *
           x[r+i][c+j];
        }

} } }
```



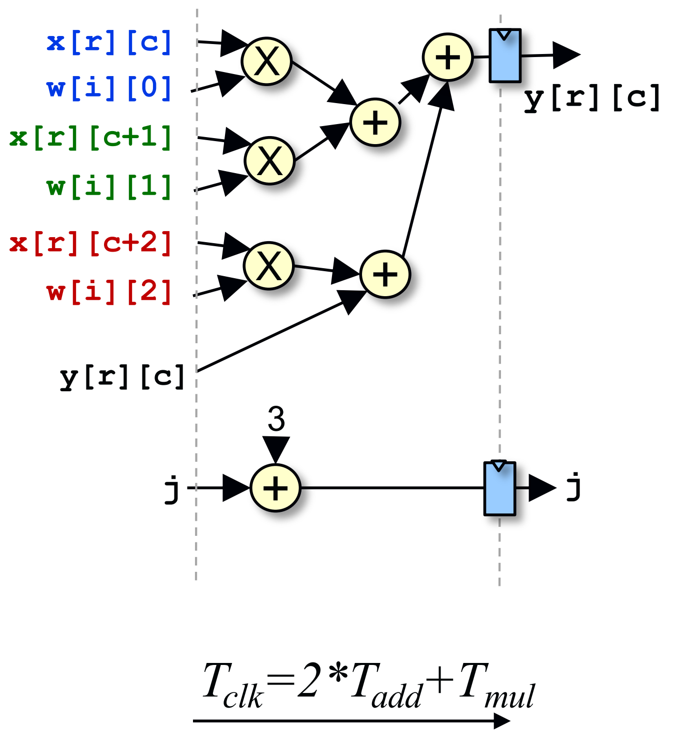


- This design does not exploit enough parallelism

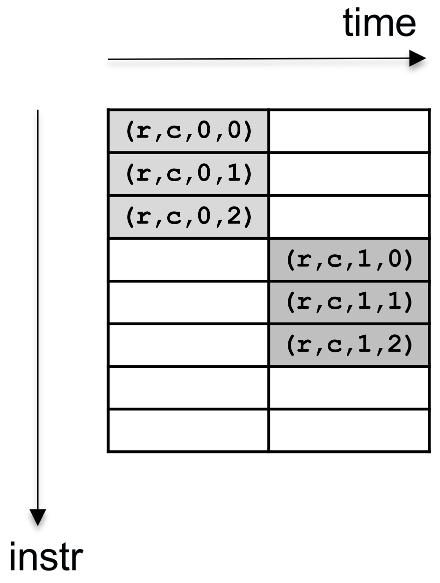
Exposing parallelism through unrolling

- Unrolling can controlled through **`#pragma`** directives
  - Full unrolling only for non constant loop bound is impossible
  - Partially unrolling for non constant loop bound is possible
- Example : full unrolling of the inner most loop

```
for(r = 0; r < ROW ; r++) {
  for(c = 0; c < COL ; c++) {
    for(i = 0; i < 3 ; i++) {
    #pragma unroll
    for(j = 0; j < 3 ; j++) {
       y[r][c] +=
          w[i][j] *
          x[r+i][c+j];
    }

} } }
```



$T_{clk}=2*T_{add}+T_{mul}$



(r,c,0,0)	
(r,c,0,1)	
(r,c,0,2)	
	(r,c,1,0)
	(r,c,1,1)
	(r,c,1,2)

Unrolling vs Interchange

- Unrolling can be combined with *loop interchange**
  - Loop interchange = swapping loops in a loop nest
- Example : interchanging loops **`j`** and **`c`**.

```
for(r = 0; r < ROW ; r++) {
  for(c = 0; c < COL ; c++) {
    for(i = 0; i < 3 ; i++) {
      #pragma unroll
      for(j = 0; j < 3 ; j++) {
         y[r][c] +=
           w[i][j] *
           x[r+i][c+j];
      }

} } }
```

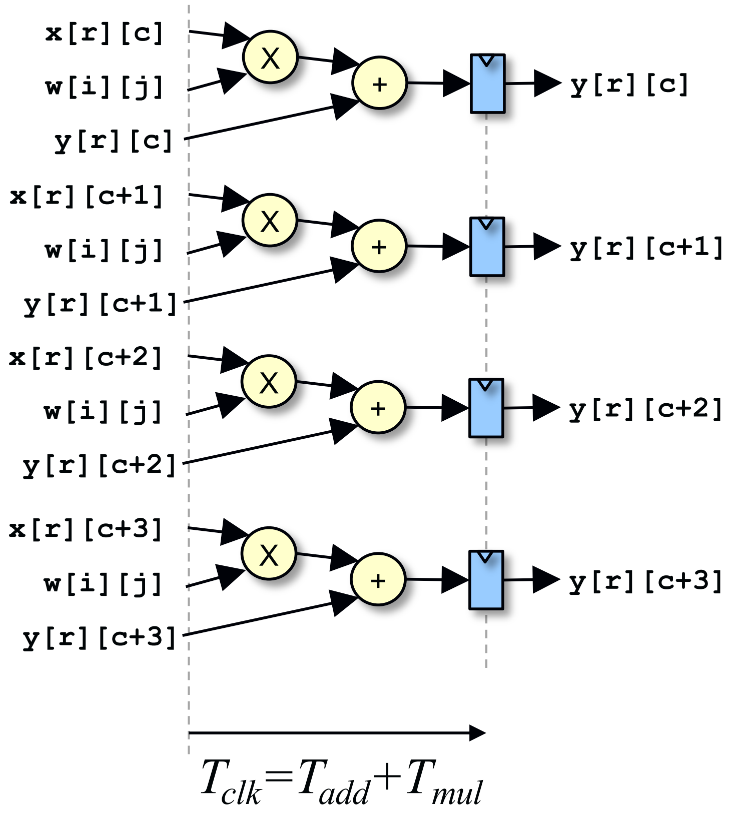
```
for(r = 0; r < ROW ; r++) {
  for(i = 0; i < 3 ; i++) {
    for(j = 0; j < 3 ; j++) {
      #pragma unroll 4
      for(c = 0; c < COL ; c++) {
         y[r][c] +=
           w[i][j] *
           x[r+i][c+j];
       }

} } }
```

* when interchange is possible

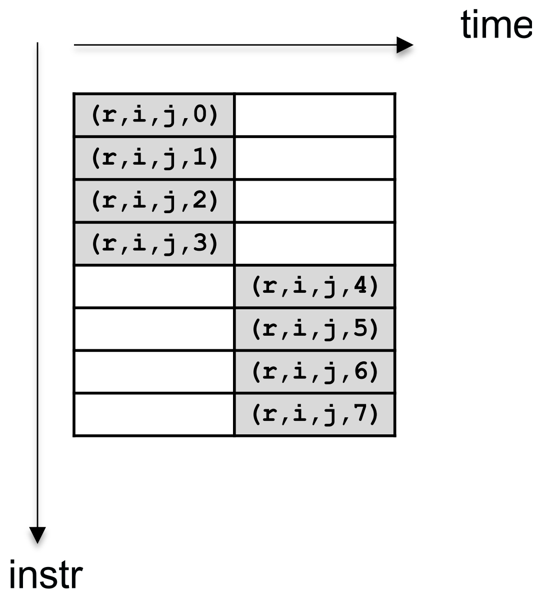
Exposing parallelism with unrolling

- Larger unrolling factors are now possible along index c
  - We can explore +/- parallelism using partial unrolling
- Example : partial unrolling by 4 after interchange

```
for(r = 0; r < ROW ; r++) {
  for(i = 0; i < 3 ; i++) {
    for(j = 0; j < 3 ; j++) {
      #pragma unroll 4
      for(c = 0; c < COL ; c++) {
        y[r][c] +=
          w[i][j] *
          x[r+i][c+j];
      }

} } }
```



$T_{clk}=T_{add}+T_{mul}$

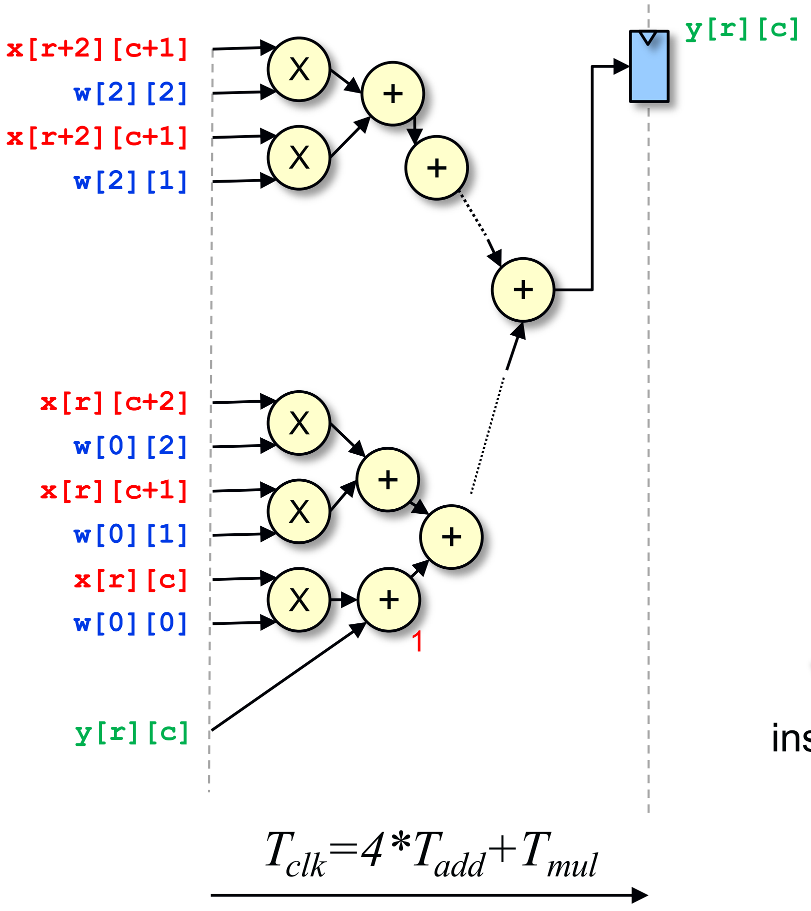


(r,i,j,0)	
(r,i,j,1)	
(r,i,j,2)	
(r,i,j,3)	
	(r,i,j,4)
	(r,i,j,5)
	(r,i,j,6)
	(r,i,j,7)

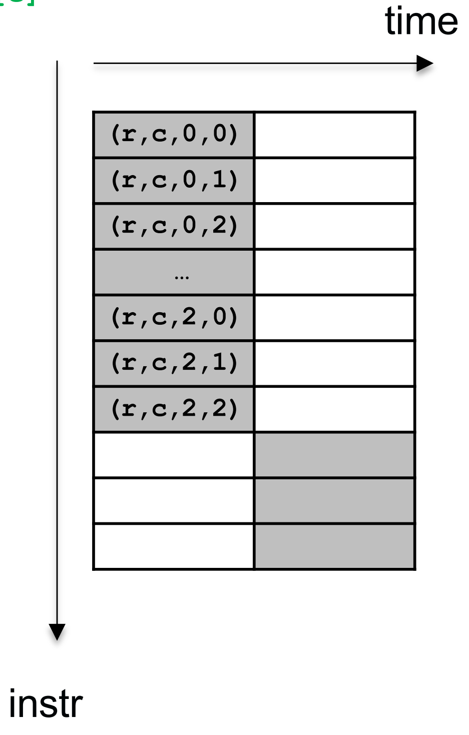
Exposing parallelism through unrolling

- Full unrolling can be applied to a loop nest
  - All innermost loops are fully unrolled
- Example : full unrolling of the two innermost loops

```
for(r = 0; r < ROW ; r++) {
  for(c = 0; c < COL ; c++) {
    #pragma unroll
    for(i = 0; i < 3 ; i++) {
       for(j = 0; j < 3 ; j++) {
         y[r][c] +=
            w[i][j] *
            x[r+i][c+j];
       }
    }
} } }
```



$T_{clk}=4*T_{add}+T_{mul}$



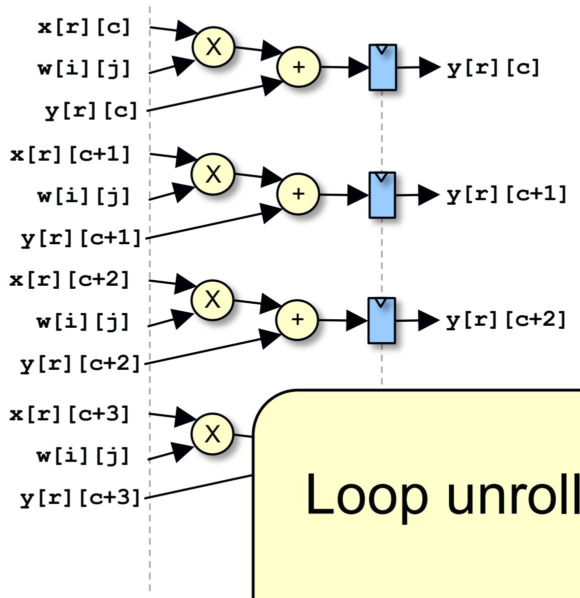
(r,c,0,0)	
(r,c,0,1)	
(r,c,0,2)	
…	
(r,c,2,0)	
(r,c,2,1)	
(r,c,2,2)	
	▨
	▨
	▨

9 iteration/cycle

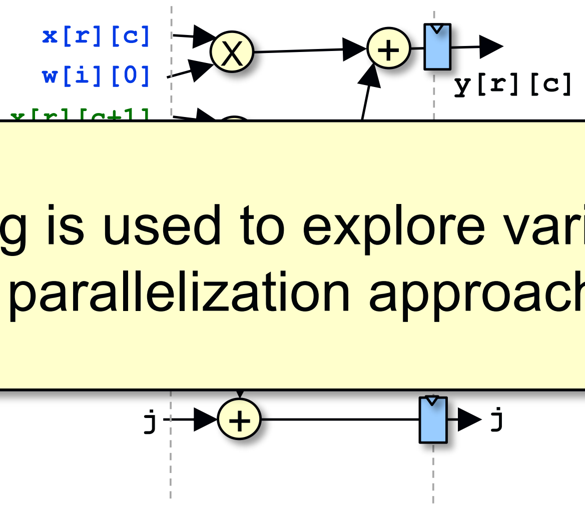
Summary

```
for(r = 0; r < ROW ; r++) {
  for(c = 0; c < COL ; c++) {
    for(i = 0; i < 3 ; i++) {
      for(j = 0; j < 3 ; j++) {
        y[r][c] +=
           w[i][j] *
           x[r+i][c+j];
      }
} } }
```

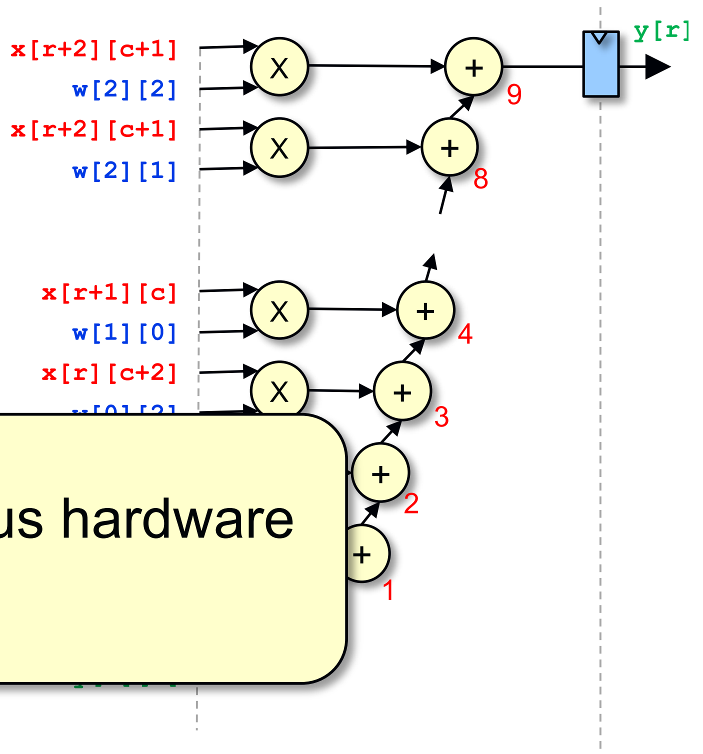
Interchange **c,j**
Partial unroll x4



Full unroll for **j**



Full unroll for **i,j**

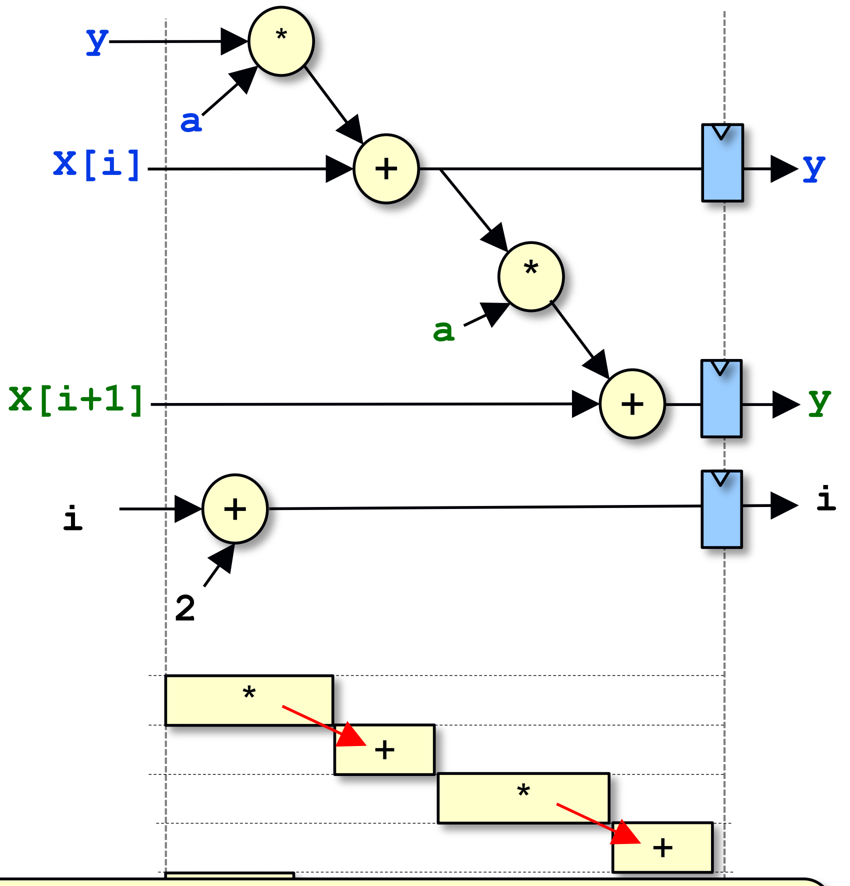


Loop unrolling is used to explore various hardware parallelization approach

Unrolling and loop carried dependencies

- Unrolling loops with carried dependencies

```
for(int i=0;i<256;i+=2){
    y = a * y + x[i];
    y = a * y + x[i];
}
```



Original version

#cycles	T_{clk}	Exec time
256	400 Mhz	0,64 us

Unrolled by x2

#cycles	T_{clk}	Exec time
128	200 Mhz	0.64 us

Unrolling does not “create” parallelism, it only uncovers it !

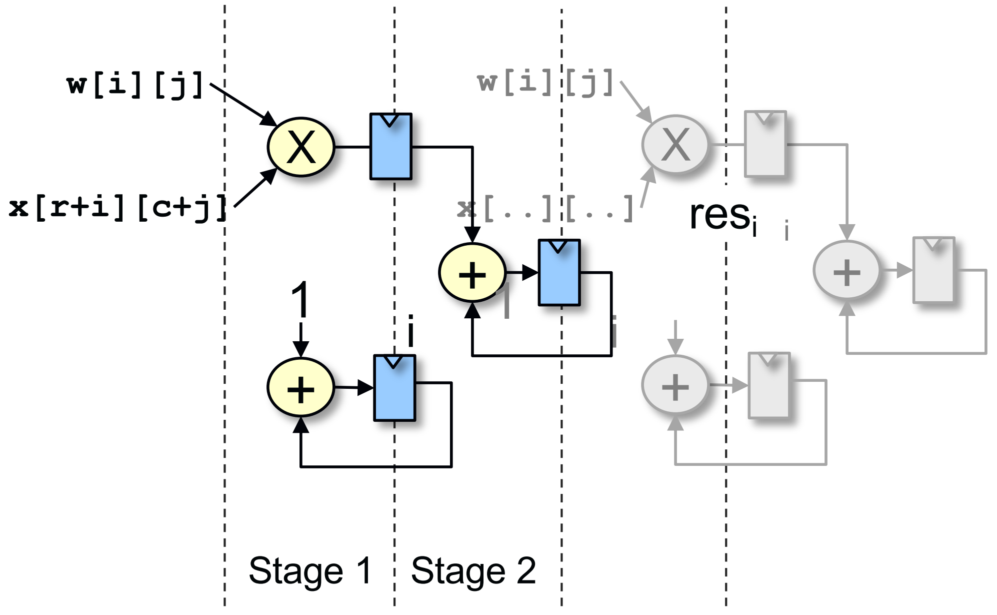
Key things to understand

1. How to take advantage of parallelism in the C kernel
   - Loop unrolling
   - **Loop pipelining**
   - Loop fusion
2. How to optimize memory accesses
3. How to maximize hardware utilization rate
4. Understanding & circumventing HLS tool limitations

Loop pipelining

- Loop Pipelining = pipelined execution of loop iterations
  - Start iteration j+1 before all operation of iteration j are finished

```
for(r = 0; r < ROW ; row++) {
  for(c = 0; c < COL ; col++) {
    for(i = 0; i < 3 ; i++) {
      for(j = 0; j < 3 ; j++) {
        y[r][c] +=
           w[i][j] *
           x[r+i][c+j];
        }

}  }  }
```

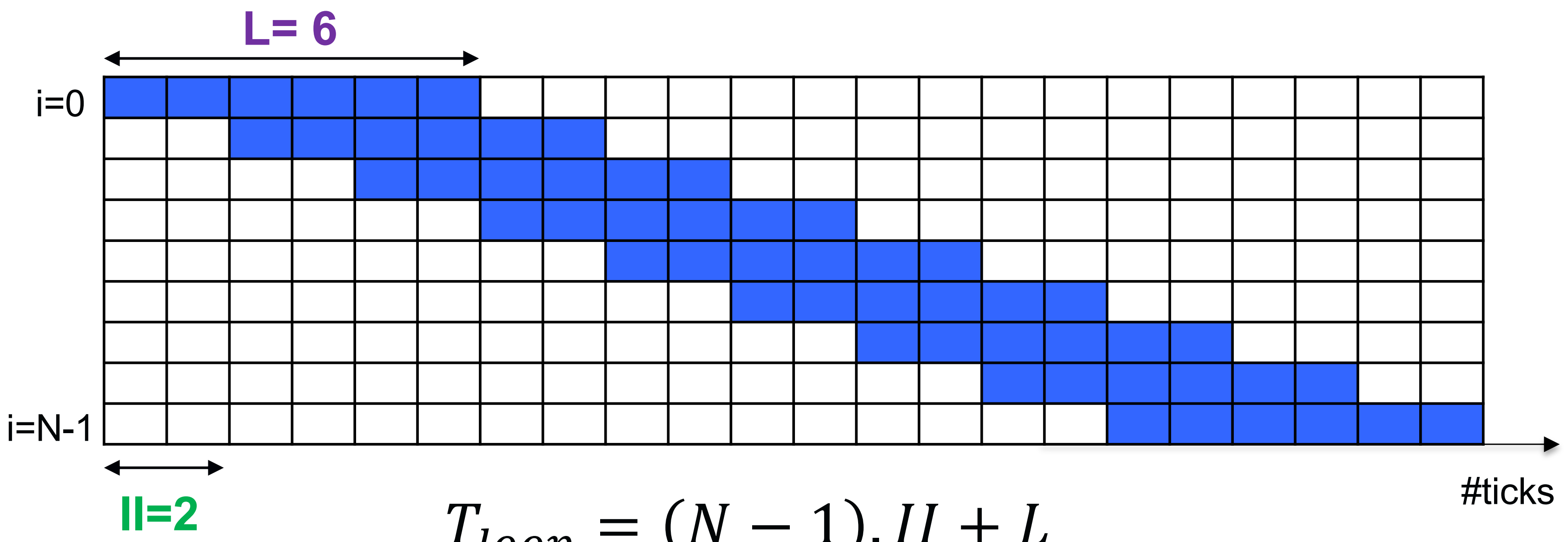


#cycles/ iteration	T_{clk}
1	500 Mhz

$$T_{clk}=max(T_{add},T_{mul})$$

Loop pipelining

- A given pipelined loop schedule is characterized by
  - **Initiation interval (II)** : delay between successive iterations.
  - **Pipeline Latency (L)** : #cycles to execute a given iteration

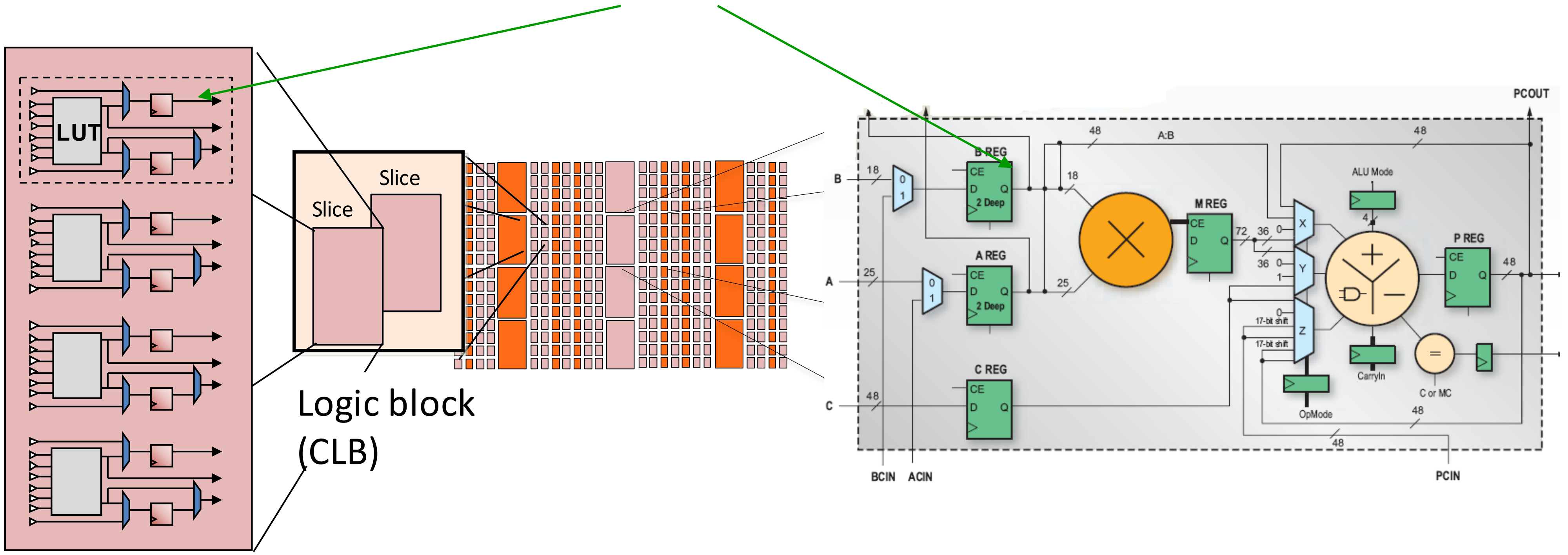


$$T_{loop} = (N-1).II + L$$

- Full pipelining when II=1(not always possible)
- Pipelining can be combined with unrolling

Fine grain pipelining in FPGAs

- FPGA have a lot of registers enabling deep pipelines



CLB SLICE

- On FPGAs 10s of stages are common (esp. for DSP)
  - Loop pipelining is therefore a key HLS optimization

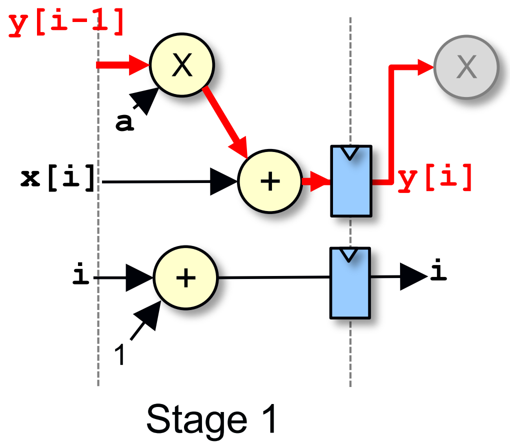
Pipelining and dependance distance

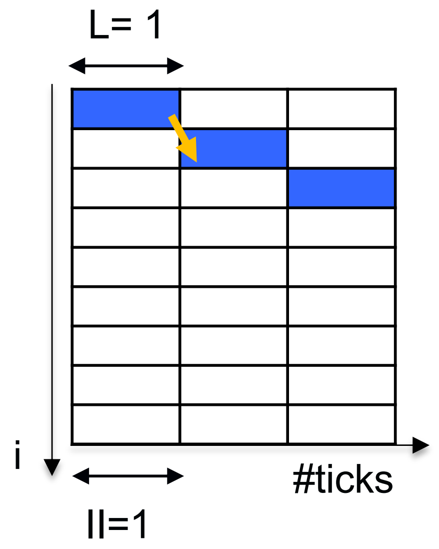
- Consider the two example below

```
for(i=1;i<256;i+=1){
    y[i] = a*y[i-1] + x[i];
}
```

Loop carried dependency over **y**

y[i] is reused at the next iteration

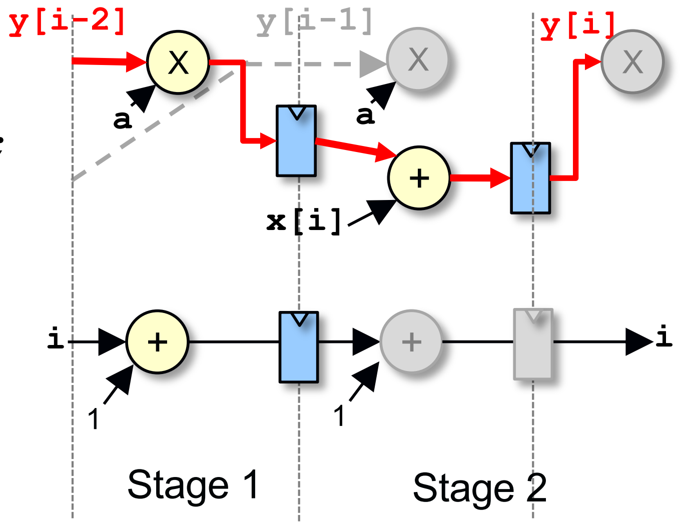


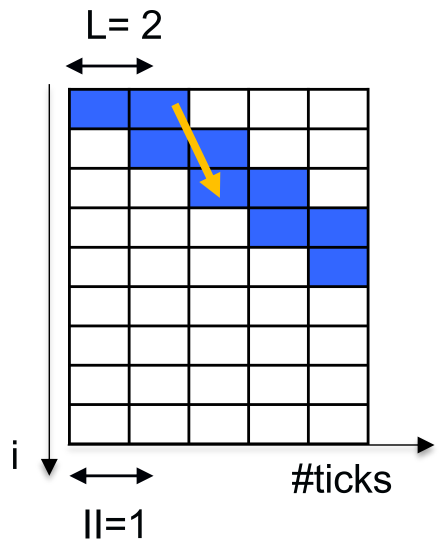


```
for(i=1;i<256;i+=1){
    y[i] = a*y[i-2] + x[i];
}
```

Loop carried dependency over **y**

y[i] is reused two iterations later

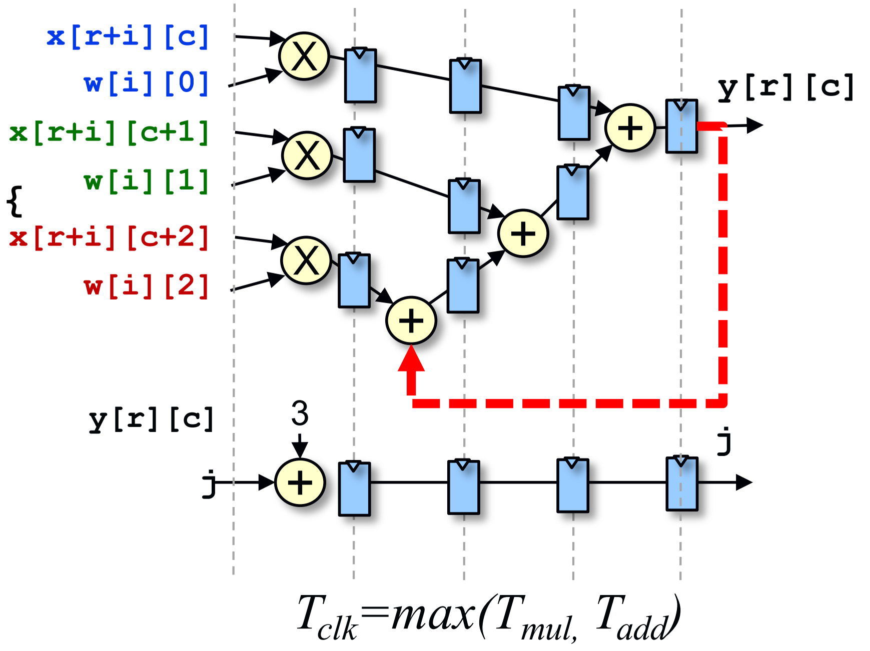


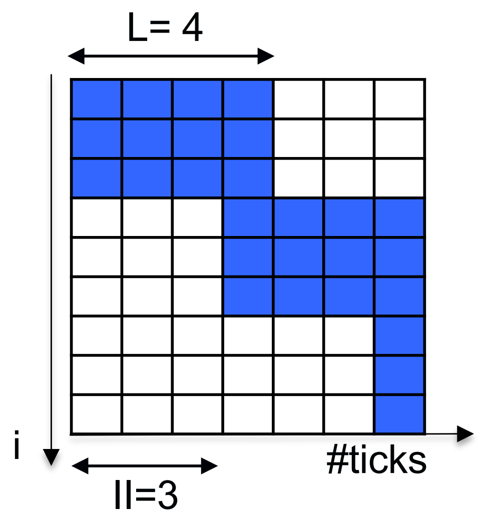


Loop pipelining + unrolling (1/2)

- Lead to a more parallel or deeper pipelined datapath
  - Depending on dependency patterns

```
#pragma unroll 3
for(j = 0; j < 3 ; j++) {
   y[r][c] +=
      w[i][j] *
      x[r+i][c+j];
   }
}
```



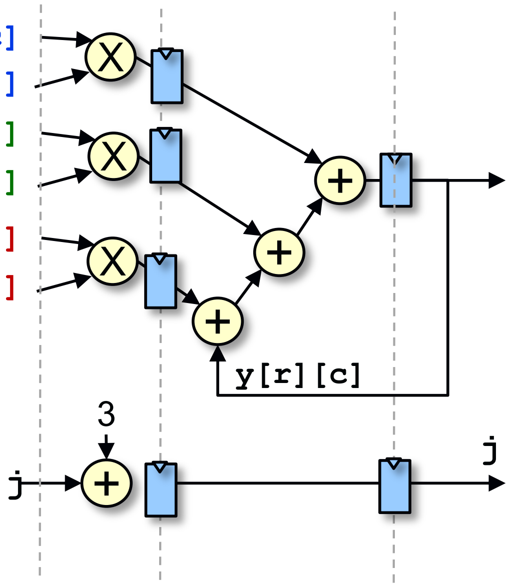


- Here, dependency over `y[r][c]` prevents full pipelining
  - We have a deep (but under utilized) pipeline

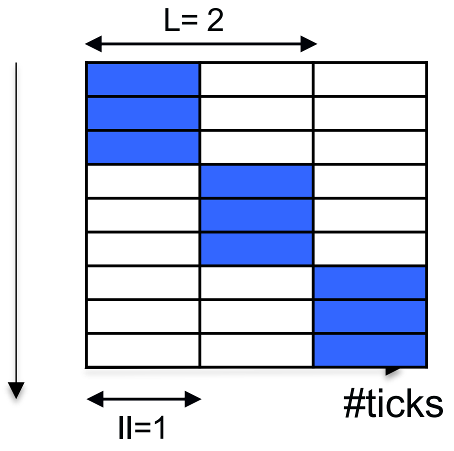
Loop pipelining + unrolling

- A less aggressive pipelined schedule might do as well
  - We can reduce clock speed f_{clk} in exchange of a lower II

```
#pragma unroll 3
for(j = 0; j < 3 ; j++) {
   y[r][c] +=
        w[i][j] *
        x[r+i][c+j];
   }
}
```



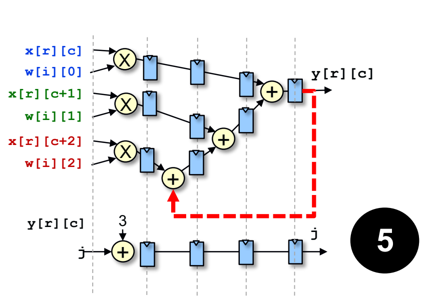
$T_{clk}=max(T_{mul},3T_{add})$

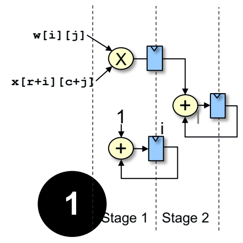


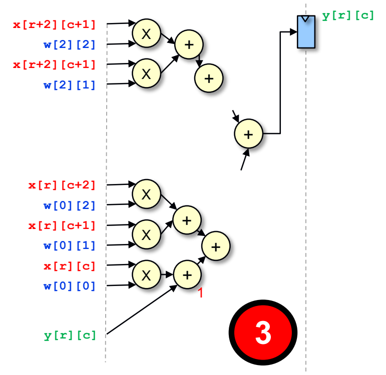
Pipelining does not “create” parallelism, it only uncovers it !

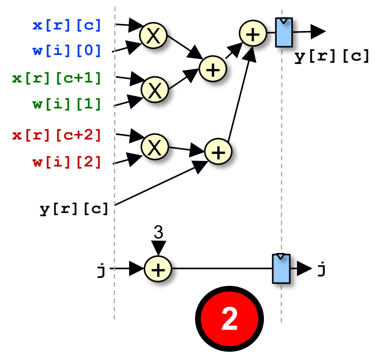
i

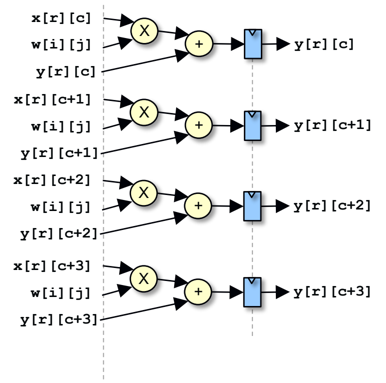
What is the best design then ?

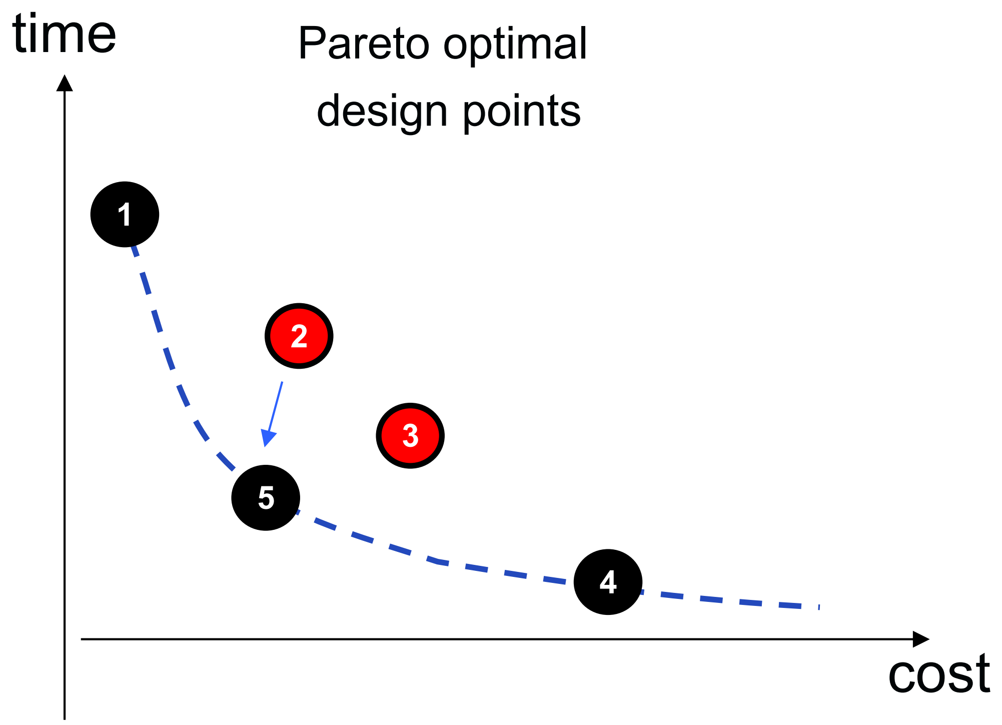












5 is both faster and cheaper than 2

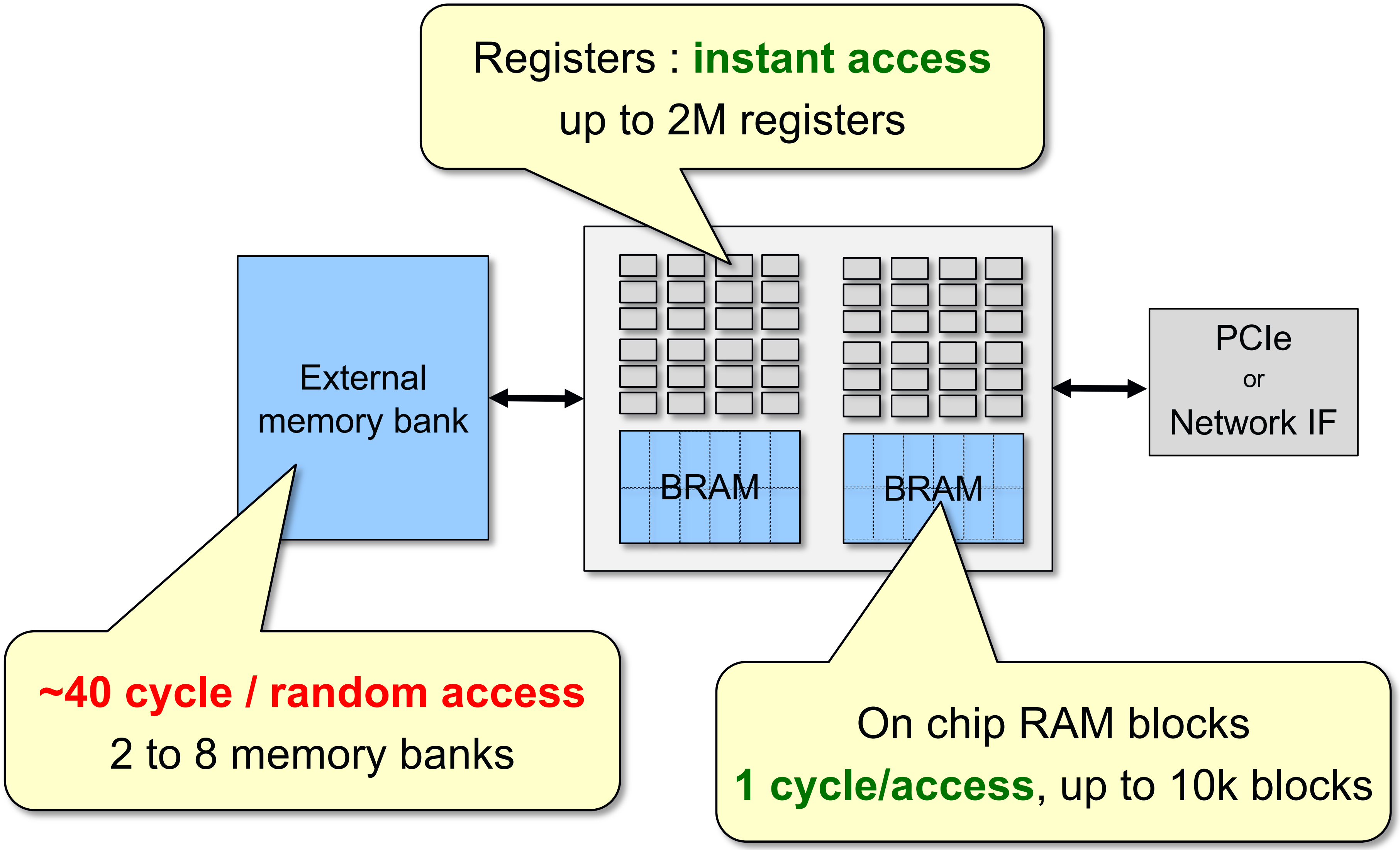
Key things to understand

1. How to take advantage of parallelism in the C kernel
   - Loop unrolling
   - Loop pipelining
2. How to optimize memory accesses
3. Understanding & circumventing HLS tool limitations
4. How to maximize hardware utilization rate

Key things to understand

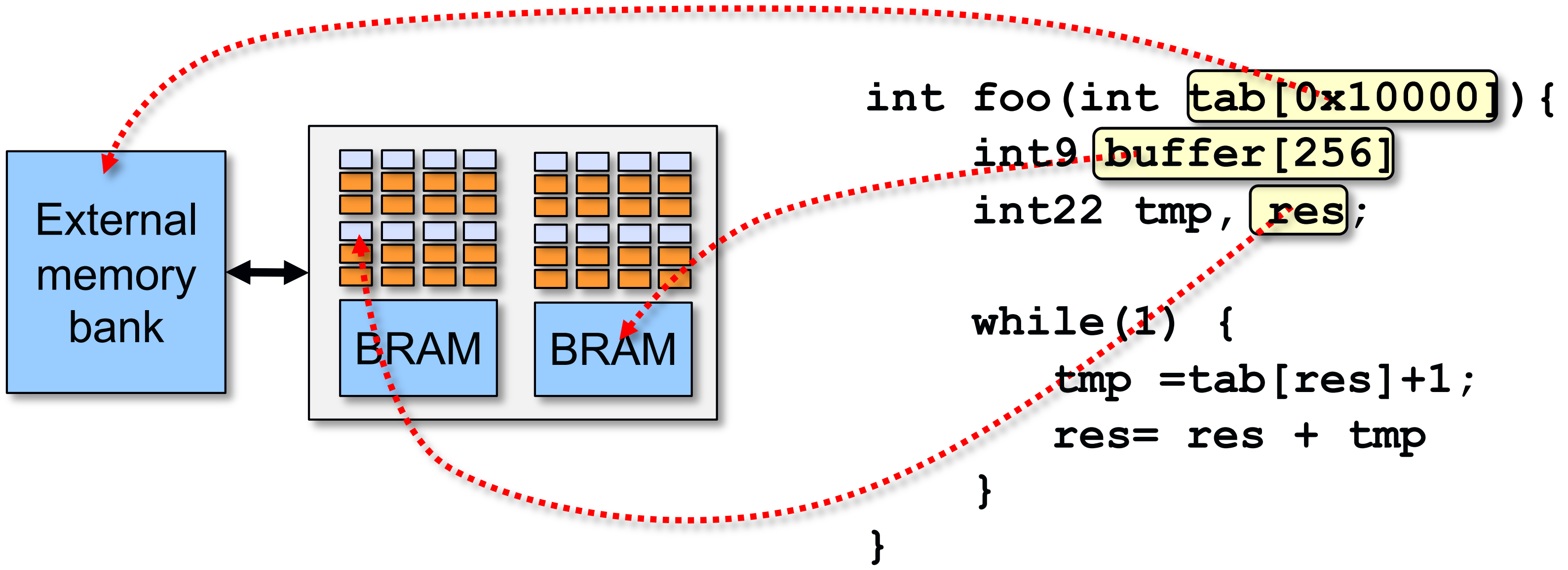
1. How to take advantage of parallelism in the C kernel
2. How to optimize memory accesses
   - **Taking advantage of memory hierarchy**
   - Managing memory bank conflicts
3. Understanding & circumventing HLS tool limitations
4. Current and future research direction in HLS

Memory hierarchy in an FPGA



Reminder from yesterday

- Mapping of program variables inferred from source code
  - Scalar variables are mapped to flip-flops or wires
  - Local arrays are stored in on-chip memory blocks
  - External arrays (arguments) are stored in external memory

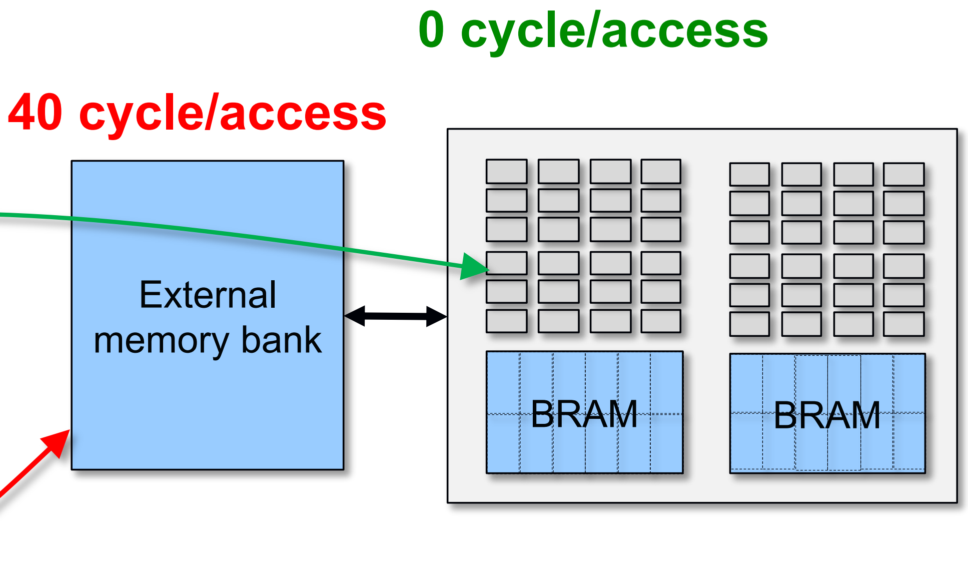


```
int foo(int tab[0x10000]){
         int9 buffer[256]
         int22 tmp, res;

         while(1) {
             tmp =tab[res]+1;
             res= res + tmp
         }
}
```

Impact of memory accesses

- Consider the kernel below, with **x**, **w**, **y**, in ext. memory

```
ac_int<6>  X[…][…];
ac_int<3>  W[…][…];
ac_int<11> y[…][…];

ac_int<11> tmp;

for(r = 0; r < ROW ; row++) {
  for(c = 0; c < COL ; col++) {
    tmp = 0;
    for(i = 0; i < 3 ; i++) {
      for(j = 0; j < 3 ; j++) {
        tmp += w[i][j] *
               x[r+i][c+j];
    } }
    y[r][c] = t;
} }
```



- Each iteration would take at best 2x40=80 cycles/iteration
  - We need to use on-chip memory to store **w[]** and x[] arrays

Using on-chip memory

- Copy data to/from off-chip from/to on-chip memory

```
void cnn_layer(int e_X[…][…], e_W[…][…], e_y[…][…]) {
  int l_X[…][…],l_W[…][…],l_y[…][…];
```

Read

(DRAM to BRAM)

```
  for(r = 0; r < ROW ; r++)
    memcpy(l_y[r] , e_y[r][c], );

  for(i = 0; i < K ; r++)
    memcpy(l_w[r] , e_w[r], K);
```

K^2+ROW*COL access

Execute

(data in BRAM)

```
  for(r = 0; r < ROW ; row++) {
    for(c = 0; c < COL ; col++) {
      tmp = 0;
      for(i = 0; i < 3 ; i++)
        for(j = 0; j < 3 ; j++)
          tmp += l_w[i][j]*l_x[r+i][c+j];
      l_y[r][c] = tmp;
  } }
```

Write

(BRAM to DRAM)

```
  for(r = 0; r < ROW ; r++)
      memcpy(e_y[r],l_y[r], COL);
```

ROW*COL access

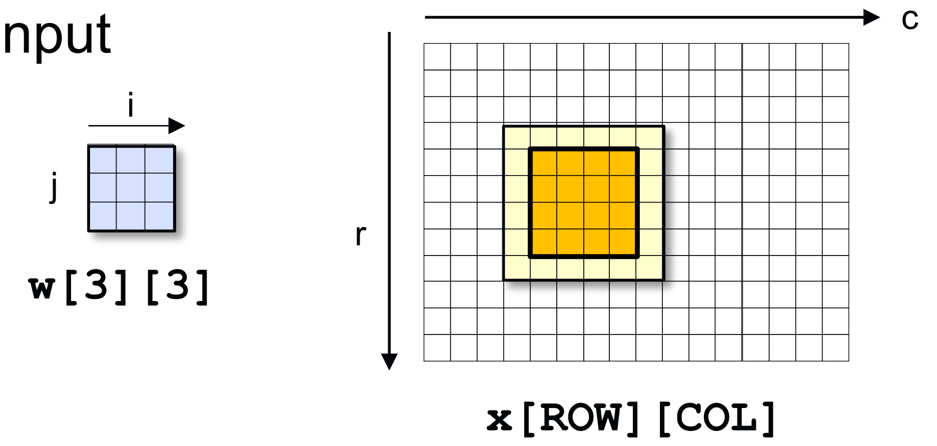
Tiling/blocking computations

- We may not want to copy whole arrays in local memory
  - We can *tile* computations so as to use a subset of data

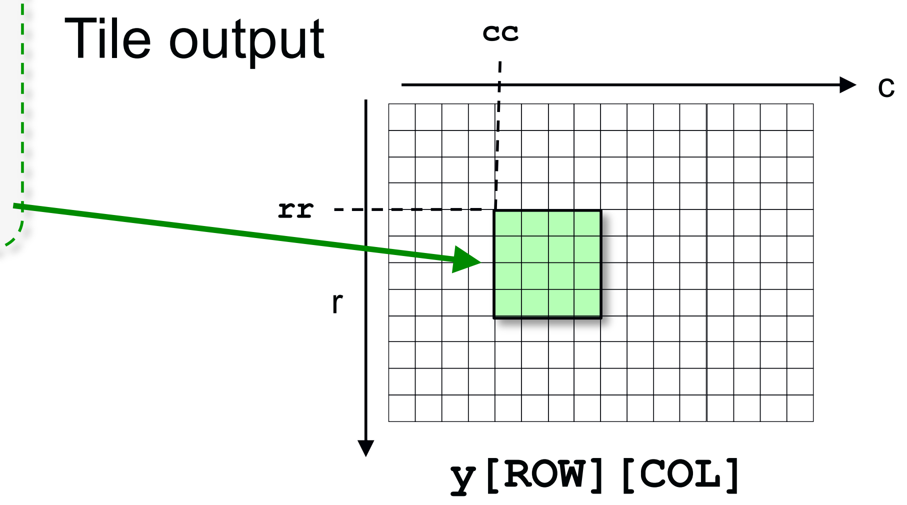
```
for(rr = 0; rr < ROW ; rr+=4)
  for(cc = 0; cc < COL ; cc+=4)
   for(r = rr; r < rr+4 ; r++) {
     for(c = cc; c < cc+ 4; c++) {
       tmp = 0 ;
       for(i = 0; i < 3 ; i++) {
         for(j = 0; j < 3 ; j++) {
           tmp += w[i][j]*x[r+i][c+j];
       } }
     y[r][c] = tmp;
} }
```

Tile of size 4x4

Tile input



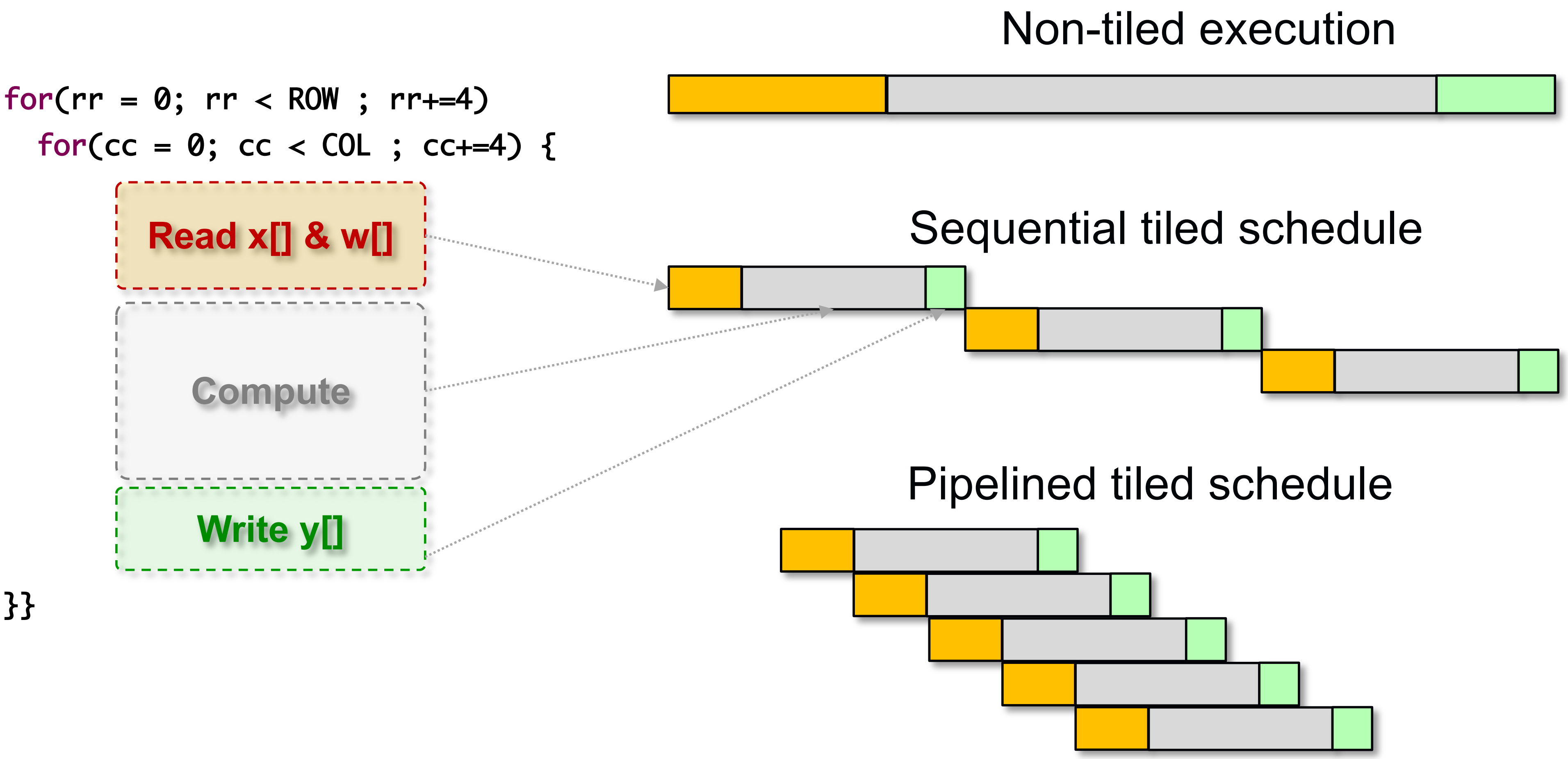
Tile output



Tile computation/comm overlap

- After Tiling + data motion code
  - We can overlap the execution of tiles (macro-pipeline)

```
for(rr = 0; rr < ROW ; rr+=4)
  for(cc = 0; cc < COL ; cc+=4) {
```



```
}}
```

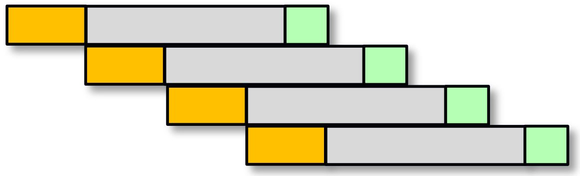
Read Execute Write macro pipeline

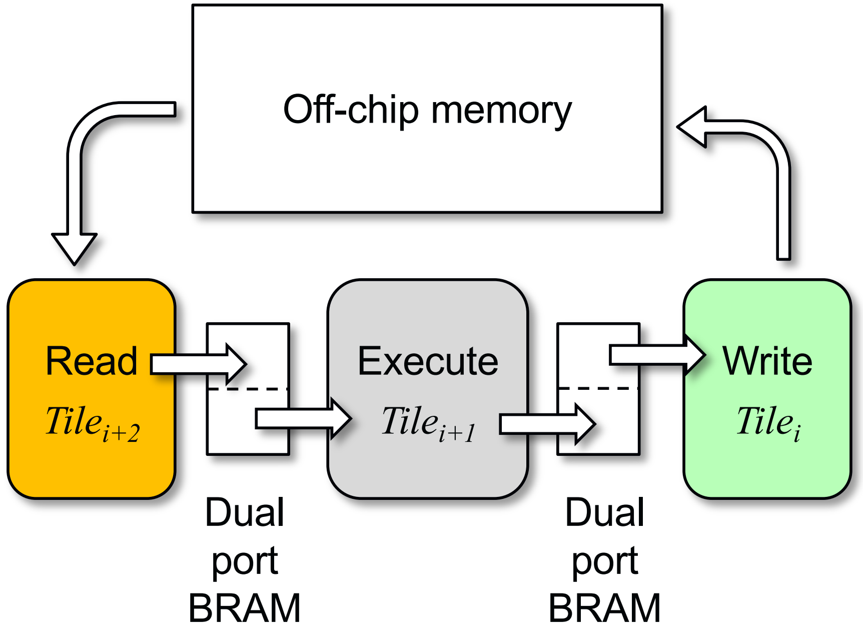
- Requires task level parallelism

Example of Vitis HLS

```
for(rr = 0; rr < ROW ; rr+=4)
  for(cc = 0; cc < COL ; cc+=4)
   #pragma HLS DATAFLOW
   {
    read_x_w(rr,cc,l_x,l_w,e_x,e_w)
    execute(rr,cc,l_x,l_w)
    write_y(rr,cc,l_y,e_y)
   }
}}
```

Pipelined tiles schedule



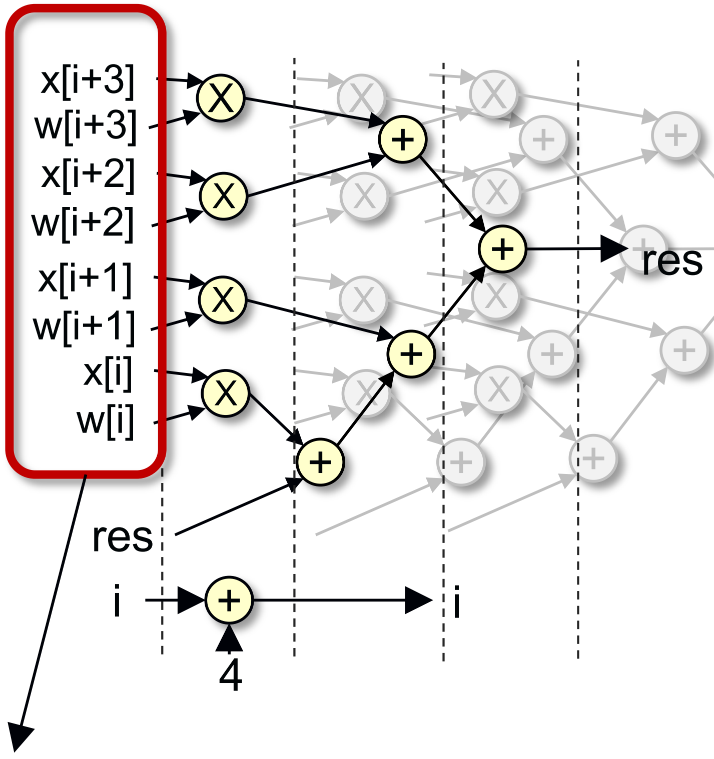


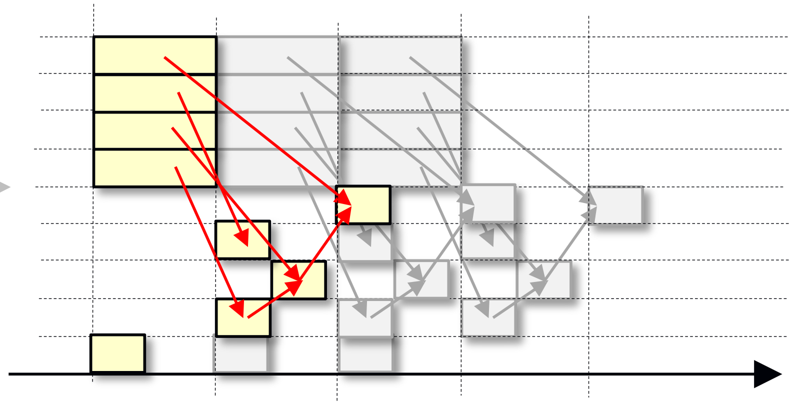
Key things to understand

1. How to take advantage of parallelism in the C kernel
2. How to optimize memory accesses
   - Taking advantage of memory hierarchy
   - **Managing memory bank conflicts**
3. Understanding & circumventing HLS tool limitations
4. Current and future research direction in HLS

Memory port resource conflicts

- main limiting factor for reaching II=1 is often memory
  - When II=1, there are generally many concurrent accesses to a same array (and thus to a same memory block).





For II=1, we must handle 8 read per cycle to the memory blocks containing array x[] and w[]

Eliminating conflicts through scalarization

- Scalarization : copy array cell value in a scalar variable
  - Use the scalar variable instead of the array whenever possible

```
float X[128];
float res=0.0;



for (i=1;i<128;i+=1){
   res=res*(X[i]-X[i-1]);
}
```

Two memory ports needed to reach II=1

```
float X[128];
float res=0.0,tmp1,tmp0;

tmp1 = X[0];
tmp0 = X[1];
for (i=2;i<128;i+=1){
   res=res*(tmp0-tmp1);
   tmp1 =tmp0;
   tmp0 = X[i];
}
```

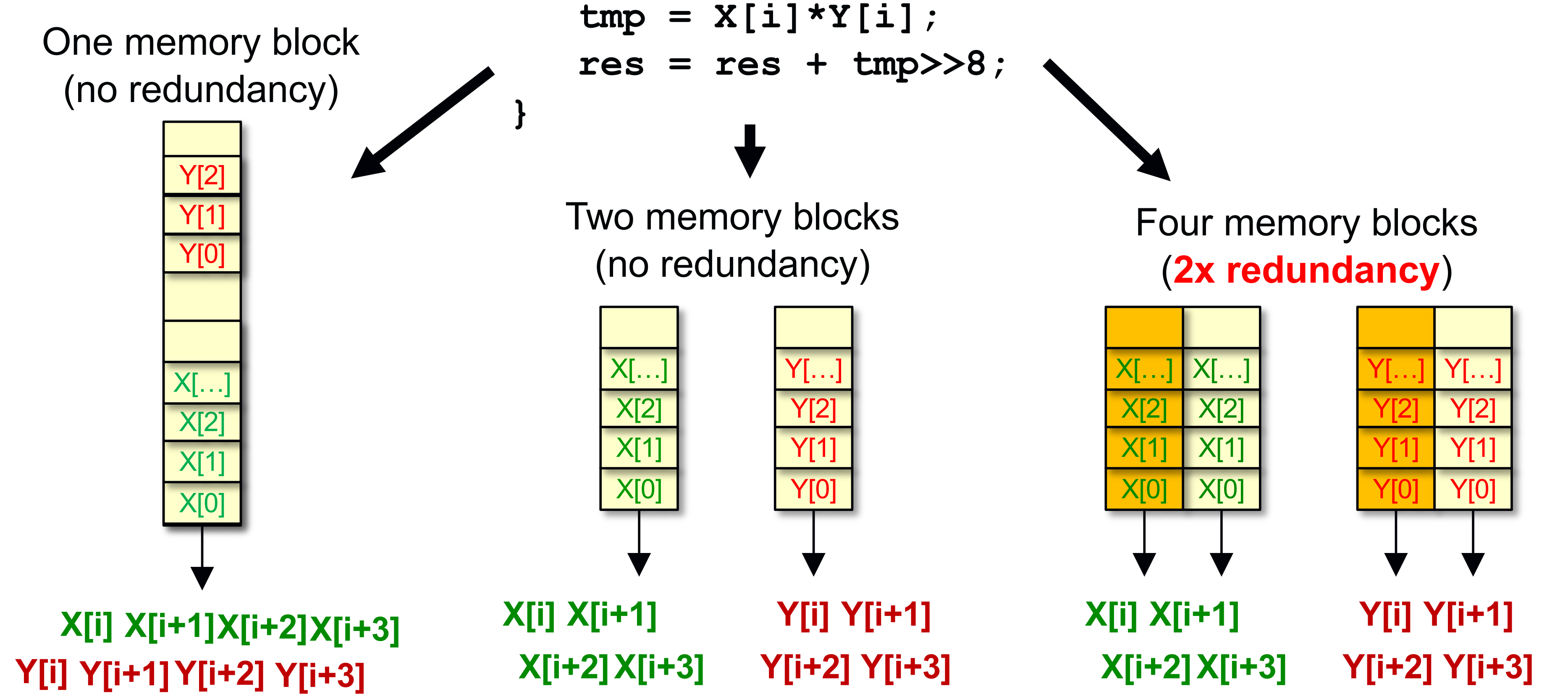
Only one memory port is needed to obtain II=1

Automatically performed by Vivado HLS since version 2016

Mapping arrays to banks

- Using multiple banks increases #access per cycle

```
int12 X[512],Y[512];

for (int9 i=0;i<512;i++){
    tmp = X[i]*Y[i];
    res = res + tmp>>8;
}
```



Up to 2 reads to **x[]** or **y[]** per clock cycle

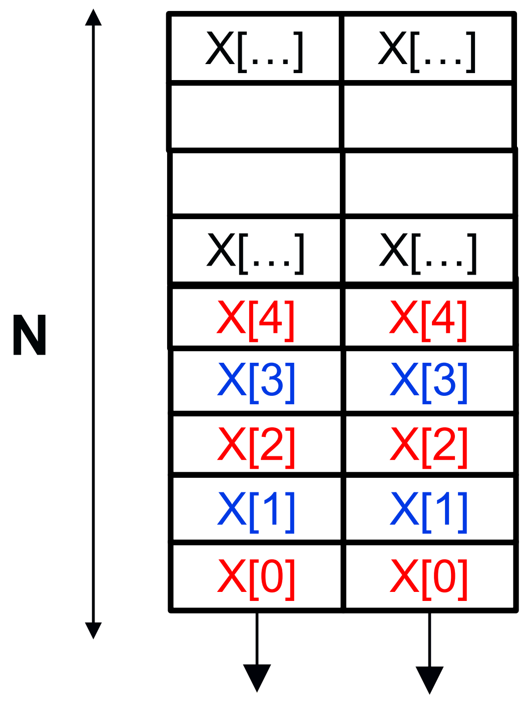
2 reads to **x[]** + 2 reads to **y[]** per clock cycle

2 reads to **x[]** + 2 reads to **x[]** per clock cycle

Partitioning arrays in banks

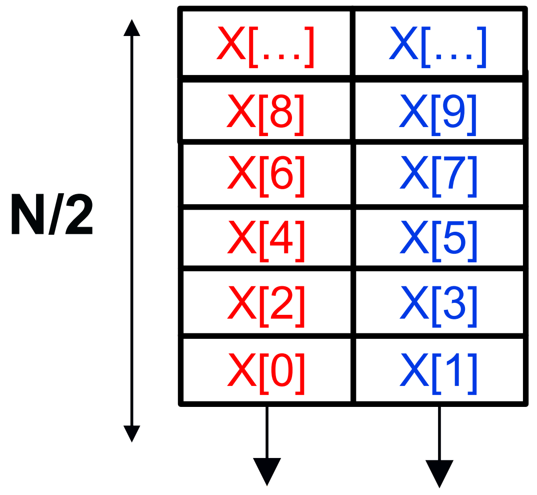
- Sometimes possible to reorganize data inside a block
  - Allow partitioning without redundancy

Four memory blocks
(**2x redundancy**)



2 reads to X + 2 reads to Y
per clock cycle

Block cyclic (cycle=2) - partitioning
(**no redundancy**)



2 reads to X + 2 reads to Y
per clock cycle

Partitioning arrays in memory banks

- Partitioning can be done in two different ways
  1. By hand at the source level (tedious but reliable)
  2. Using directives (may not always work as expected)

```
float X[32],Y[32];

for (int i=0;i<8;i+=2){
    res += X[i]*Y[i];
    res += X[i+1]*Y[i+1];
}
```

1

```
float X0[16],X1[16], Y0[16],Y1[16];

for (int i=0;i<8;i+=4){
    res += X0[i]*Y0[i]+ X1[i]*Y1[i];
}
```

2

```
#pragma HLS ARRAY_PARTITION var=X \
                type=cycle factor=2 dim=1
#pragma HLS ARRAY_PARTITION var=Y \
                type=cycle factor=2 dim=1
float X[32],Y[32];

for (int i=0;i<8;i+=4){
    res += X[i]*Y[i]+X[i+1]*Y[i+1];
}
```

Key things to understand

1. How to take advantage of parallelism in the C kernel
2. How to optimize memory accesses
3. Understanding & circumventing HLS tool limitations
   - Static Dependency & Alias Analysis
   - Overriding compiler dependency analysis
4. Current and future research direction in HLS

Dealing with dependency analysis

- HLS often « miss » parallelization opportunities
  - Compilers rely on pessimistic dependency analysis

What you understand

```
for(i=0;i<N;i++){
  for(j=i;j<M;j++){
    x[j] = foo(x[i-1]);
  }
}
```

`i<=j ⟹ i-1 < j`

Therefore all iterations in the innermost loop can run in parallel

What the compiler understands

```
for(i=0;i<N;i++){
  for(j=i;j<M;j++){
    x[j] = foo(x[dunno]);
  }
}
```

We have `dunno` ∈ $\mathbb{Z}$, therefore we cannot prove that ∄ `j` / `j=dunno`. The loop is considered as **not** parallel

User provided dependency information

- Tools support user provided hints for parallelisation
  - Overrides HLS/compiler dependency analysis results
  - To be used with care (nasty bugs are possible)

```
for(i=0;i<N;i++)
  for(j=i;j<M;j++){
    #pragma no_dep x
    x[j] = foo(x[i-1]);
  }
```

Tells the HLS tools that all iterations in the innermost loop can run in parallel

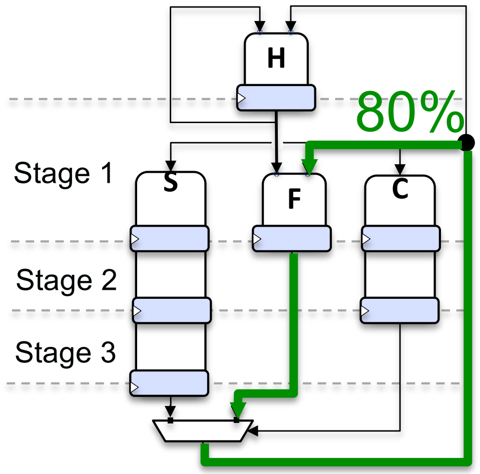
```
for(i=4;i<N;i++)
  for(j=i;j<M;j++){
    #pragma dep_distance x 4
    x[j] = foo(x[j-i]);
  }
```

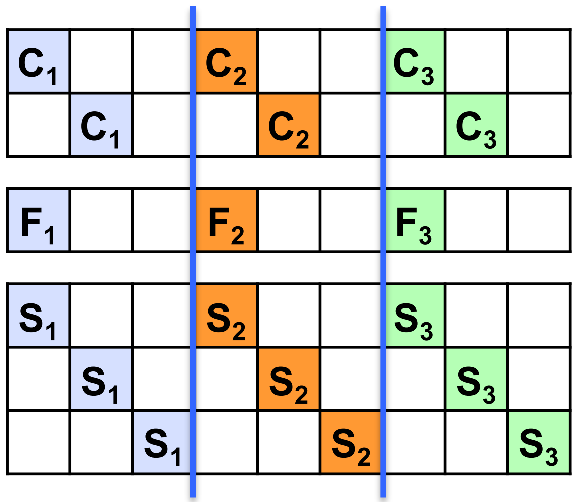
Tells the HLS tools that up to 4 consecutive iterations can be run in parallel

Limits of static loop pipelining

- Dependance distance is determined at compile-time
  - Poor support for runtime/data-dependent control-flow
  - Pipelined schedule is based on the worst-case scenario

```
do {
  tmp=x;
  if(C(tmp)) {
    // slow
    x = S(tmp);
  } else {
    // fast
    x = F(tmp,y);
  }
  y = H(tmp,y)
} while (!x)
```





Loop pipelining
(speculative code motion)
CPI=3

What is not possible (and why)

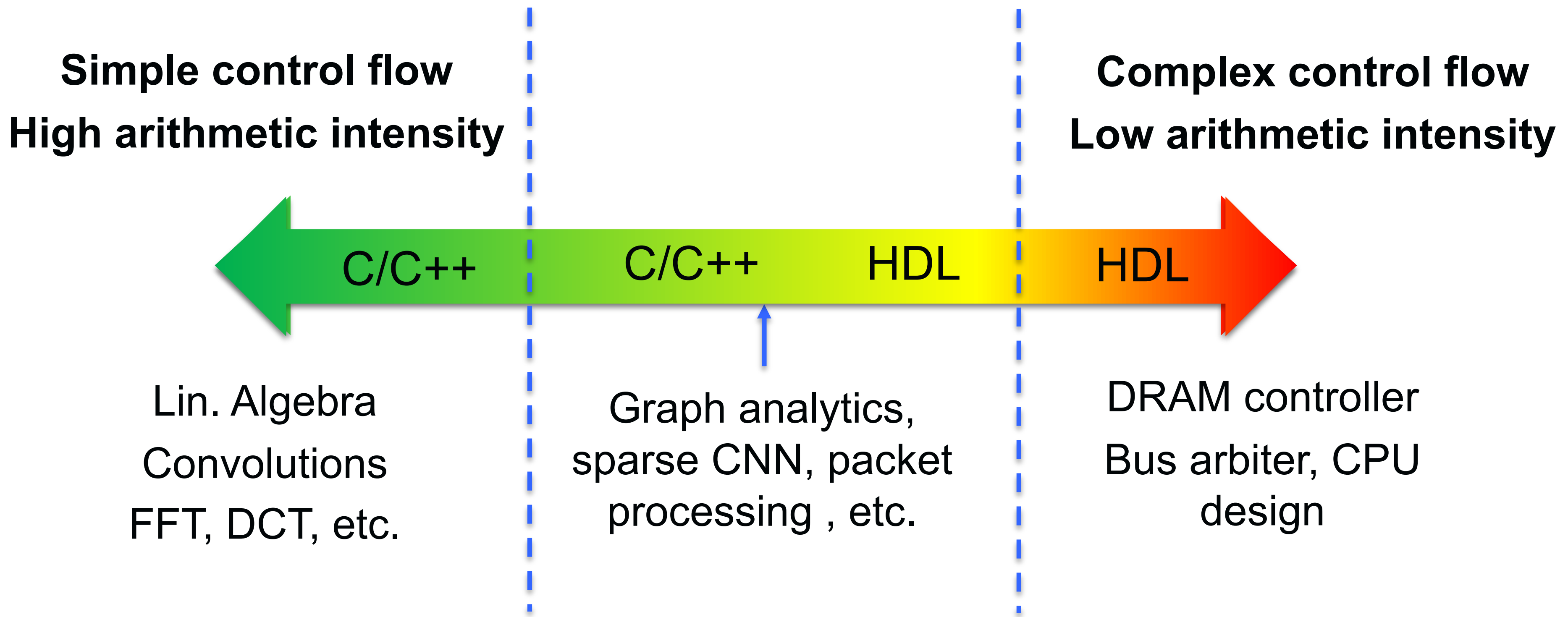
- Goto statements : they are supported by most tools
  - That’s not an excuse for using them
- Pointer arithmetic : they are supported by most tools
  - Supported as long as it does not raise aliasing issues
- Dynamic Memory Allocation (**`malloc/free`**)
  - The root of the problem lies in pointer aliasing
  - Possible if all allocated objects lie within the same bank

Key things to understand

1. How to take advantage of parallelism in the C kernel
2. How to optimize memory accesses
3. Understanding & circumventing HLS tool limitations
4. **Current and future research direction in HLS**

Toward new application domains

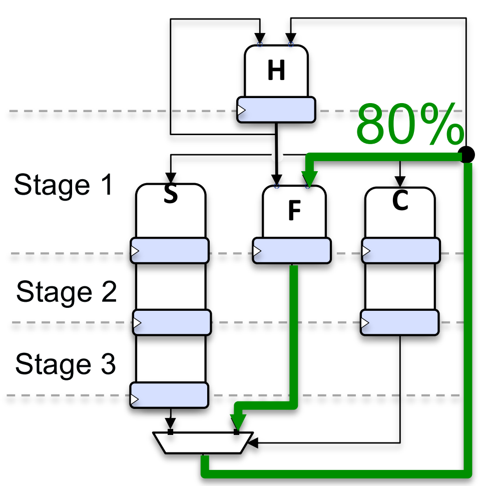
- HLS build on classical compiler optimization techniques
  - Based on compile-time knowledge of the program
  - Mainly targeting compute intensive kernels



Dynamic & speculative HLS

- Lift the restriction to statically defined schedules
  - Enable dynamic and speculative pipeline schedules

```
do {
  tmp=x;
  if(C(tmp)) {
    // slow
    x = S(tmp);
  } else {
    // fast
    x = F(tmp,y);
  }
  y = H(tmp,y)
} while (!x)
```



C_1	C_2	C_3	~~C_4~~	-	C_4	C_5	C_6	C_7
	C_1	C_2	C_3	-	-	C_4	C_5	C_6

F_1	F_2	F_3	~~F_4~~	-	F_4	F_5	F_6	F_7

S_1	S_2	S_3	~~S_4~~	-	S_4	S_5	S_6	S_7
	S_1	S_2	S_3	-	-	S_4	S_5	S_6
		S_1	S_2	S_3	-	-	S_4	S_5

This work

II=1.6

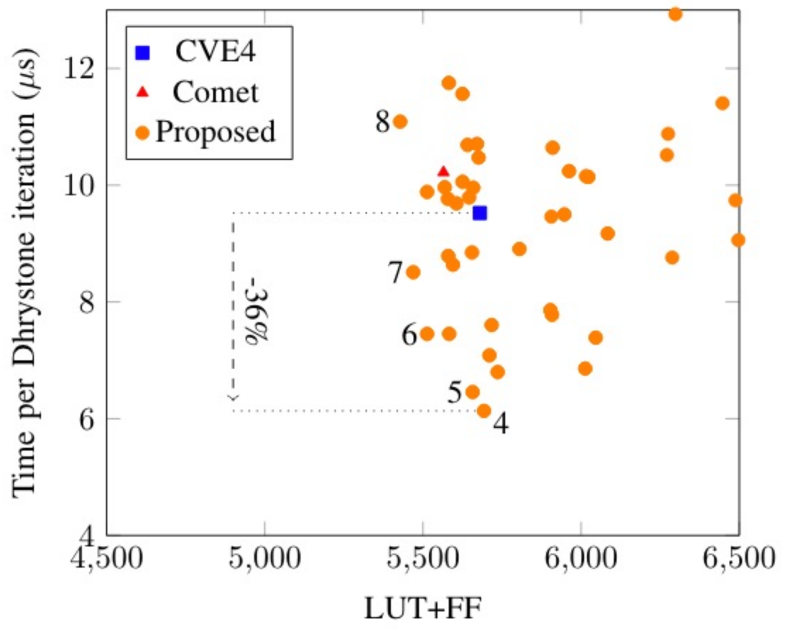
CPI=3

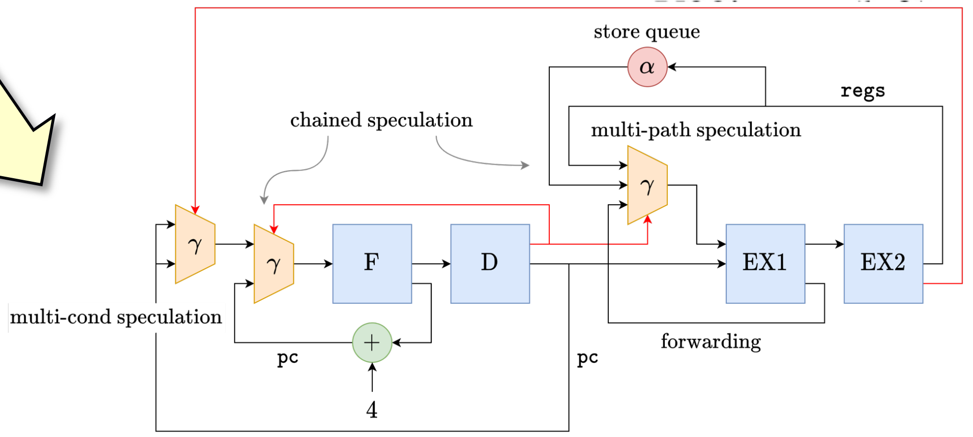
- Many open research challenges!
  - Overhead, correctness, exploration, etc.

Synthesizing more complex arch

- Example : synthesizing RISC-V CPU cores from C

```
while(1) {
 IR =  mem[PC], nxtPC = PC + 4;
 a = rs1(IR), b = rs2(IR), d = rd(IR);
 switch (op(IR)) {
  case ADD: X[d] = ALU(op(IR),X[a],X[b]);
   break;
  case MUL: X[d] = X[a]*X[b];
   break;
  case MULI: X[d] = X[a]*ext32(imm(IR));
   break;
  case LDW: X[d] = Mem[X[b]+imm(IR)];
   break;
  case STW: Mem[X[b]+imm(IR)] = X[a];
   break;
  case BEQ:
   nxtPC += X[a]==X[b] ? imm(IR) : 0;
   break;
  // ...
  }
  PC = nxtPC;
}
```





HLS in 2023

- More an more automated program transformations
  - More complex (semi) automatic transformations to come
- DSL framework building on top of HLS tools
  - For ML, graph processing, etc.
- Torward support for dynamic/speculative data-structures
  - Speculative execution techniques from processor design may help widen the scope of applicability of HLS to new domains
- Certified HLS (with coq) for HLS soon enough