UNIVERSITÉ DE RENNES 1

Programming FPGAs in C/C++ with High Level Synthesis

Outline

UNIVERSITÉ DE RENNES 1

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

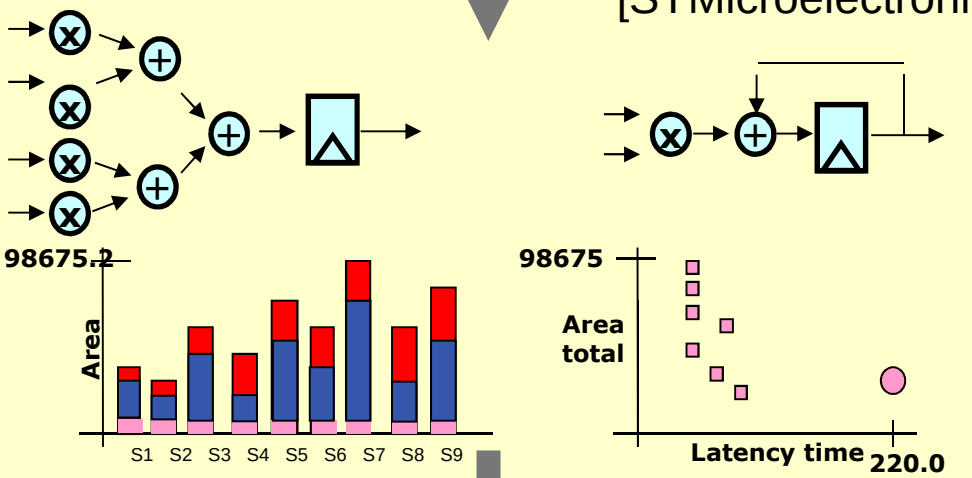
HLS : from algorithm to Silicon

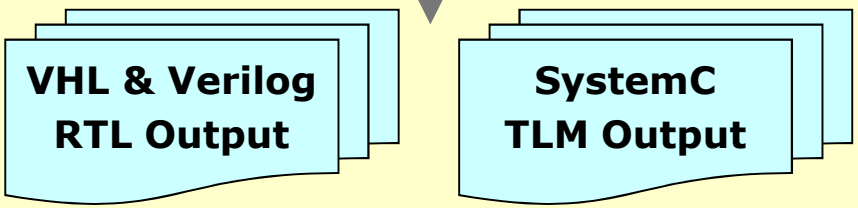
UNIVERSITÉ DE RENNES 1

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

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

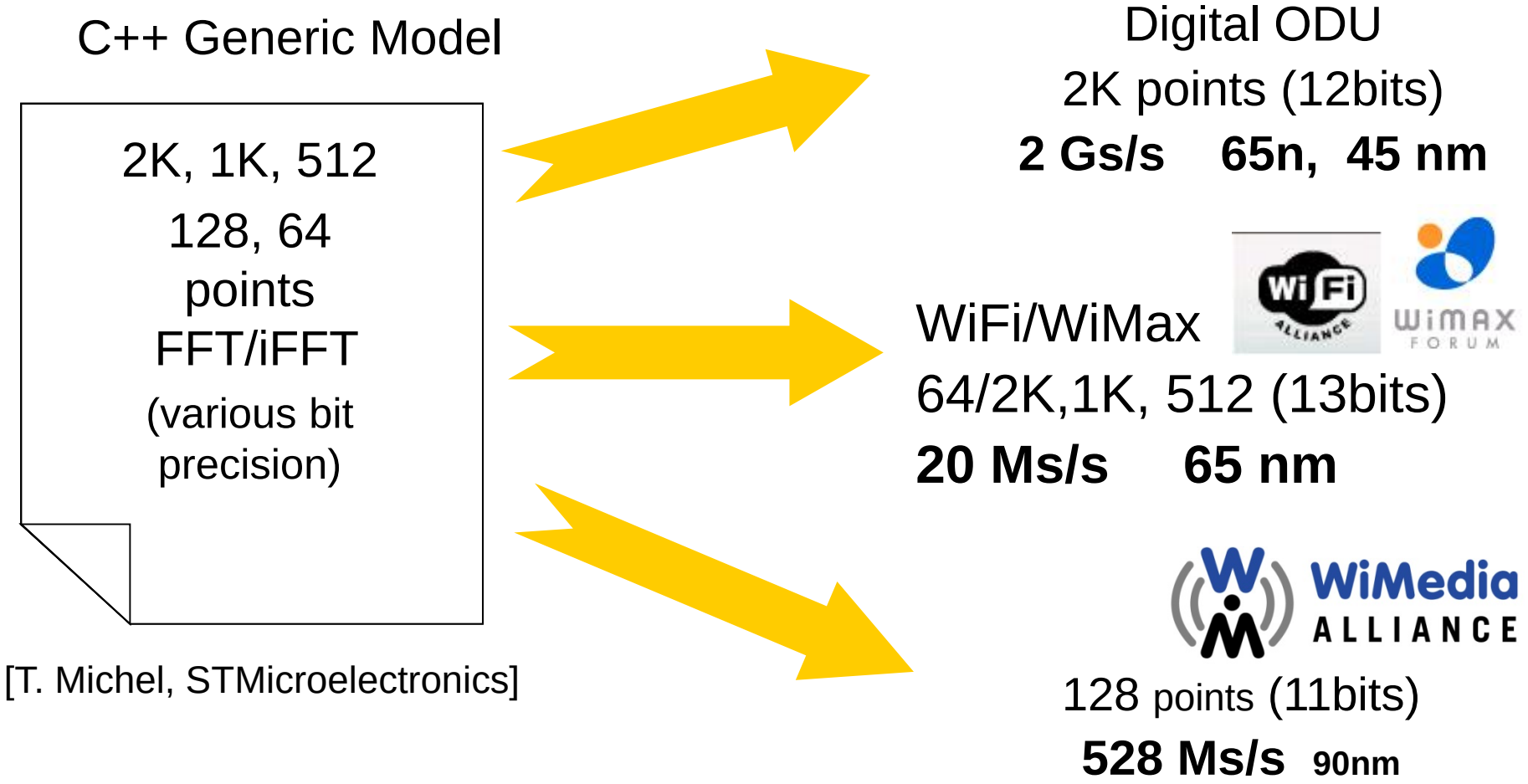
[STMicroelectronics]





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

Example: FFT at STMicroelectronics



[T. Michel, STMicroelectronics]

Better Reusability of High Level Models

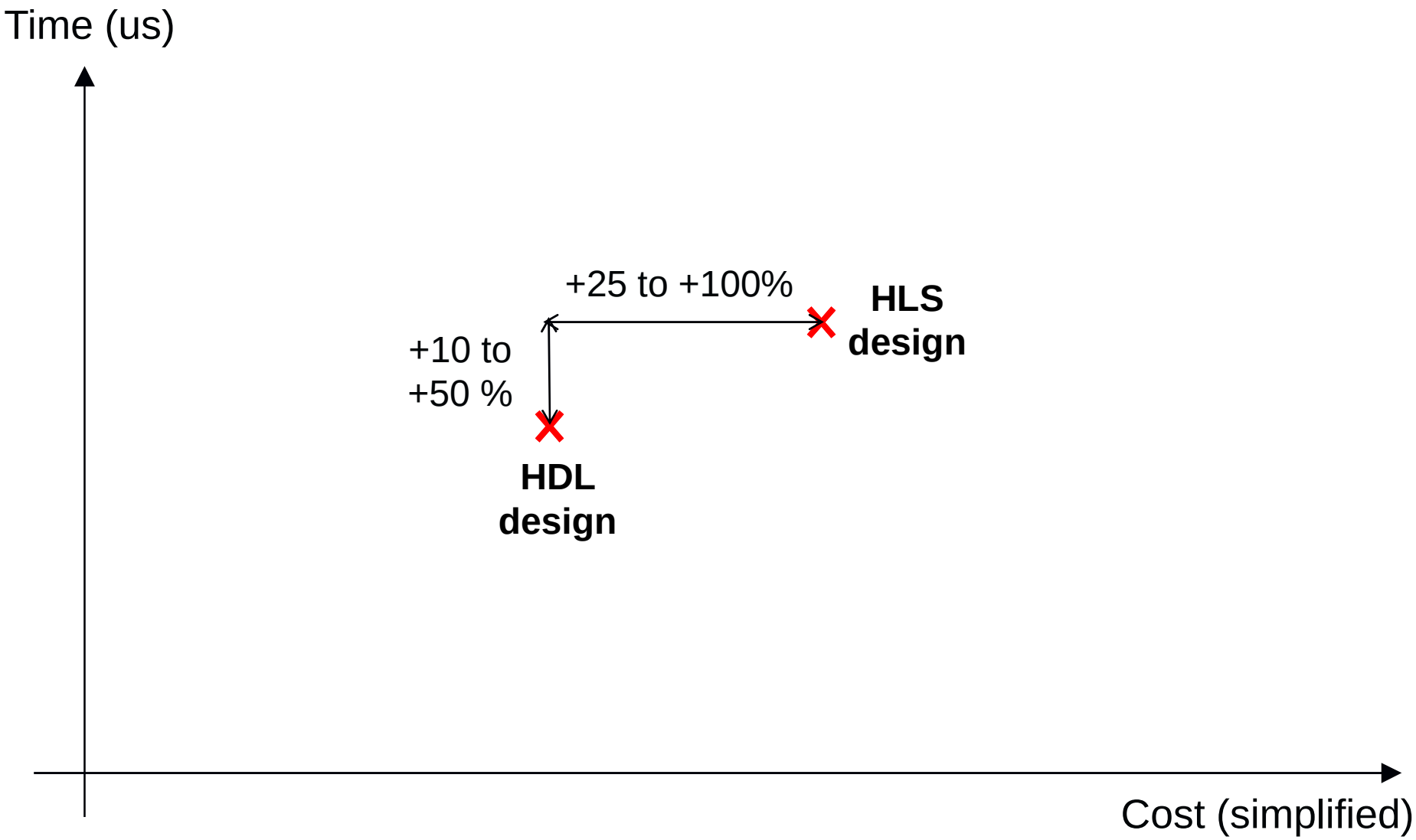
MRI-CSS

HLS : does it really work ?

UNIVERSITÉ DE RENNES 1

- High Level Synthesis is not a new idea
  - Seminal academic resarch work starting in the early 1990's
  - First commercial tools on the market around 1995
- First generation of HLS tools was a complete disaster
  - Worked only on small toy examples
  - QoR (area/power) way below expectations
  - Blatantly overhyped by marketing folks
- It took 15 years to make these tools actually work
  - Now used by worldclass chip vendors (STMicro, Nxt, …)
  - Many designers still reject HLS (bad past experience)

HLS vs HDL performance



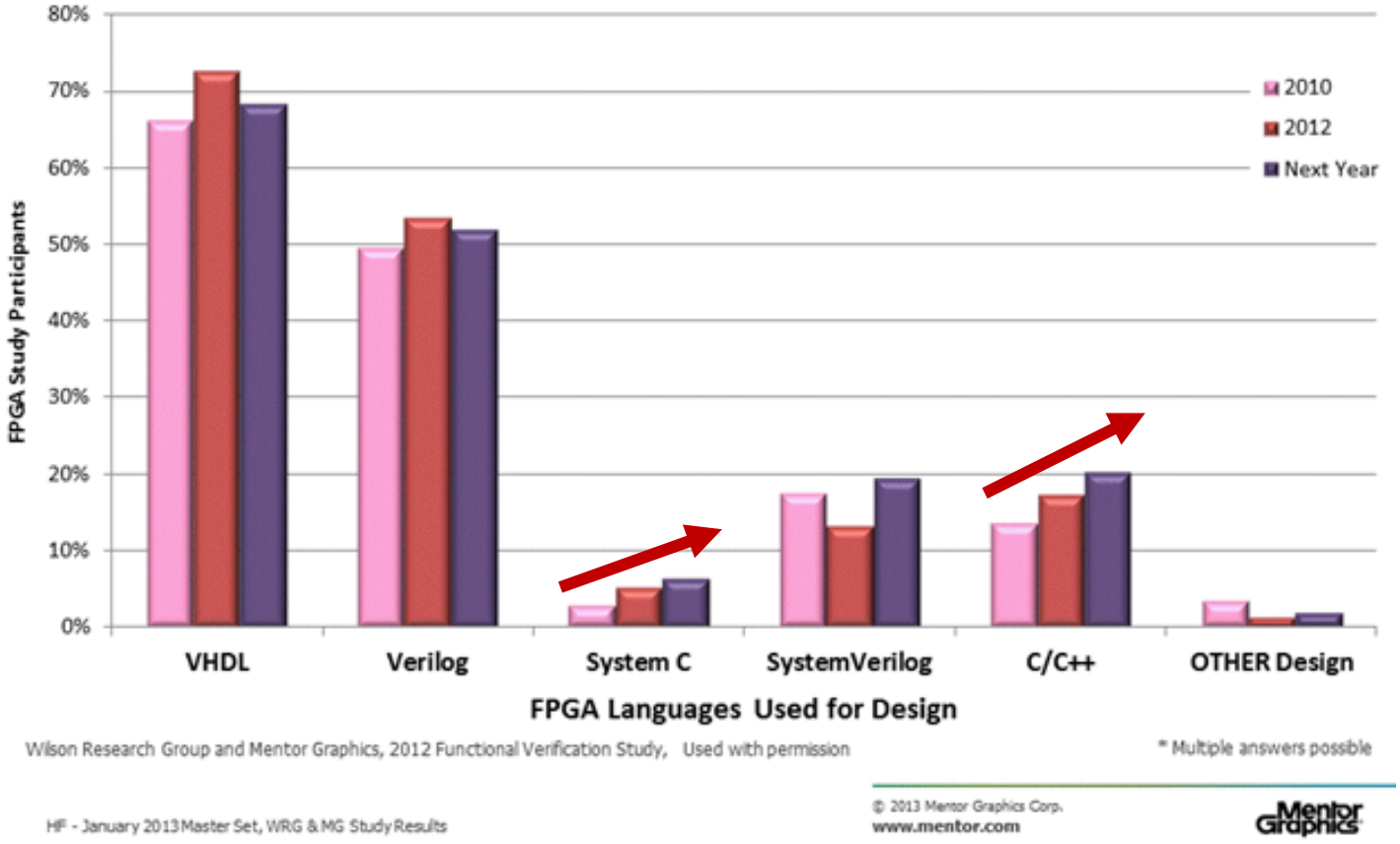
HLS in 2016

UNIVERSITÉ DE RENNES 1

- Major CAD vendors have their own HLS tools
  - **Synphony** from Synopsys (C/C++)
  - **C2S** from Cadence
  - **Catapult-C** from Mentor Graphics (C/C++/SystemC)
- FPGA vendors have also entering the game
  - **Free Vivado HLx** by Xilinx (C/C++)
  - OpenCL compiler by Altera (OpenCL)

HLS adoption

- Design languages used for FPGA design



For FPGA designs, High Level Synthesis from C/C++ will become mainstream within 5-10 years from now

Outline

UNIVERSITÉ DE RENNES 1

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

Describing digital designs in C/C++

UNIVERSITÉ DE RENNES 1

Is the C/C++ language suited for hardware design ?

- No, it follows a sequential execution model
  - Hardware is intrinsically parallel (HDLs model concurrency)
  - Deriving efficient hardware involves parallel execution
- No, it has flat/linear vision of memory
  - In an FPGA/ASIC memory is distributed (memory blocks)
- No, it supports too few datatypes (char, int, float)
  - Hardware IP uses customized wordlength (e.g 12 bits)
  - May use special arithmetic (ex: custom floating point)

How to describe circuit with C, then ?

UNIVERSITÉ DE RENNES 1

- Use classes to model circuits component and structure
  - Instanciate components and explicitely wire them together
  - Use of template can eases the design of complex circuits
  - Easy to generate a low level representation of the circuit

Still operates at the Register to Logic Level

Does not really raise the level of abstraction

No added value w.r.t HDL

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

Where is my clock ?

UNIVERSITÉ DE RENNES 1

- In C/C++, there is no 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 semantic*** ???
  - 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 …

HLS and FPGA memory model

UNIVERSITÉ DE RENNES 1

- In C/C++ memory is a large & flat address space
  - Enabling pointer arithmetic, dynamic allocation, etc.
- HLS has strong restrictions on memory management
  - No dynamic memory allocation (no malloc, no recursion)
  - No global variables (HLS operate at the function level)
  - At best very limited support for pointer arithmetics
- Mapping of program variables infered from source code
  - Scalar variables are stored in distributed registers,
  - Arrays are stored in memory blocks or in external memory
  - Arrays are often partitioned in banks (capacity or #ports)

HLS bit accurate datatypes

- C/C++ standard types limit hardware design choices
  - Hardware implementations use a wide range of wordlength
  - Minimum precision to keep algorithm correctness while reducing are improving speed.
- 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]

Outline

UNIVERSITÉ DE RENNES 1

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

The right use of HLS tools

UNIVERSITÉ DE RENNES 1

- To use HLS, one **still** need to « *think in hardware* »
  - The designer specifies an high level architectural template
  - The HLS tools then infers the complete architecture

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

Designers then use that knowledge to make the tool derive the accelerator they need.

- 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

UNIVERSITÉ DE RENNES 1

1. How the HLS tool *infer* the accelerator interface (I/Os)
   - Very important issue for system level integration
2. How the HLS tool handles array/memory accesses
   - Key issue when dealing with image processing algorithms
3. How the HLS tool handles (nested) loops
   - Does it performs automatic loop pipelining/unrolling/merging ?

Outline

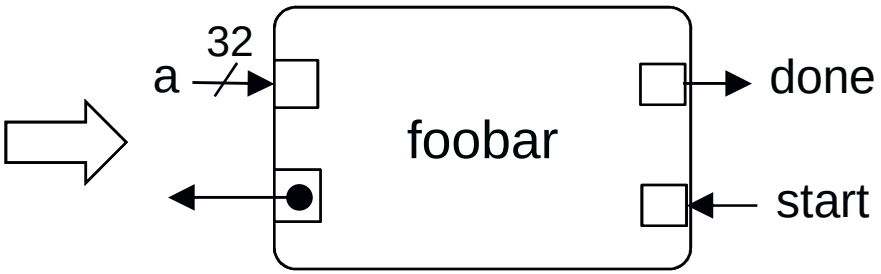
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell

High Level Synthesis from C/C++

UNIVERSITÉ DE RENNES 1

- HLS tools operate at the function/procedure level
  - Functions may call other functions (no recursion allowed)
  - Each function is mapped to a specific hardware component
  - Synthesized components are then exported as HDL IP cores
- The I/O interface infered 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){
  …
}
```

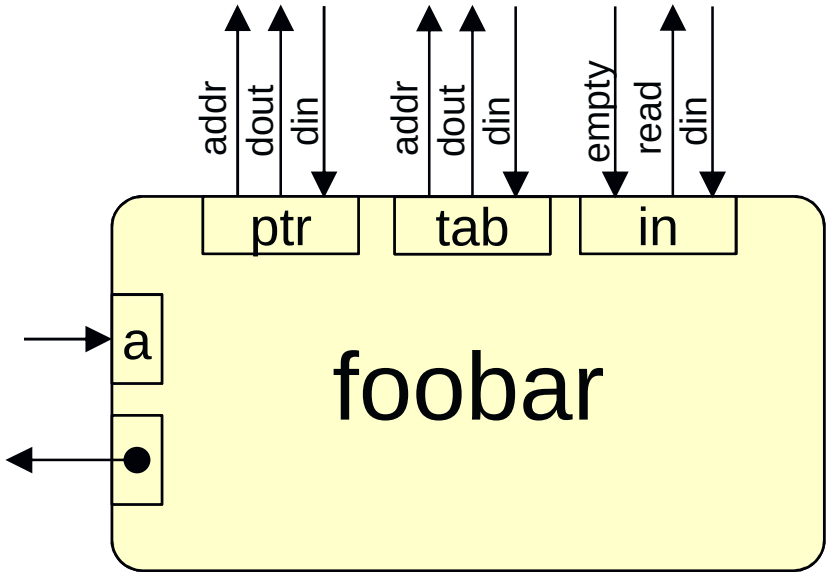


This is not enough, we may need to access memory from our hardware component …

I/O interface and system level integration

- HLS can infer more complex interfaces/protocols
  - Pointers/arrays as argument = master access on system bus
  - Streaming interface can also be infered using directives or specific templates.

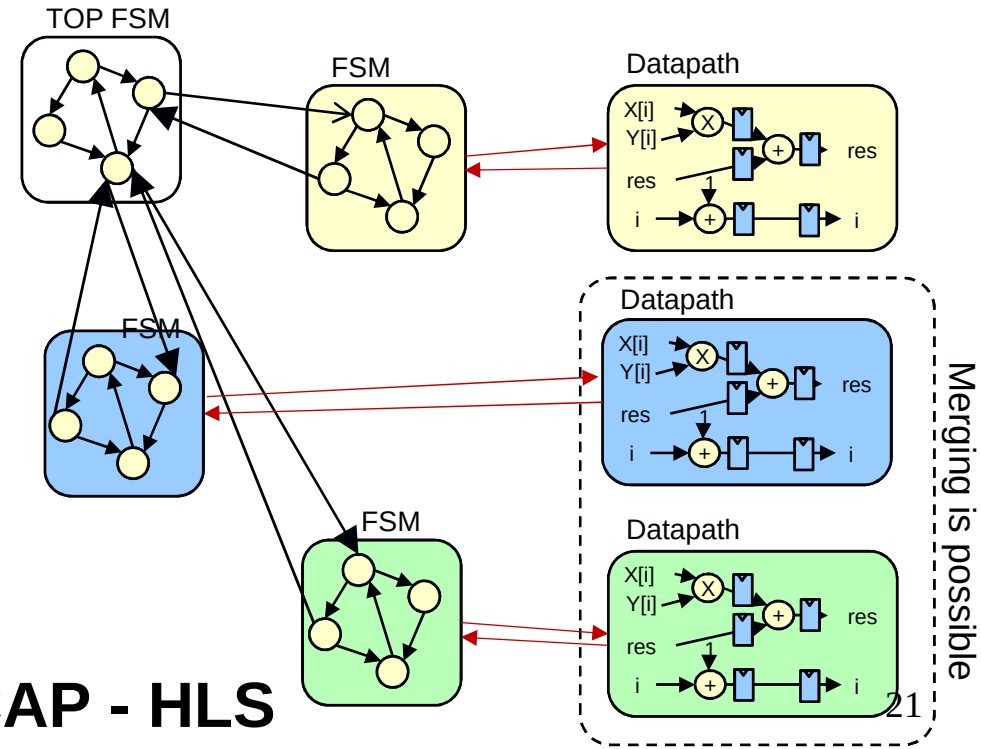
```
int foobar(
        int a,
        int* ptr,
        int tab[],
        sc_fifo<int> in){

   … = in.read()
}
```



Synthesizing hardware from a function

- HLS tools decompose the program into *blocks*
  - Each block is realized in hardware as a FSM + datapath
  - A global FSM controls which block is active according to current program state.
  - The datapath of mutually exclusive blocks can be merged to save hardware resources (very few tools have this ability)

```
for (int i=0;i<32;i++) {
   res += X[i]*Y[i];
}
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];
  }
}
```



Array access management

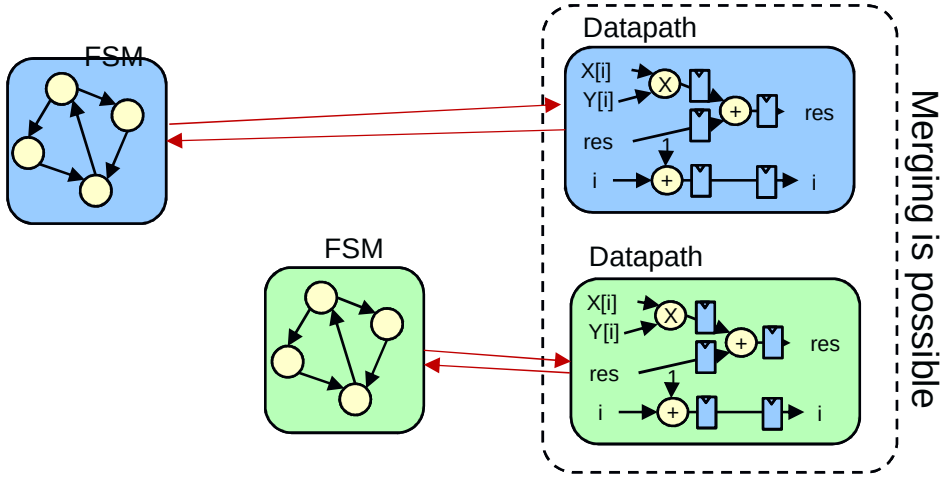
UNIVERSITÉ DE RENNES 1

- In HLS arrays are mapped to local or external memory
  - Local memory maybe local SRAM or large register files
  - External memory = bus interface or SRAM/SDRAM, etc.
- External memory access management
  - External memory access takes a indefinite number of cycles
  - Only one pending access at a time (no overlapping)
  - Burst mode accesses are sometimes supported
- Local memory management
  - Memory access always take one cycle (in read or write mode)
  - The number of read/write ports per memory can be configured
  - Memory data type can be any type supported by the HLS tool

HLS support for conditionals

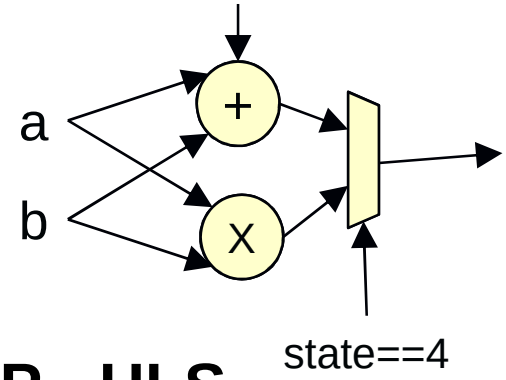
- HLS tools have two different ways of handling them
  - By considering branches as two different blocks

```
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;
}
```



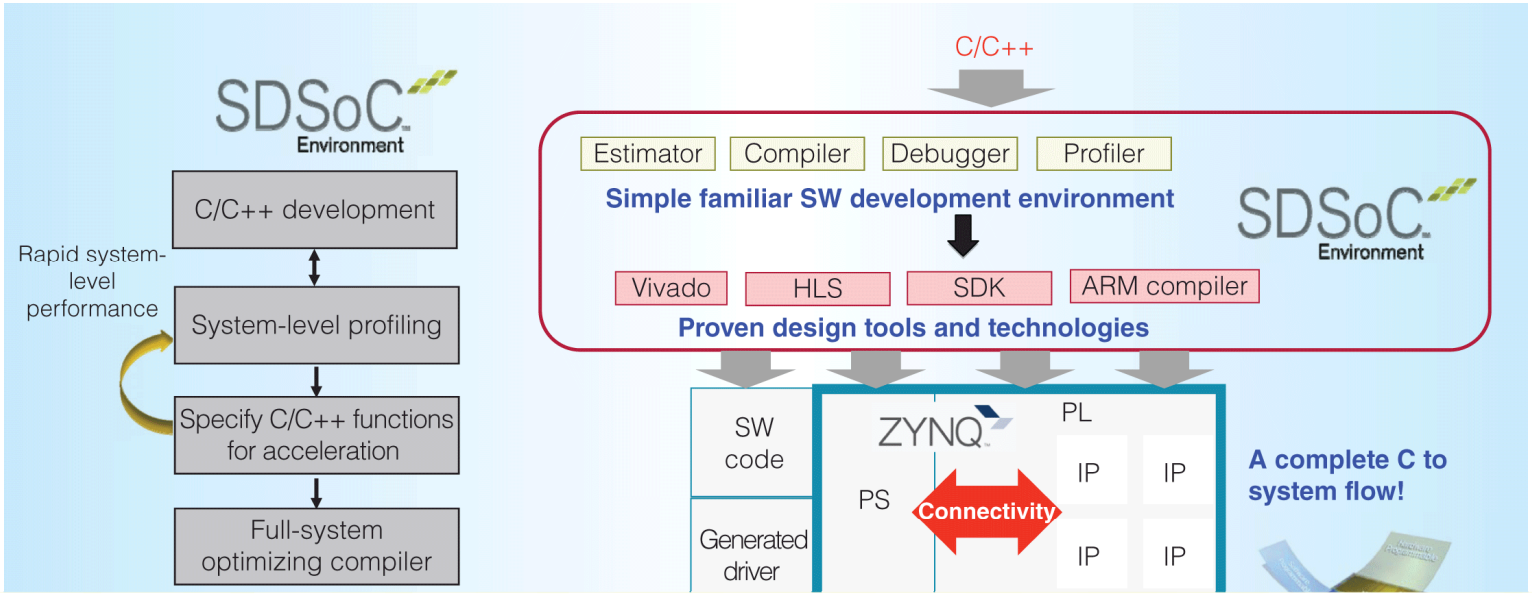
HLS support for loops

UNIVERSITÉ DE RENNES 1

- 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 : Synopsys Synphony, Mentor Catapult-C
  - Worst in class : C2S from Cadence, Impulse-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

HLS with SDSOC

- Hardware-software codesign from single C/C++ spec.
  - Key program functions annotated as FPGA accelerators
  - HLS used to synthesize accelerator hardware
  - Automatic synthesis of glue logic and device drivers



With SDSoC you can design and run an accelerator in minutes (used to take month …)

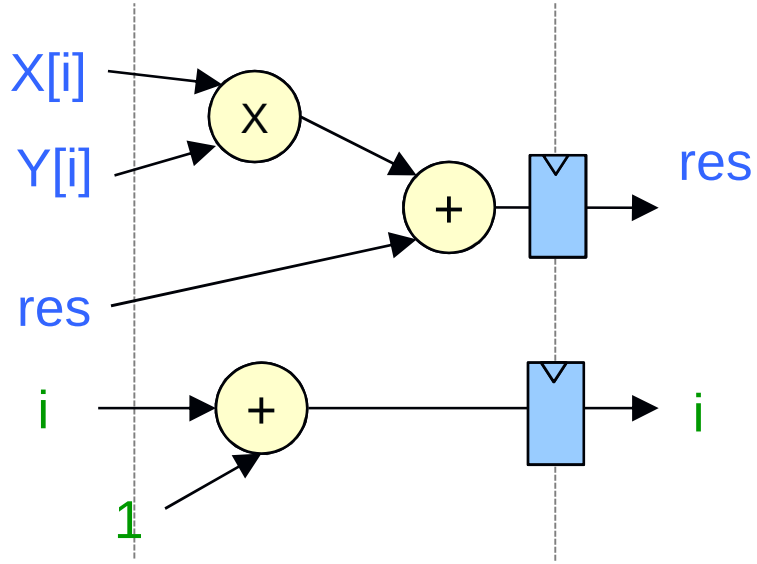
Outline

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

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

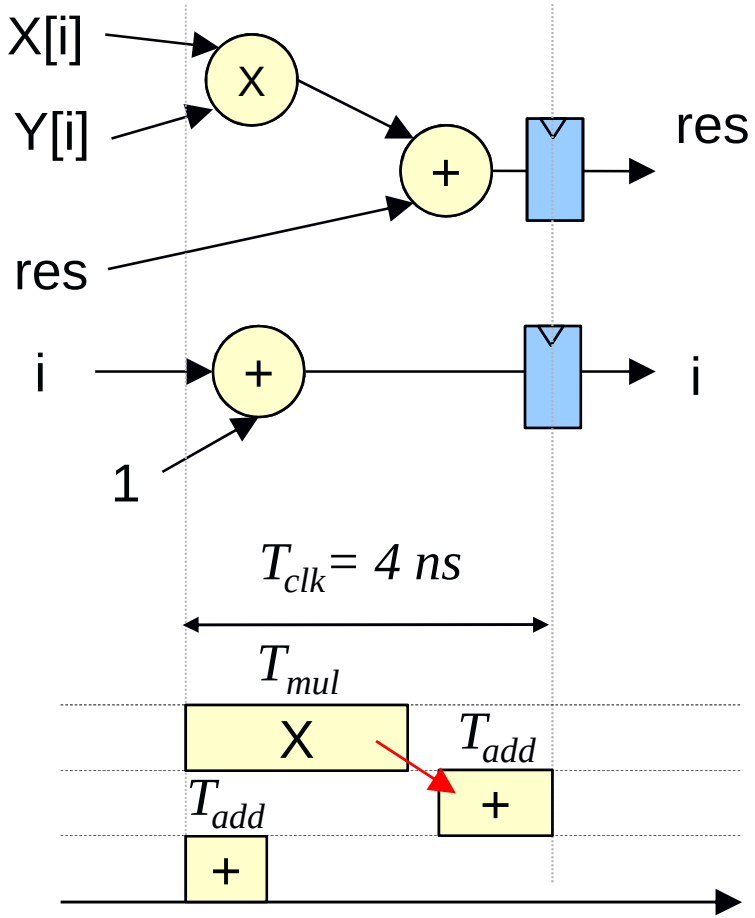
```
float X[512],Y[512];
res=0.0;
for (int i=0;i<512;i++){
   tmp = X[i]*Y[i];
   res = res + tmp;
}
```



- 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 $T_{add}=1ns$

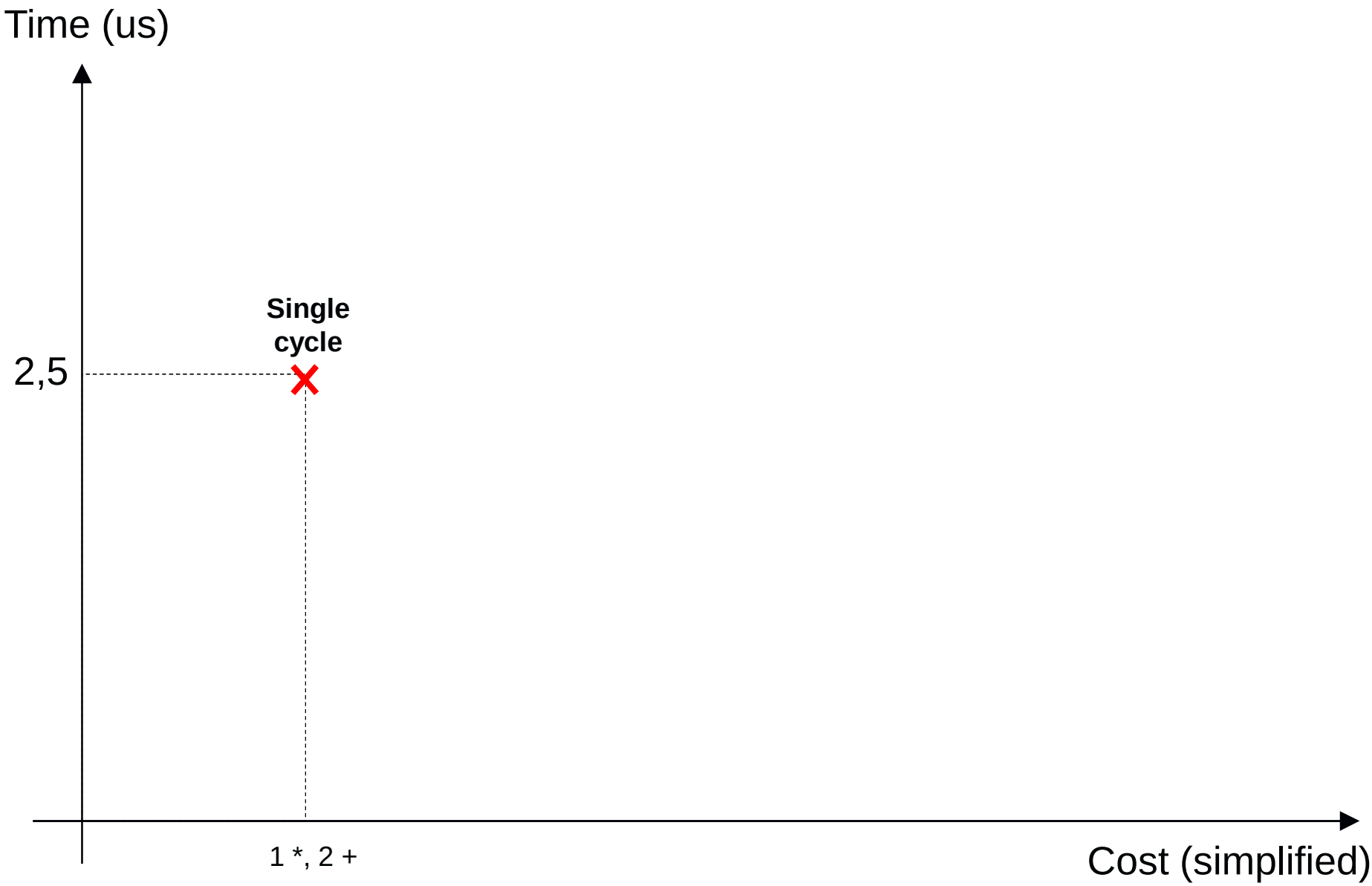


- No control on the clock speed !

```
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
512+1	250Mhz	2,5 us

Loop single-cycle execution

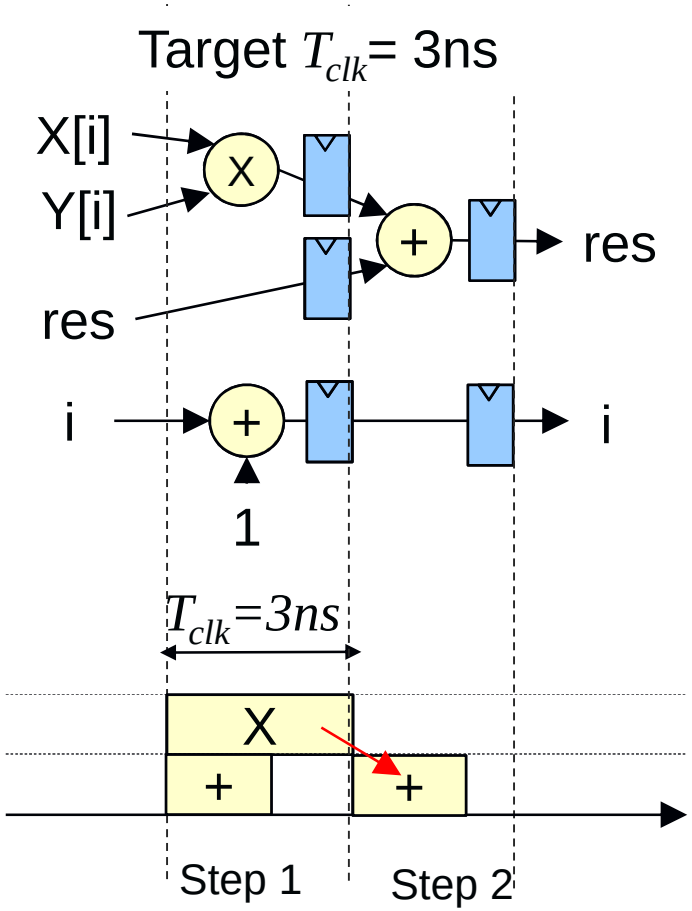


Multi-cycle execution

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

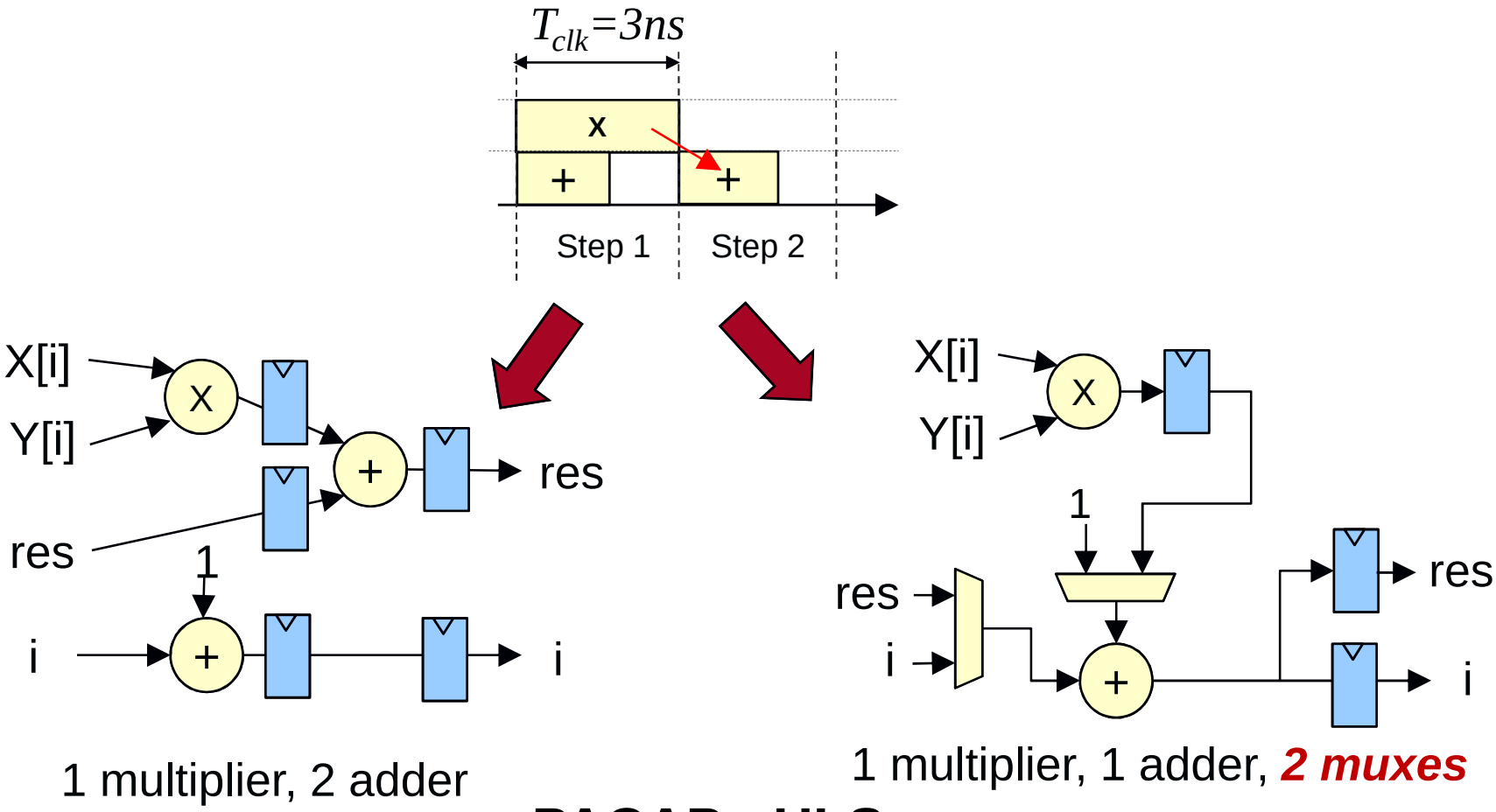
```
int X[512],Y[512];
int tmp,res=0;
for (int i=0;i<512;i++){
   tmp = X[i]*Y[i];
   res = res + tmp;
}
```

#cycles	f_{clk}	Exec time
512*2+1	330Mhz	3,07us



Hardware reuse in multi-cycle execution

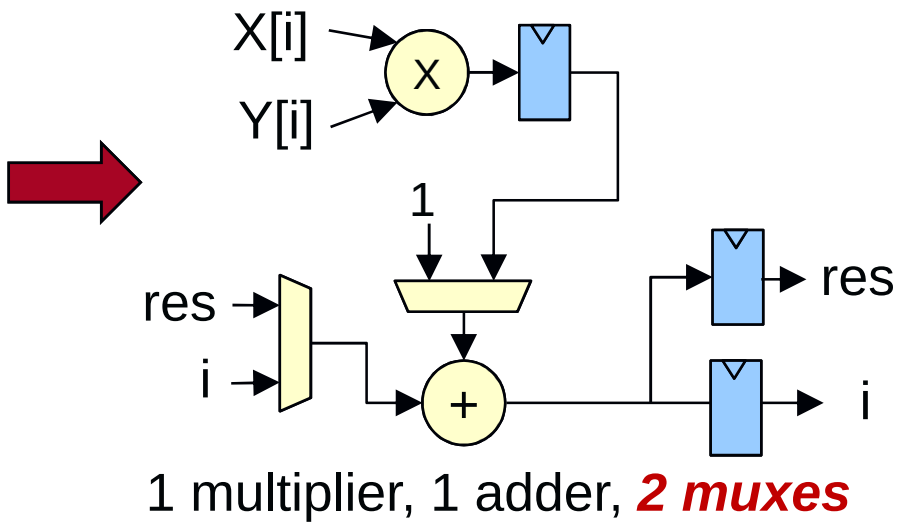
- For multi-cycle execution, HLS enable operators reuse
  - HLS tools balance hardware operator usage of over time
  - Trade-off between operator cost and muxes overhead



Hardware reuse in multi-cycle execution

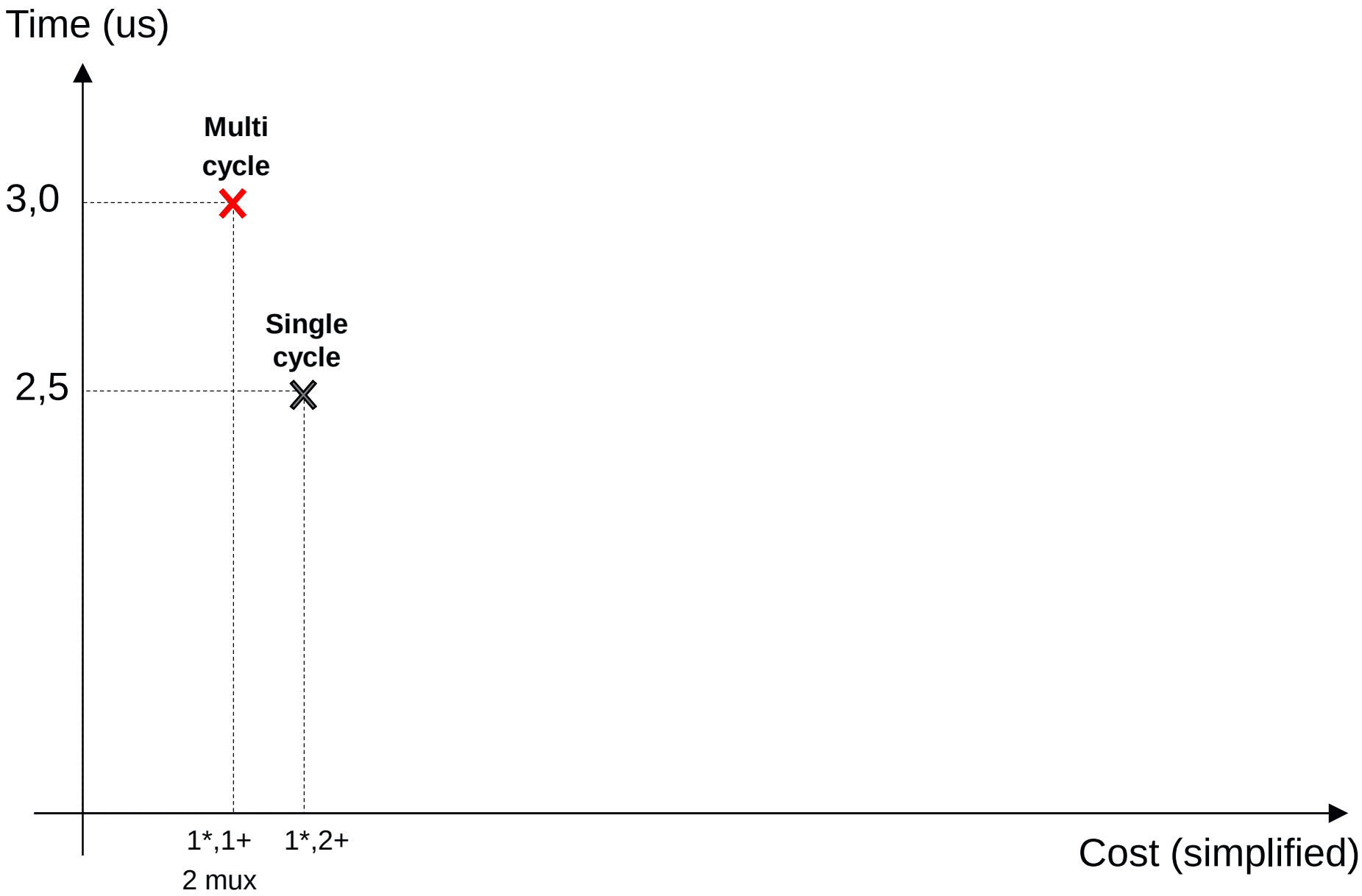
- Hardware reuse rate is controlled by the user
  - Trade-off between parallelism (performance) and reuse (cost)
  - Users provides constraints on the # and type of resources
  - Mostly used for multipliers, memory banks/ports, etc.
- The HLS tools decides what and when to reuse
  - Optimizes for area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

Loop single-cycle execution



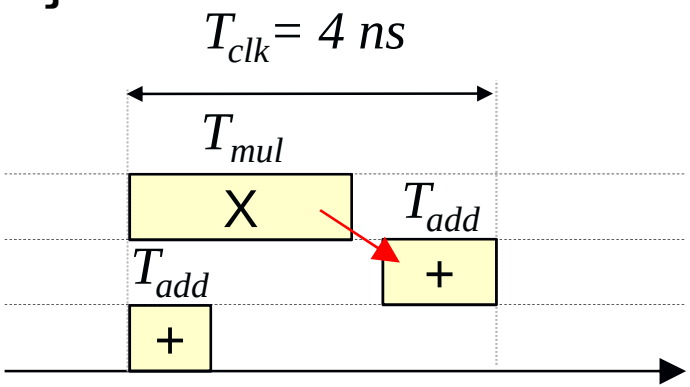
Outline

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

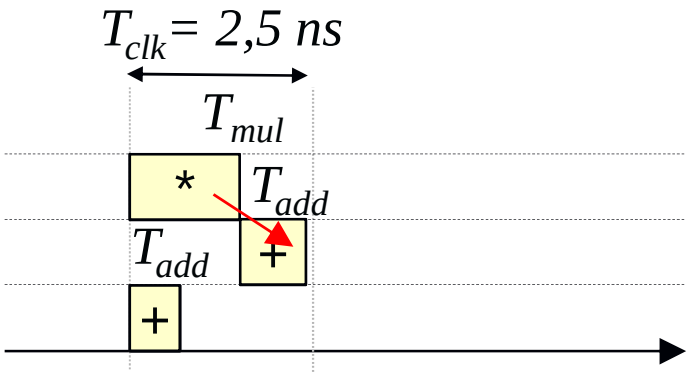
How to further improve performance ?

- 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;
}
```

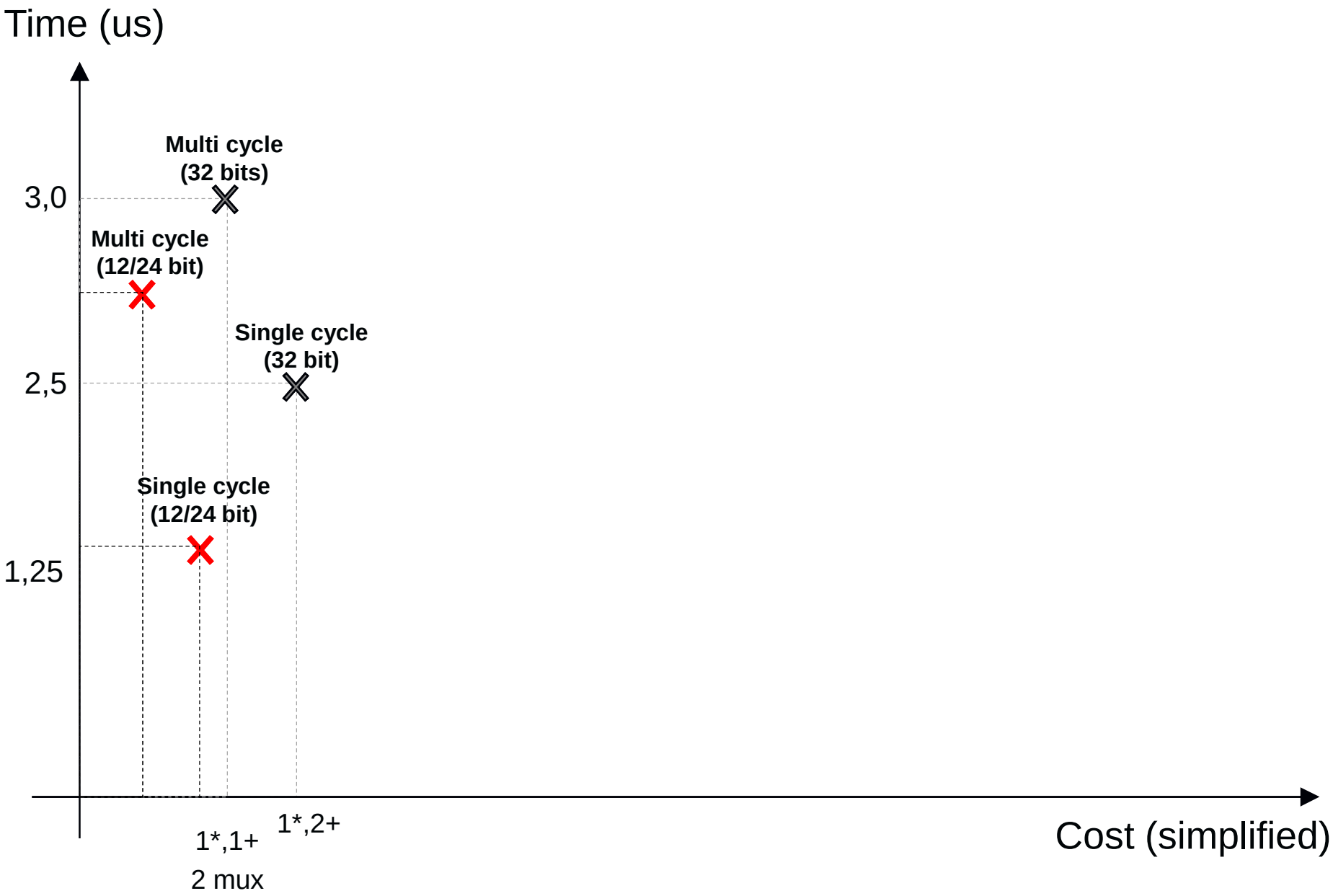



```
int12 X[512],Y[512];
int16 res=0; int24 tmp;
for (int9 i=0;i<512;i++){
   tmp = X[i]*Y[i];
   res = res + tmp>>8;
}
```



Very effective as it improves performance and reduce cost

Loop single-cycle execution



Outline

UNIVERSITÉ DE RENNES 1

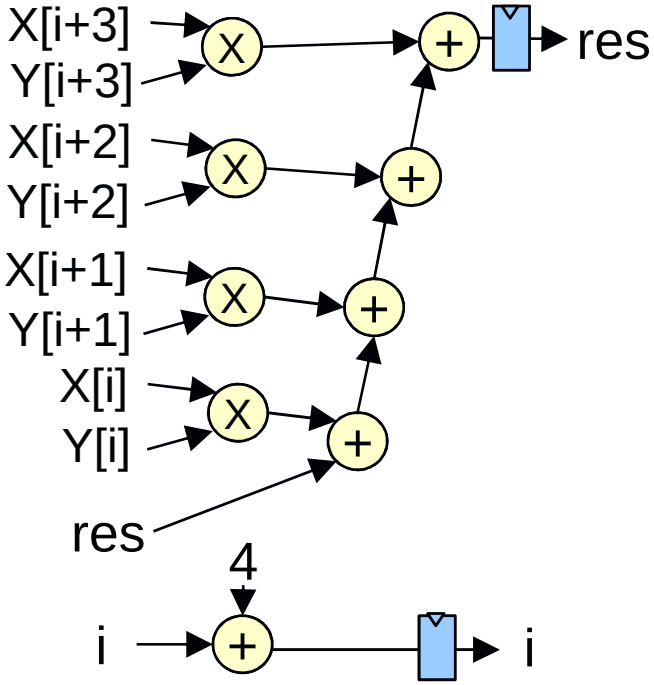
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Performance optimizations for High Level Synthesis

How to further improve performance ?

UNIVERSITÉ DE RENNES 1

- Increase the # of operations executed per iteration
  - More “work” performed within a clock cycle
  - This can be achieved with partial loop unrolling

```
int12 X[512],Y[512];
int16 res=0; int24 tmp;
for (int9 i=0;i<128;i+=4){
   tmp = X[i]*Y[i];
   res = res + tmp;
   tmp = X[i+1]*Y[i+1];
   res = res + tmp;
   tmp = X[i+2]*Y[i+2];
   res = res + tmp;
   tmp = X[i+3]*Y[i+3];
   res = res + tmp;
}
```

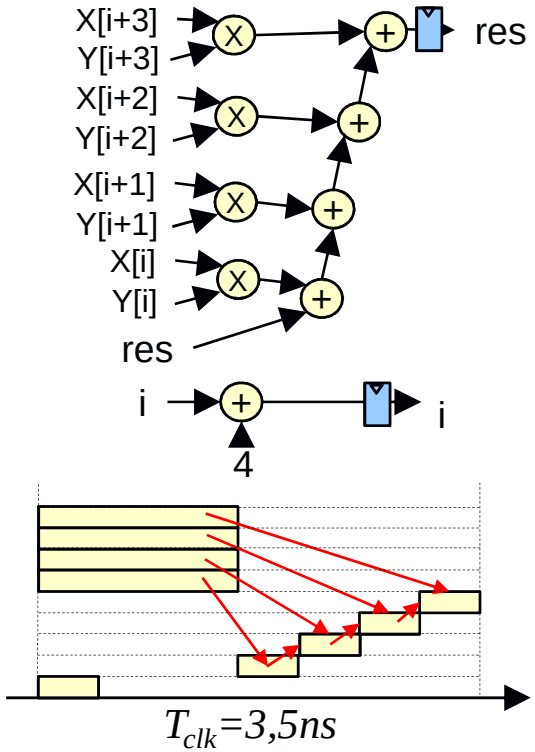


- This may also increase the datapath critical path

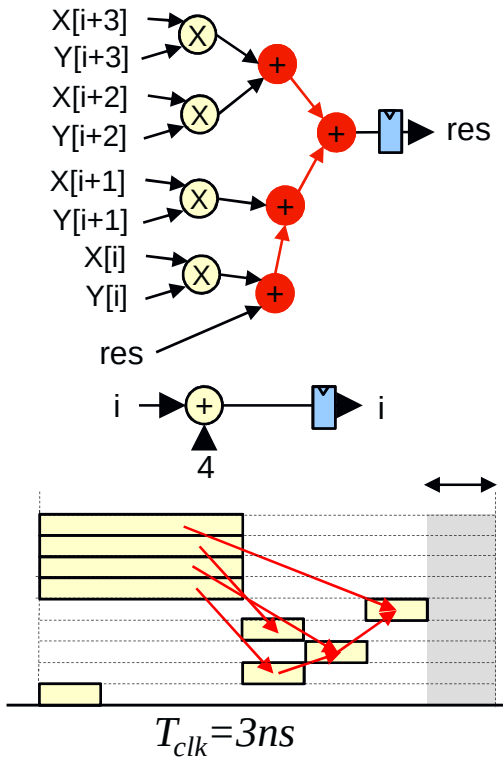
  .. and end-up being counter productive …

Loop unrolling (partial/full)

- But, the unrolled datapath can also be optimized
  - Cascaded integer adder are reorganized as adder trees
  - Some array accesses can be removed (replaced by scalars)

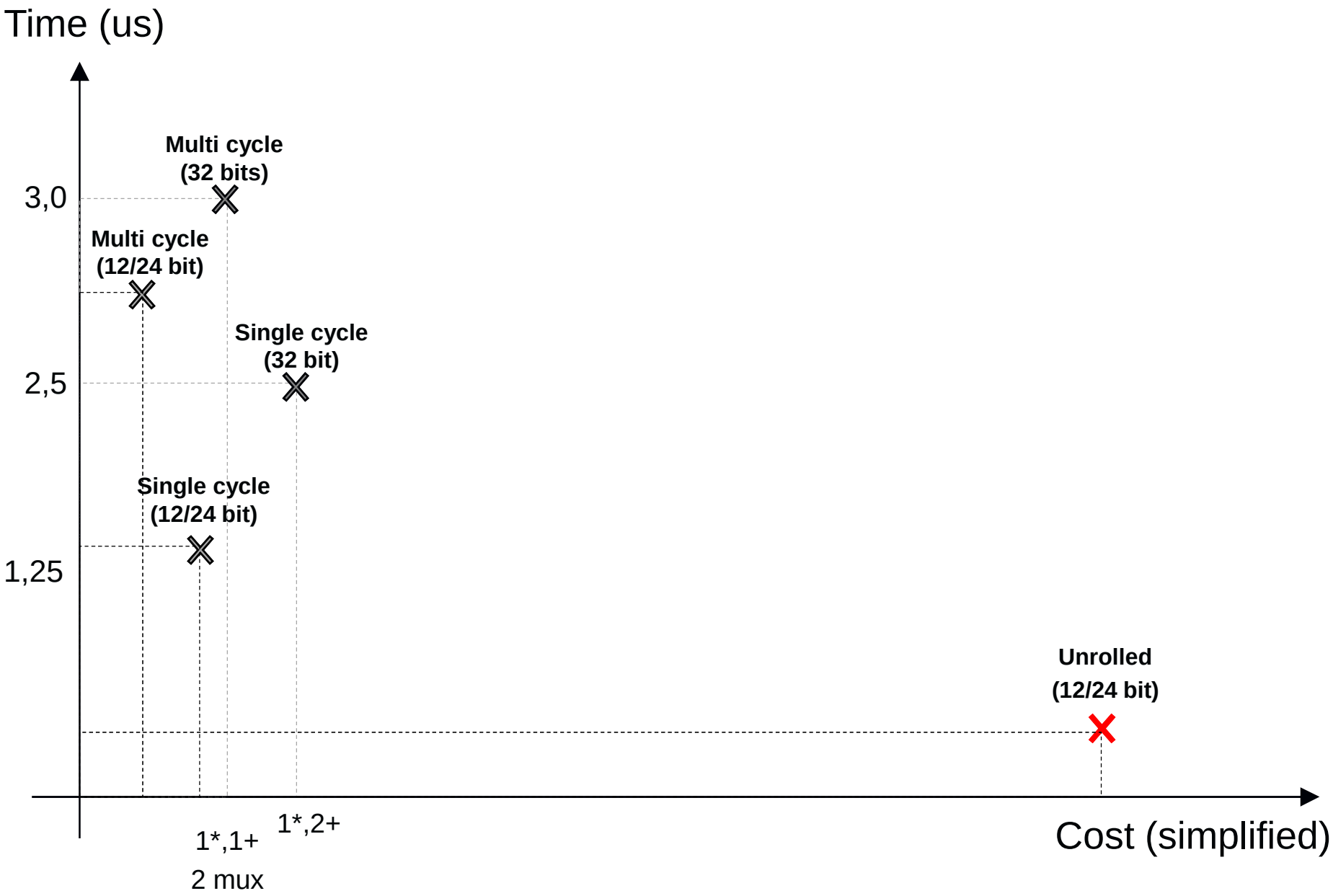


#cycles	f_{clk}	Exec time
128+1	280Mhz	0,45us



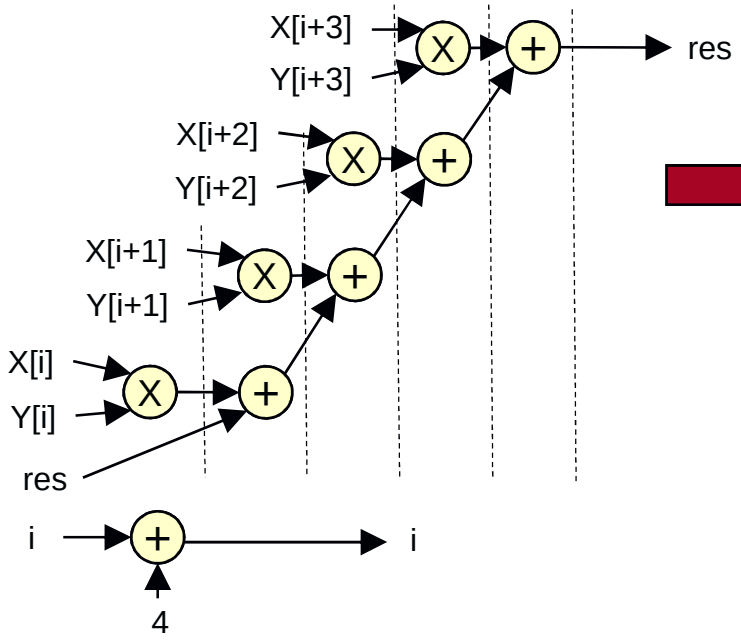
#cycles	f_{clk}	Exec time
128+1	333 Mhz	0,38 us

Loop unrolling perf/area trade-off

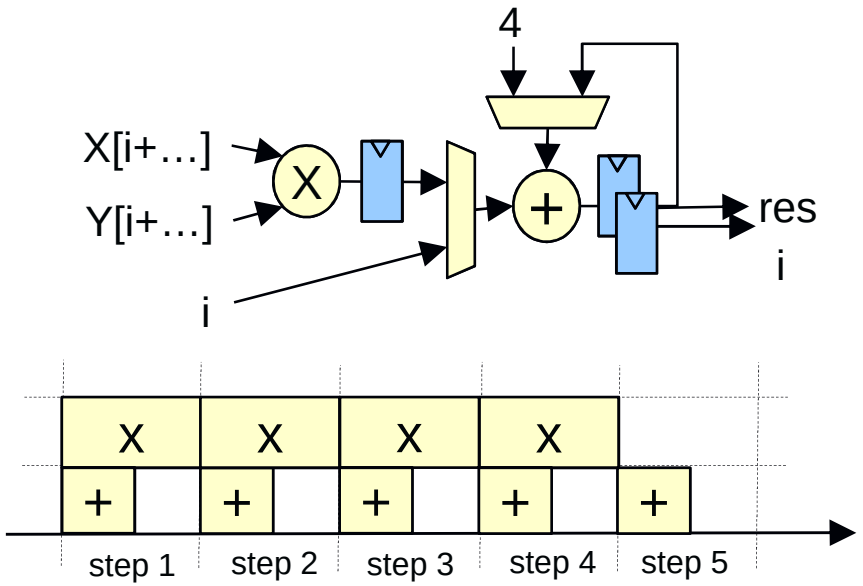


Loop unrolling + multi-cycle

- Unrolling can help improving multi-cycle schedules
  - By increasing hardware operator utilization rate thanks to additional reuse opportunities.

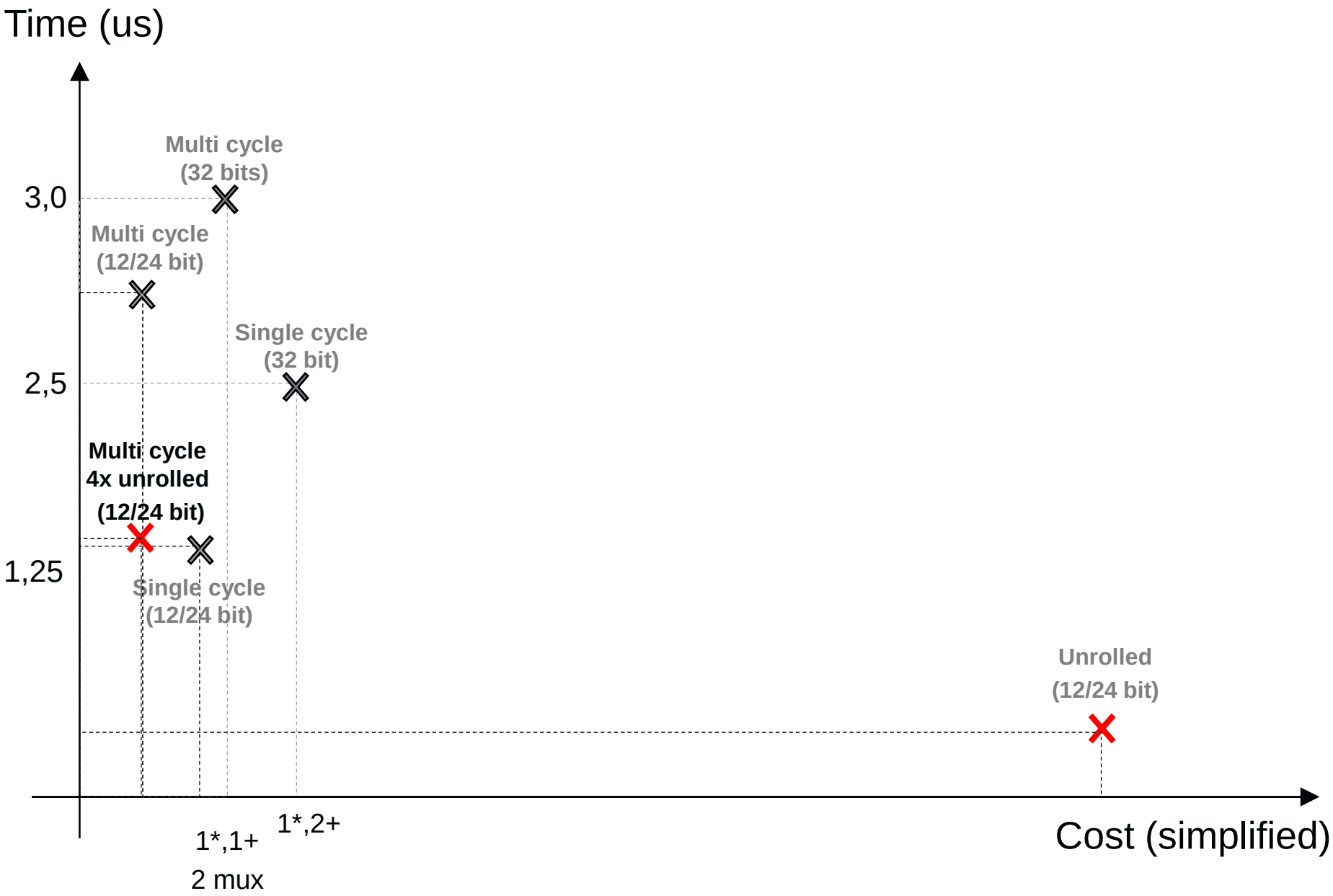


Same cost as the initial multi-cycle version, but more efficient use rate of the multiplier



#cycles	f_{clk}	Exec time
5*128+1	500 Mhz	1,28 us

Loop unroll + multi-cycle execution



Loop unrolling (partial/full)

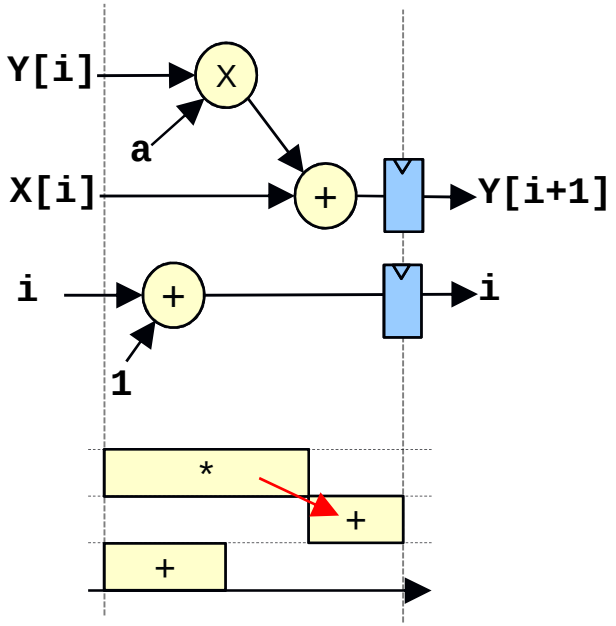
UNIVERSITÉ DE RENNES 1

- Most HLS tool can unroll loops automatically
  - Generally controlled through a compiler directive (see below)
  - Unrolling is often restricted to loops with constant bounds
  - Full unrolling is only used for small trip count loops
- Unrolling ***does not*** always improve performance
  - Loop carried dependencies can hurt/annhilate benefits
  - See example next slide …

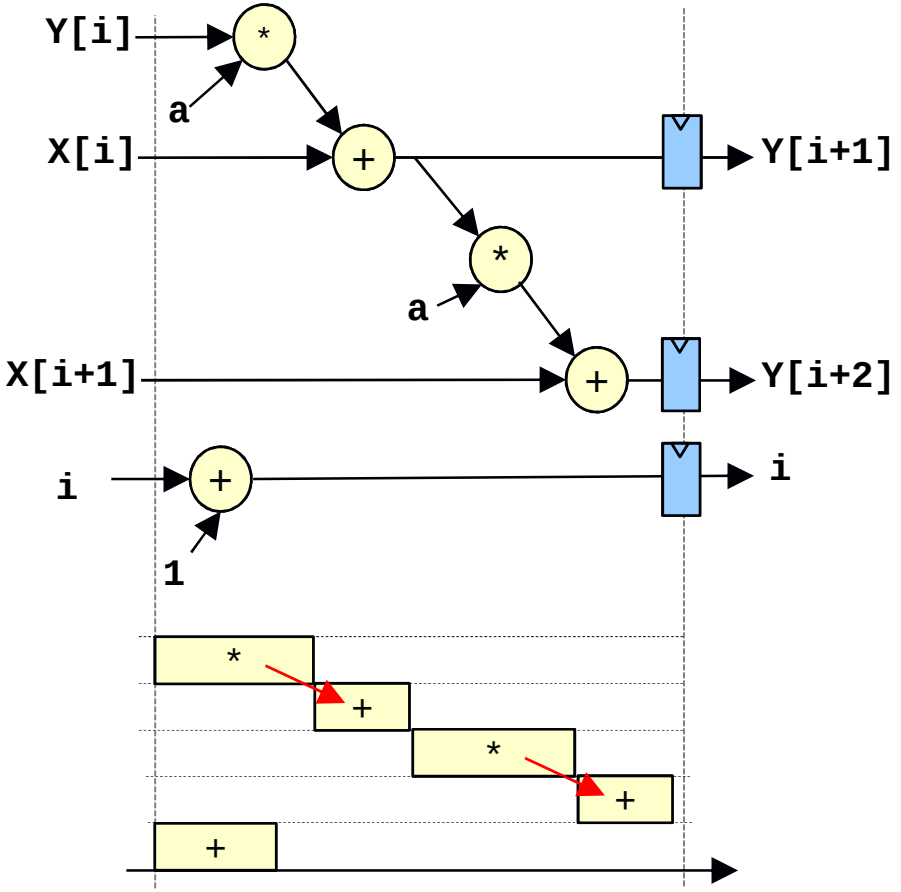
Loop unrolling (partial/full)

- Unrolling loops with carried dependencies

```
for(int i=0;i<256;i+=1){
   y[i+1] = a*y[i] + x[i];
}
```



#cycles	T_{clk}	Exec time
256+1	400 Mhz	0,64 us



#cycles	T_{clk}	Exec time
128+1	200 Mhz	0.64 us

Loop unrolling (partial/full)

UNIVERSITÉ DE RENNES 1

- Older HLS tools would fully unroll all loops
  - … even if the goal is not to get the fastest implementation !
  - … even for loops with large iteration counts (e.g. images)

Lack of proper support for loops is one of the reason behind the failure of the first generation of HLS tools

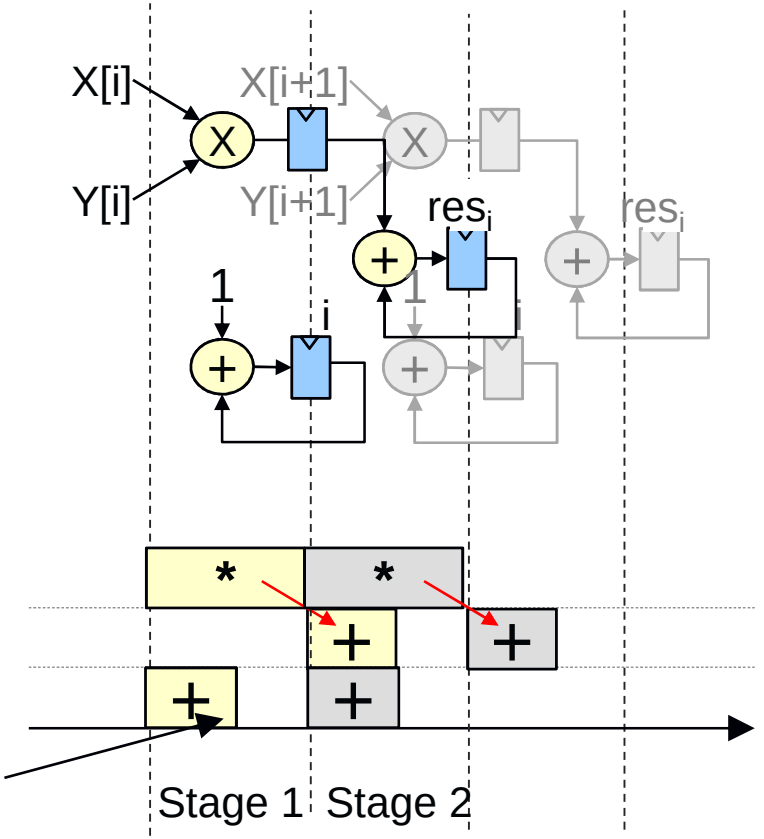
Loop pipelining

- Loop Pipelining = pipelined execution of loop iterations
  - Start iteration i+1 before all operation of iteration i are finished
  - Principles similar to software pipelining for VLIW/DSPs

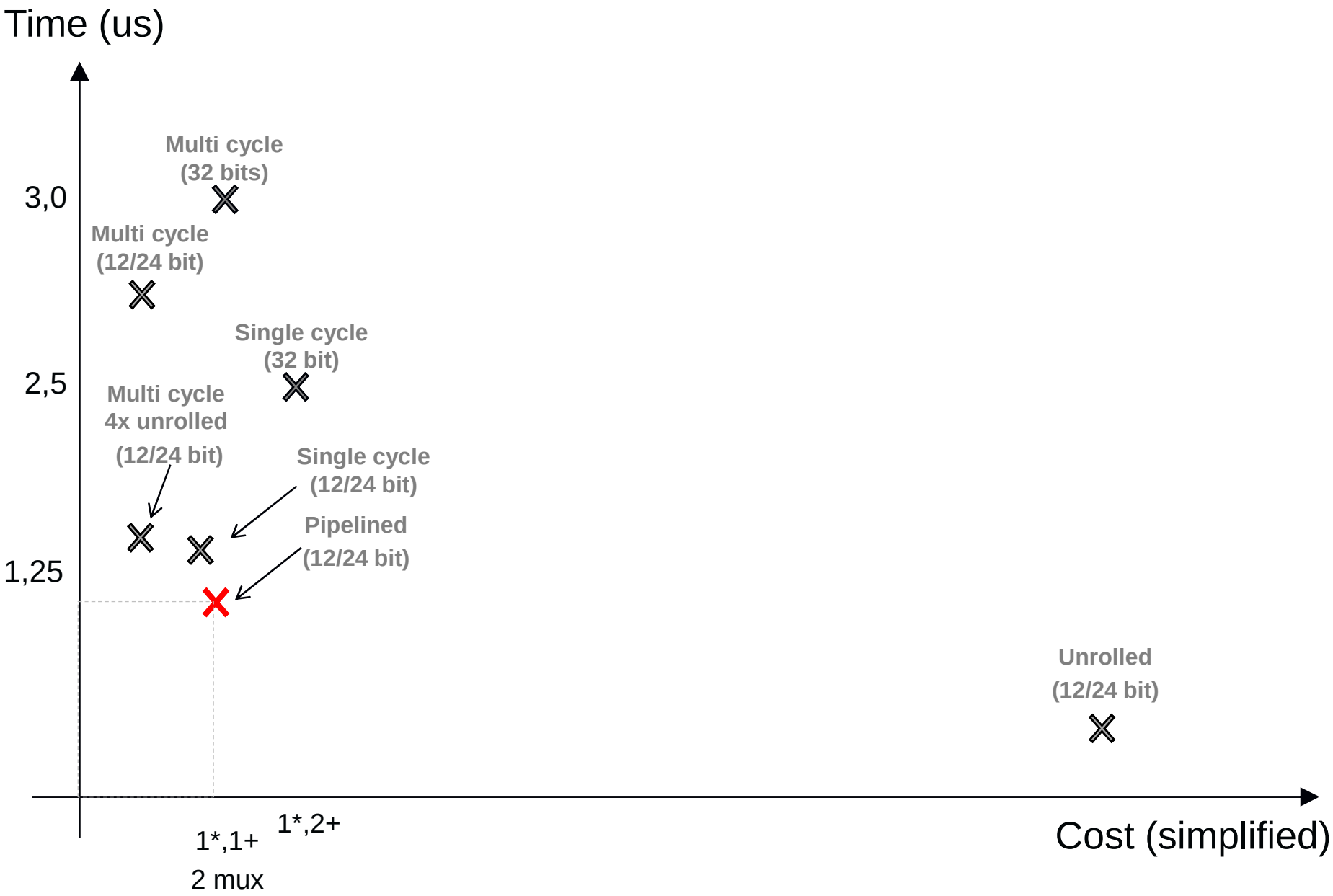
```
int12 X[512],Y[512];
int16 res=0; int24 tmp;
for (int9 i=0;i<512;i++){
   tmp = X[i]*Y[i];
   res = res + tmp>>8;
}
```

In this example a new iteration starts every cycle, all operations in the loop body have their own operator.

#cycles	T_{clk}	Exec time
512+1+1	500 Mhz	1,02us



Loop pipelining perf/area trade-off



Loop pipelining

UNIVERSITÉ DE RENNES 1

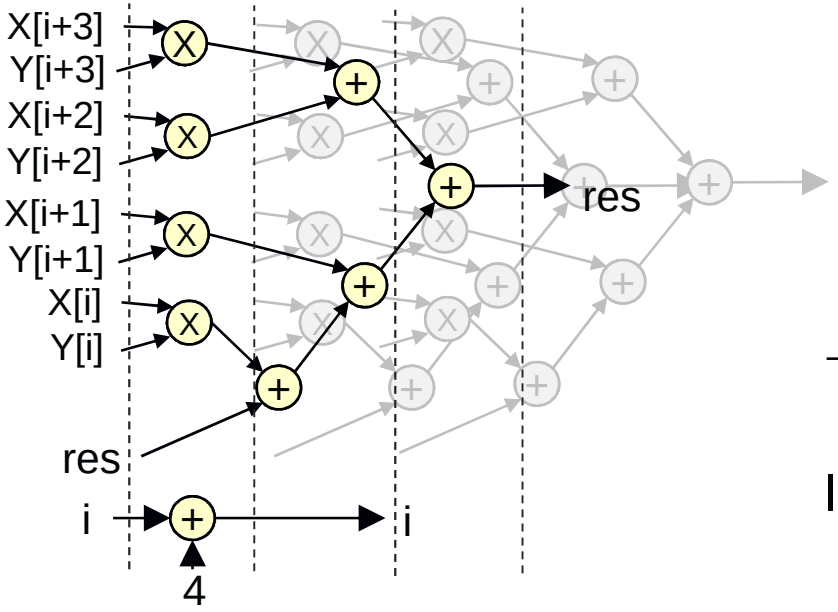
- A given pipelined loop schedule is characterized by
  - **Initiation interval (II)** : delay separating the execution of two successive iterations. A perfect loop pipelining lead to II=1.
  - **Pipeline Latency (L)** : number of cycles needed to completely execute a given iteration (we have L>1).

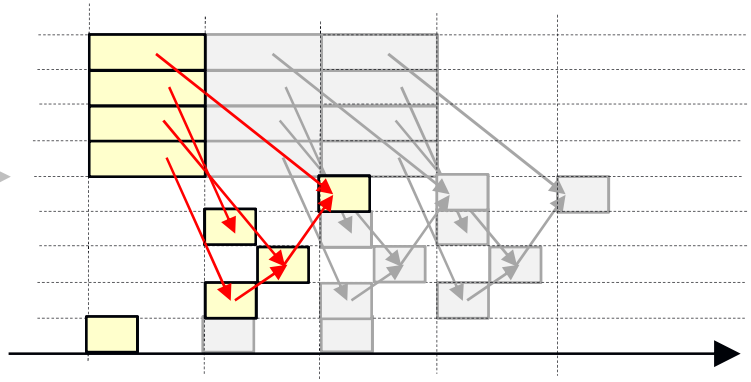
Targeting a value II=1 is common as the number of operators is not constrained by a predefined architecture as in VLIW/DSPs

- Loop pipelining enables very deeply pipelined datapath
  - Operators are also aggressively pipelined (e.g float)
  - Latency in the order of 100's of cycles is not uncommon
- Pipelining is the most important optimization in HLS
  - Huge performance improvements for marginal resource cost (plenty of registers in FPGA cells).

Pipelining + unrolling = vectorizing

- Pipelining + unrolling by K further increase performance
  - We execute *K* iterations following a SIMD execution model
  - Execution of the *K* iterations packets are overlapped



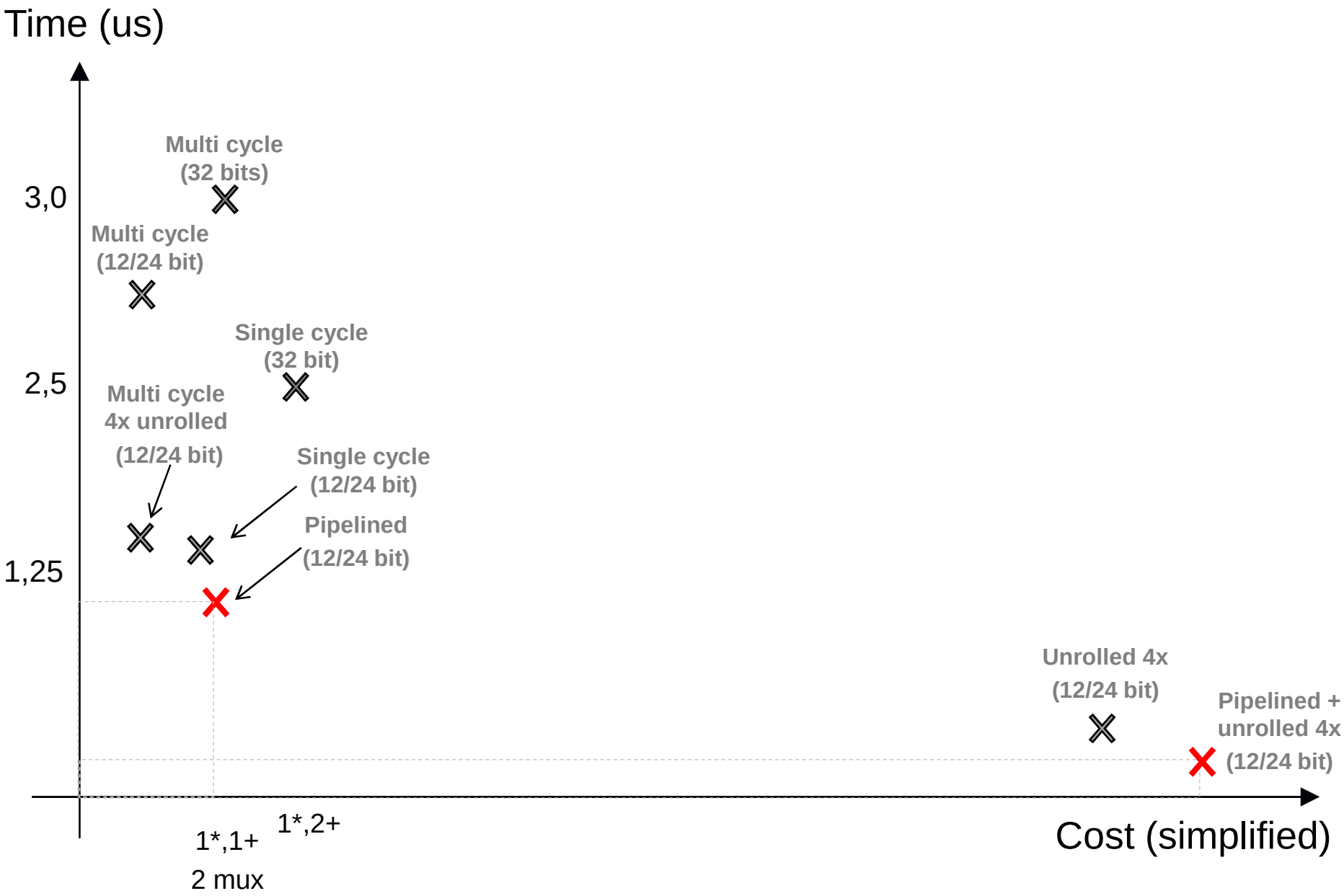


II=1, L=3; $T_{clk}=\max(T_{mul},2*T_{add})$

Loop executed in 8+2 cycles

- However, achieving II=1 is often not straighforward
  - We'll see that in a later slide ….

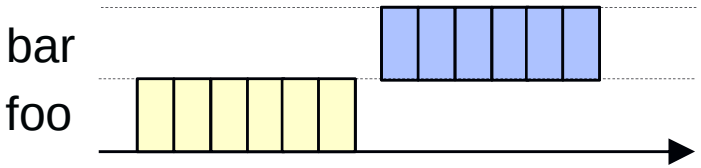
Loop pipelining perf/area trade-off



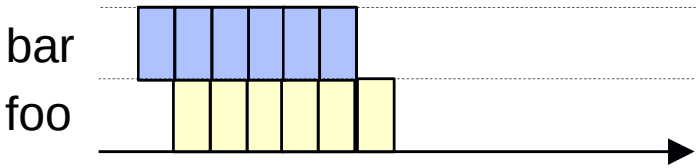
Loop fusion

- Merging consecutive loops into a single one
  - Everything happens as if the two loops are executed in parallel
  - It helps reducing the impact of loop pipelining fill/flush phases

```
for(i=0;i<256;i++){
    y[i] = foo(x[i]);
}
for(i=0;i<256;i++){
    z[i] = bar(y[i]);
}
```




```
for(i=0;i<256;i++){
    if(i<256)
       y[i] = foo(x[i]);
    if(i>0)
       z[i-1] = bar(y[i-1]);
}
```



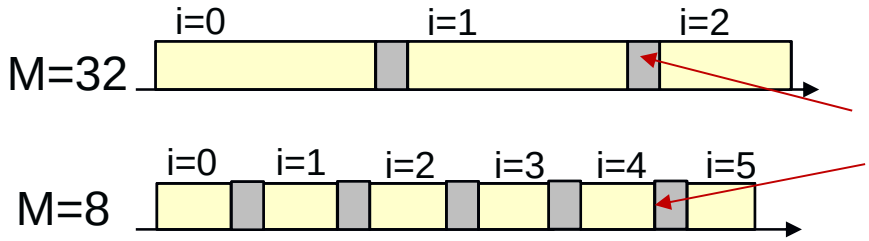
- Loop merging is not alway obvious due to dependencies
  - In some cases, merging must be combined with loop shifting

Nested loop support in HLS

- In general HLS tools only pipeline the innermost loop
  - No big deal when innermost loops have large trip counts

```
for(i=0;i<256;i++){
  for(j=0;j<M;j++){
    x[i][j] = foo(x[i][j]);
  }
}
```

- There are cases when this leads to suboptimal designs
  - For example when processing small 8x8 pixel macro-blocks, or for deeply pipelined datapath with 100's of stages

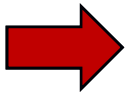


Pipeline overhead is more significant for short trip count innermost loops

How to pipeline outer loops ?

- Completely unroll all innermost loops to get a single loop
  - Supports non perfectly nested innermost loops
  - Only viable for small constant bounds inner most loops

```
for(i=0;i<256;i++){
  for(j=0;j<4;j++){
    x[i][j] = foo(x[i][j]);
  for(j=0;j<2;j++){
    x[i][j] = bar(x[i][j]);
}
```



```
for(i=0;i<256;i++){
  x[i][0] = foo(x[i][0]);
  x[i][1] = foo(x[i][1]);
  x[i][2] = foo(x[i][2]);
  x[i][3] = foo(x[i][3]);
  x[i][0] = bar(x[i][0]);
  x[i][1] = bar(x[i][1]);
}
```

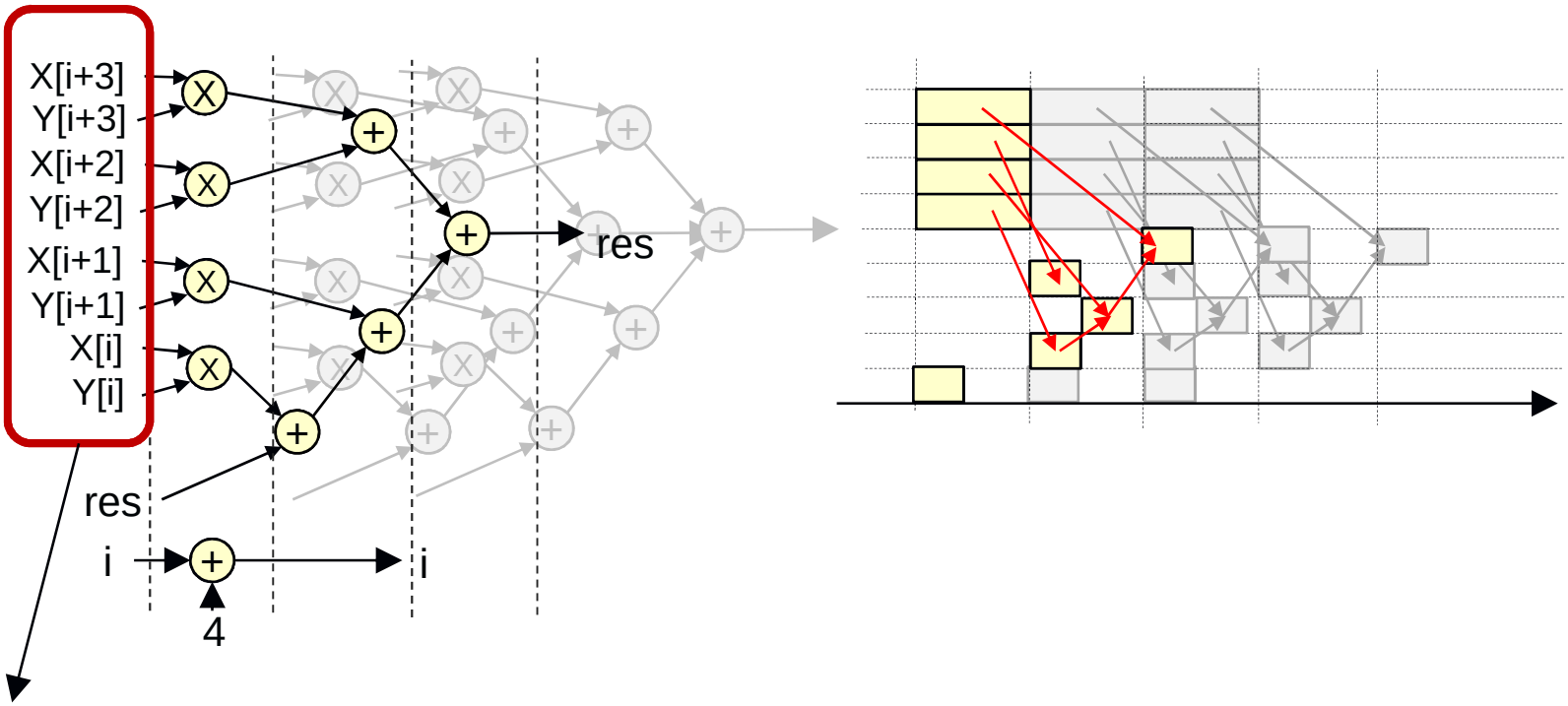
- Flattening/coalescing the loop nest hierarchy
  - Only for perfectly nested loops, use with caution !

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

```
for(k=0;k<M*N;k++){
  i=k/N;
  i=k%N;
  x[i][j] = foo(x[i][j]);
}
```

Memory port resource conflicts

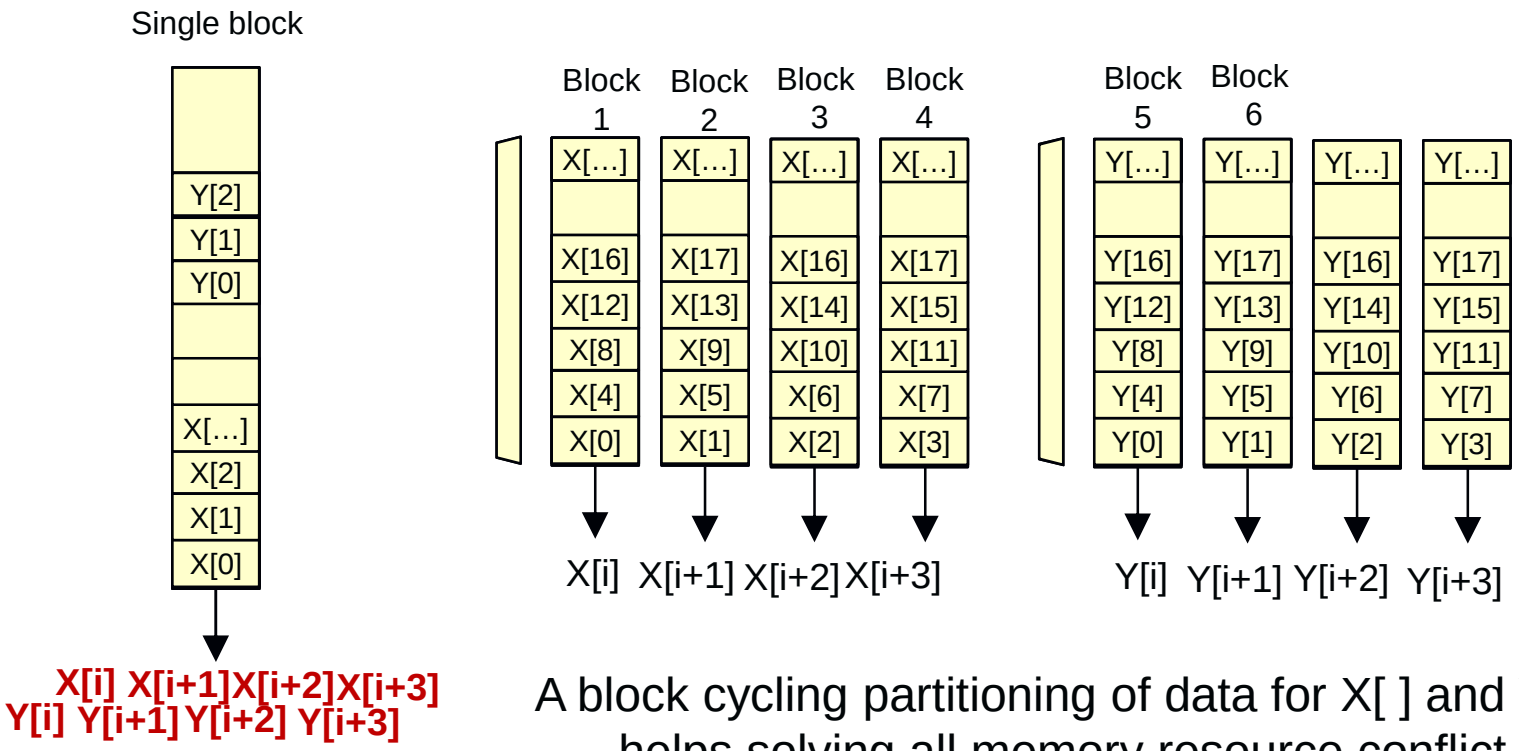
- The main limiting factor for reaching II=1 is memory
  - When II=1, there are generally many concurrent accesses to a same array (and thus to a same memory block).



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

Partitioning arrays in banks

- Duplicating memory reduces memory port conflicts
  - Naïve duplication is not efficient, array data must be allocated to several memory blocks to reduce the number of conflicts.



A block cycling partitioning of data for X[ ] and Y[ ] helps solving all memory resource conflict

Partitioning arrays in memory banks

- Partitioning can be done at the source level
  - Can be tedious, many HLS tools offer some automation

```
float X[32],Y[32];
res=0.0;


for (int i=0;i<8;i+=4){
   tmp = X[i]*Y[i];
   res = res + tmp;
   tmp = X[i+1]*Y[i+1];
   res = res + tmp;
   tmp = X[i+2]*Y[i+2];
   res = res + tmp;
   tmp = X[i+3]*Y[i+3];
   res = res + tmp;
}
```

```
float X0[8],X1[8],X2[8],X3[8];
float Y0[8],Y1[8],Y2[8],Y3[8];
res=0.0;

for (int i=0;i<8;i+=4){
   tmp = X0[i]*Y0[i];
   res = res + tmp;
   tmp = X1[i]*Y1[i];
   res = res + tmp;
   tmp = X2[i]*Y2[i];
   res = res + tmp;
   tmp = X3[i]*Y3[i];
   res = res + tmp;
}
```

Eliminating conflicts through scalarization

- Scalarization : copy array cell value in a scalar variable
  - Use the scalar variable instead of the array whenever possible
  - Obvious reuse is often performed by the compiler itself
  - Beware, reuse accross iteration of a loop is not automated

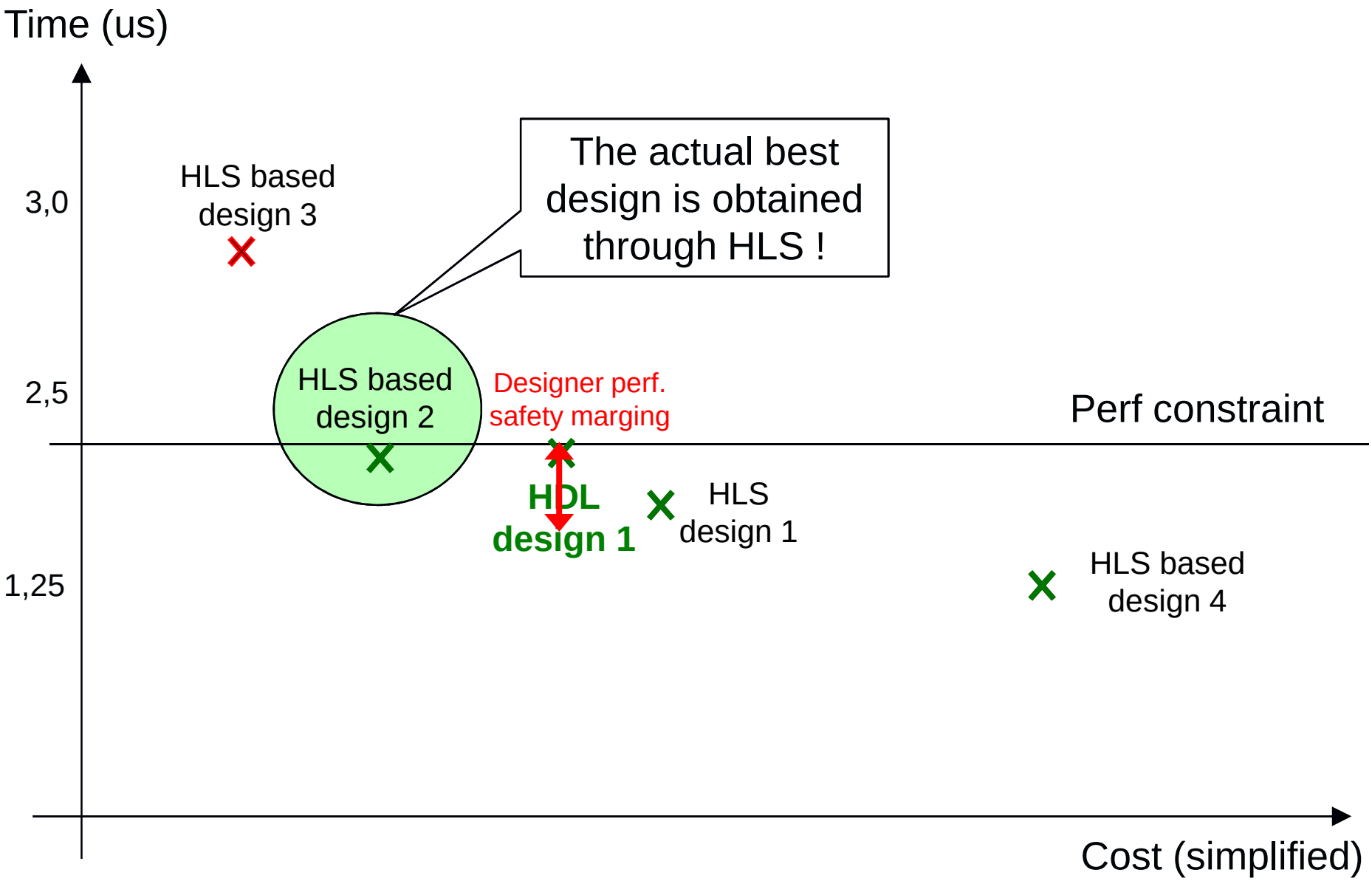
```
float X[128];
float res=0.0;


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

Another take on HLS vs HDL



Outline

- Why High Level Synthesis ?
- Challenges when synthesizing hardware from C/C++
- High Level Synthesis from C in a nutshell
- Exploring performance/area trade-off
- Program optimizations for hardware synthesis
- The future of HLS …

HLS in 2020

UNIVERSITÉ DE RENNES 1

- HLS tools are focusing on hardware IP design
  - In 2016, a whole system can be specified within a single flow
  - In 2020 hardware/software boundaries will have faded away
- Today, poor support for dynamic data-structures
  - Speculative execution techniques from processor design may help widen the scope of applicability of HLS to new domains
- Most program transformations are done by hand
  - More complex semi automatic program transformations !
  - Complex combination of program transformations to obtain the best design (iterative compilation + High Level Synthesis)
- PhP developpers still can't design hardware
  - Well, that is not likely to change yet ……

Loop tiling

- Break the loops into « tiles » or blocks
  - Expose coarse grain parallelism (for threads)
  - Improve temporal reuse over loop nests
  - Legal only if all loop are permutable (i.e. can be interchanged)

```
float **A,**B,**C;
for(i=0;i<16;i++) {
  for(j=0;j<16;j++) {
    for(k=0;k<16;k++)
S0:   C[i,j]+=B[i,k]*A[k,j];
  }
}
```

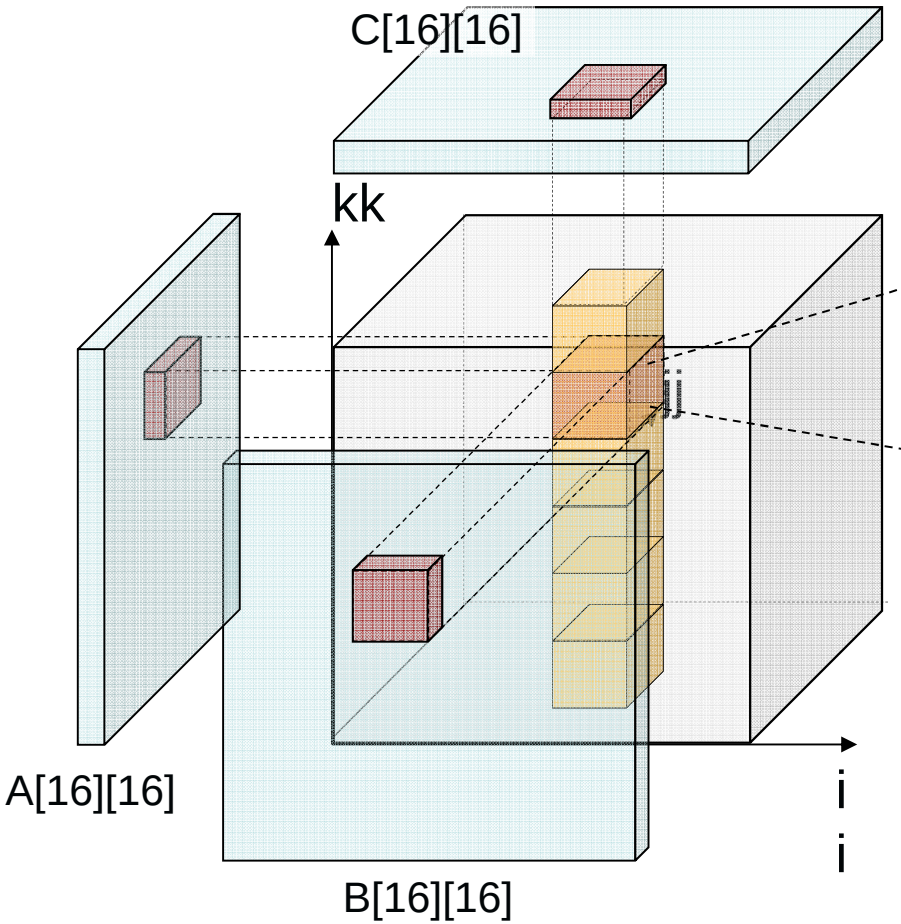
```
float **A,**B,**C;
for(ii=0;ii<16;ii+=4)
 for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
    for(i=0;i<4);i++)
     for(j=0;j<4;j++)
       for(k=0;k<4;k++)
S1:     C[i+ii,j+jj]+=
           B[i+ii,k+kk]*A[k+kk,j+jj];
```

A 4x4x4 Tile

- Best understood with an example
  - We consider a fully associative 32 word data cache
  - The cache is organized as 8 cache lines of 4 words each

Loop Tiling example (matrix product)

- Tiling helps improving spatial and temporal locality
  - One chooses tile size such that all data fits into the cache



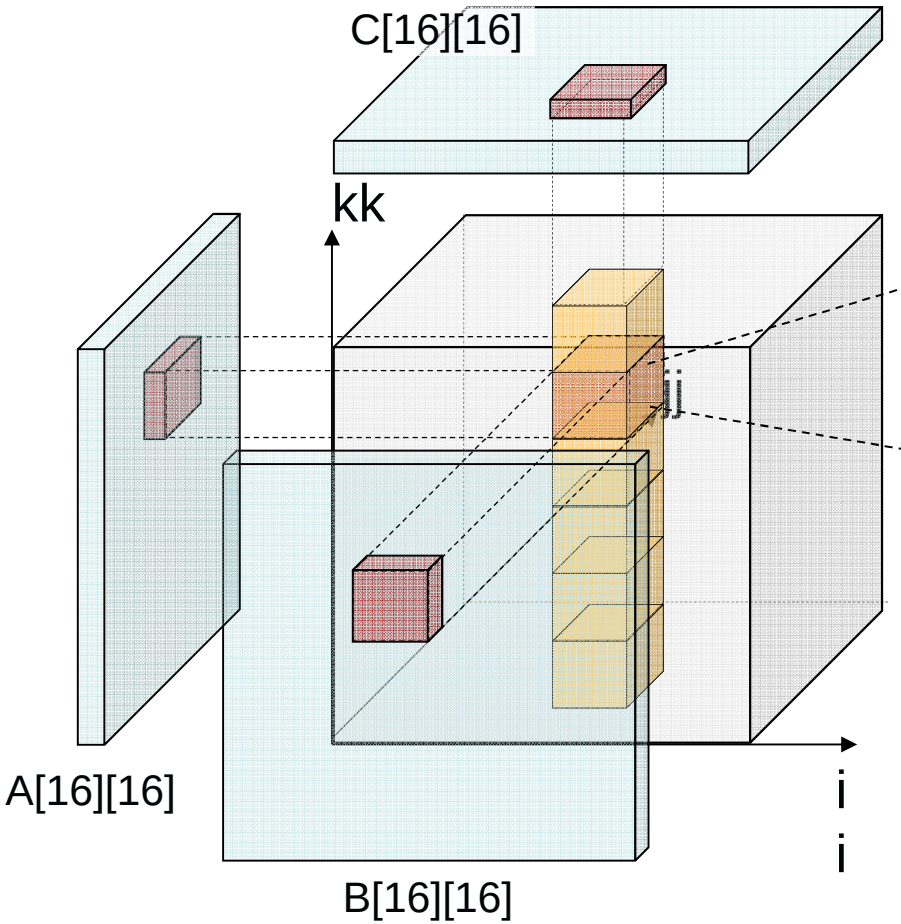

```
float **A,**B,**C;
for(ii=0;ii<16;ii+=4)
 for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
```

A 4x4x4 Tile

```
   for(i=0;i<4);i++)
    for(j=0;j<4;j++)
      for(k=0;k<4;k++)
S1:    C[i+ii,j+jj]+=
          B[i+ii,k+kk]*A[k+kk,j+jj];
```

Loop Tiling example (matrix product)

- Tiling helps improving spatial and temporal locality
  - One chooses tile size such that all data fits into the cache

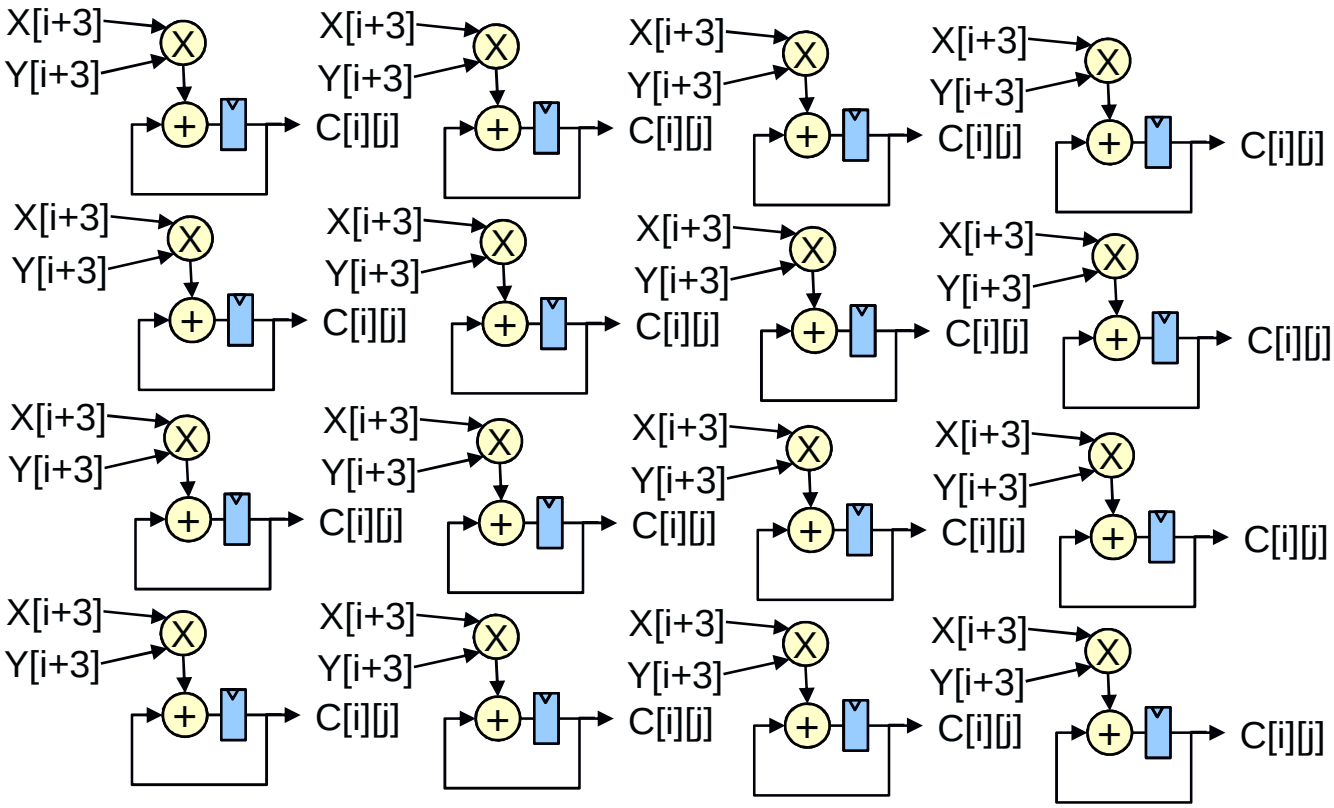


A 4x4x4 Tile

```
float **A,**B,**C;
for(ii=0;ii<16;ii+=4)
 for(jj=0;jj<16;jj+=4)
   #pragma pipeline II=1
   for(kk=0;kk<16;kk+=4)
     memcpy(A,lA,…)
   for(k=0;k<4;k++)
     for(i=0;i<4);i++)
     for(j=0;j<4;j++)
S1:    C[i+ii,j+jj]+=
          B[i+ii,k+kk]*A[k+kk,j+jj];
```

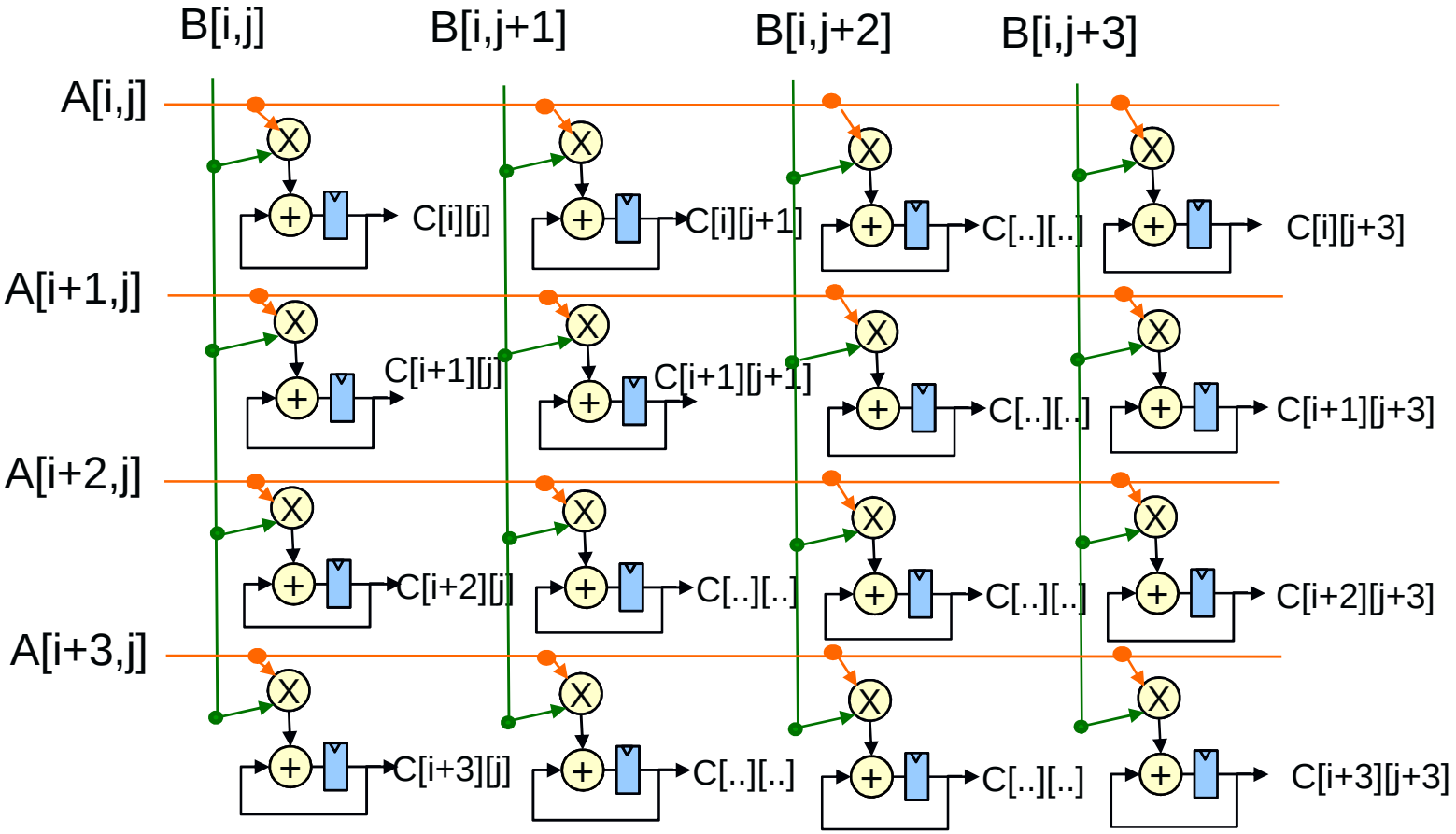
Loop tiling

- Execution of the initial kernel
  - We show which parts of A,B,C arrays are in the cache



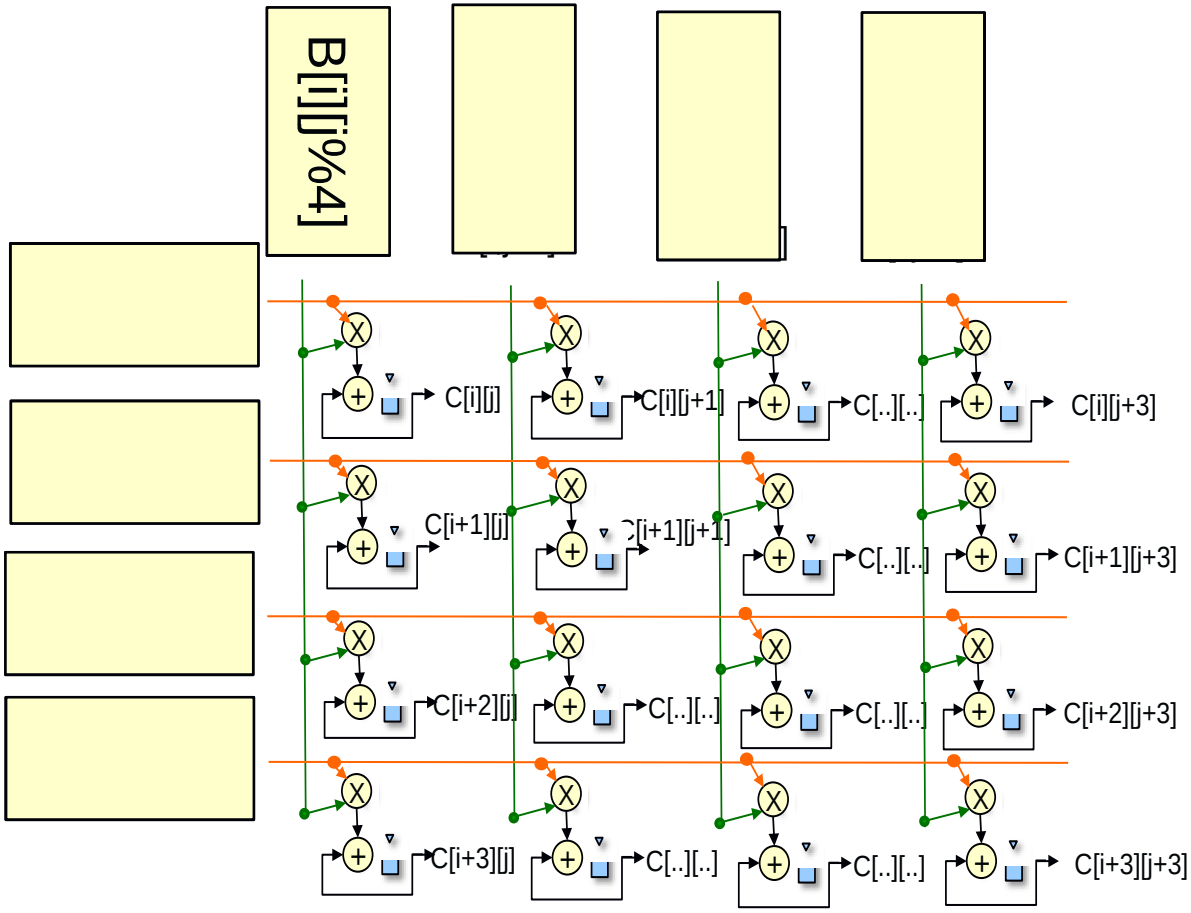
Loop tiling

- Execution of the initial kernel
  - We show which parts of A,B,C arrays are in the cache



Loop tiling

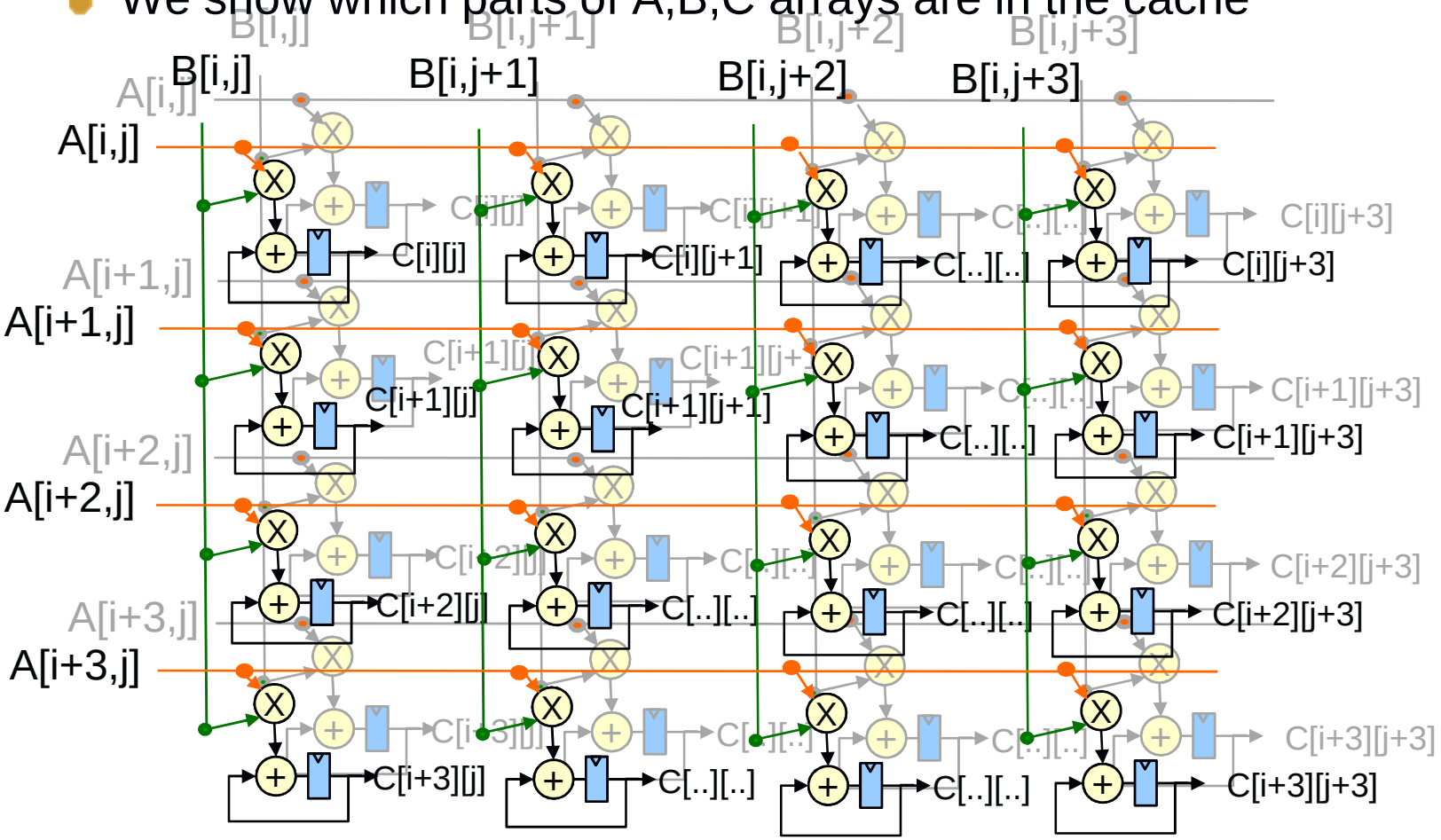
- Execution of the initial kernel
  - We show which parts of A,B,C arrays are in the cache



M2R-CSS

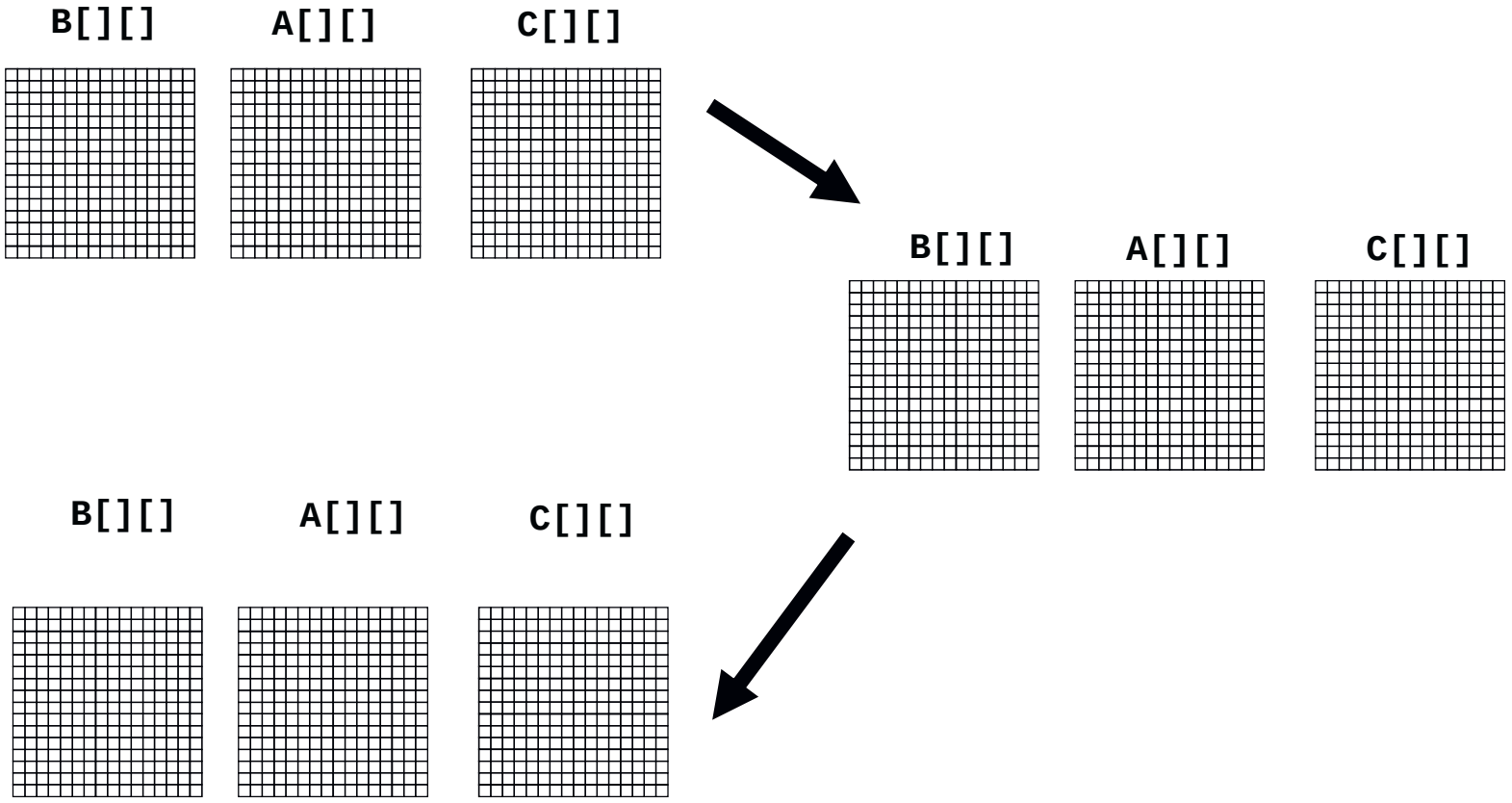
Loop tiling

- Execution of the initial kernel
  - We show which parts of A,B,C arrays are in the cache



Loop tiling

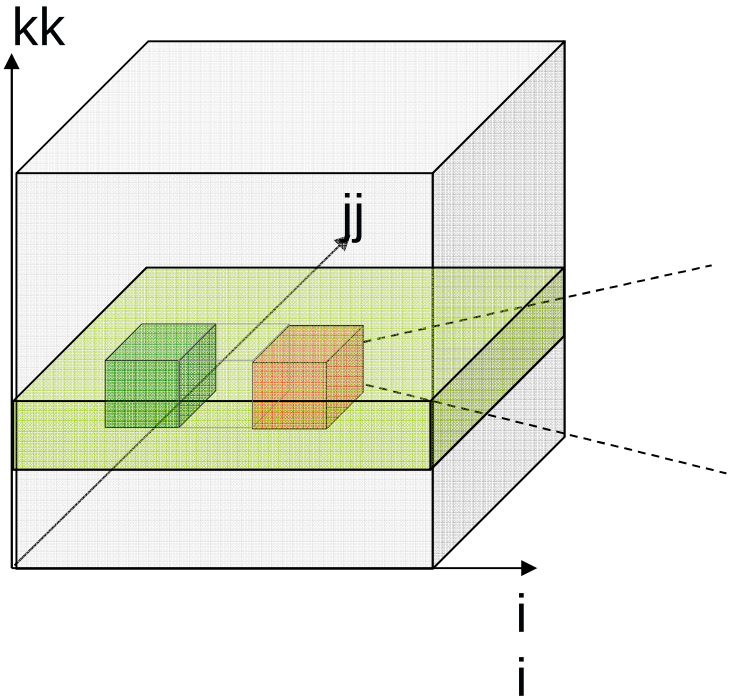
- Execution of the tiled kernel
  - For a single tile (ii=jj=kk=0)



Loop Tiling & parallelization

UNIVERSITÉ DE RENNES 1

- Simple way of exposing coarse grain parallelism
  - Tiles are executed as *atomic* execution units, there is no synchronization during a tile execution.
- Tiling enables efficient parallelization
  - It improves locality and reduces synchronization overhead
  - Finding the “right” tile size and shape is difficult (open problem)

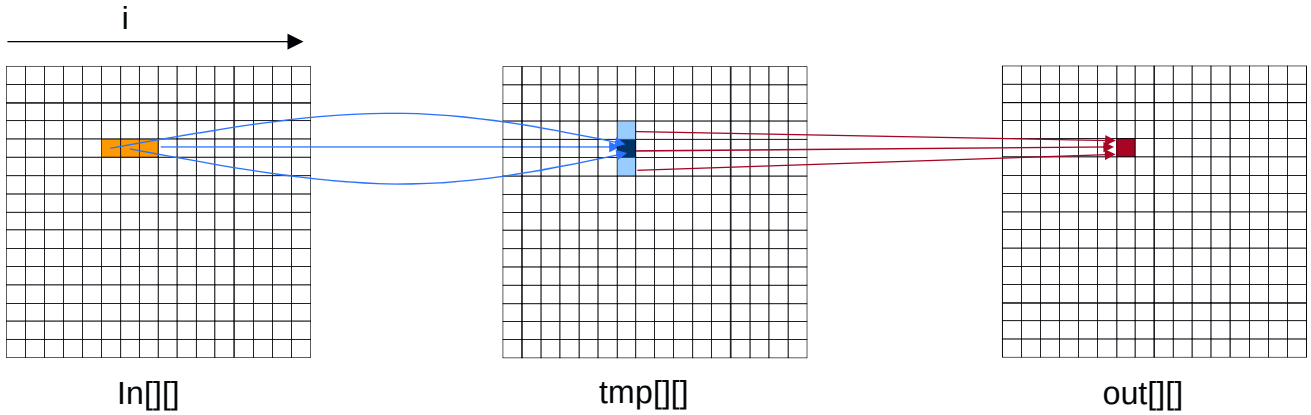



```
float **A,**B,**C;
#pragma omp parallel for private(ii)
for(ii=0;ii<16;ii+=4)
#pragma omp parallel for private(jj)
 for(jj=0;jj<16;jj+=4)
  for(kk=0;kk<16;kk+=4)
    for(i=0;i<4);i++)
     for(j=0;j<4;j++)
      for(k=0;k<4;k++)
S1:     C[i+ii,j+jj]+=
            B[i+ii,k+kk]*A[k+kk,j+jj];
```

Challenge

- Image processing pipeline example

```
for(i=1;i<N-1;i++)
  for(j=0;j<M;j++)
S0:  tmp[i][j]=f(in[i][j],in[i-1][j],in[i+1][j]);

for(i=1;i<N-1;i++)
  for(j=1;j<M-1;j++)
S1:  out[i][j]=f(tmp[i][j],tmp[i][j-1],tmp[i][j+1]);
```

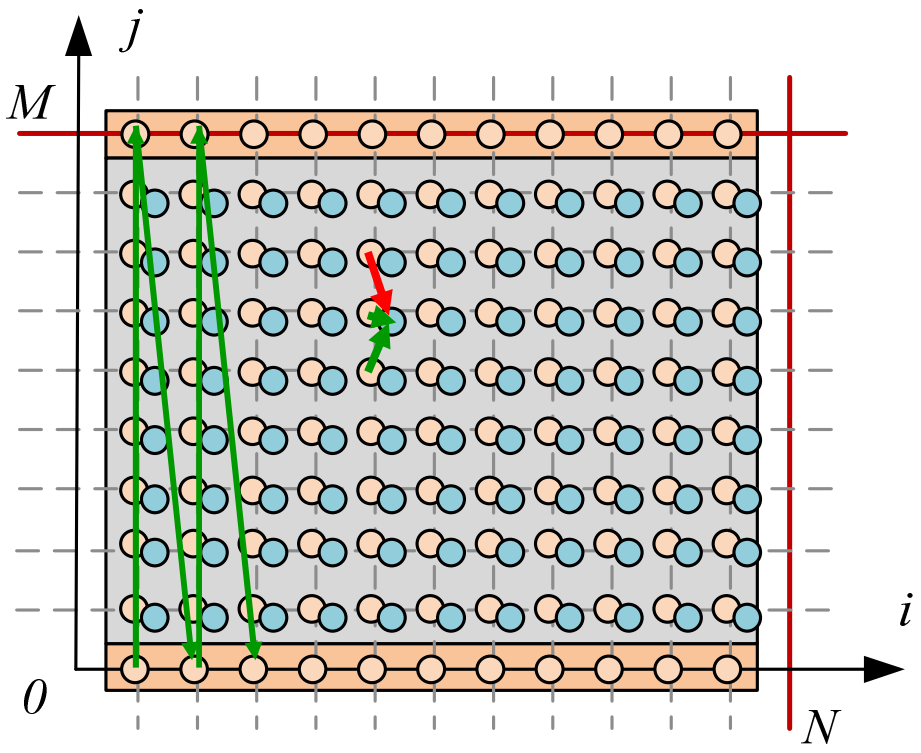


- Find a legal fusion and optimize memory accesses

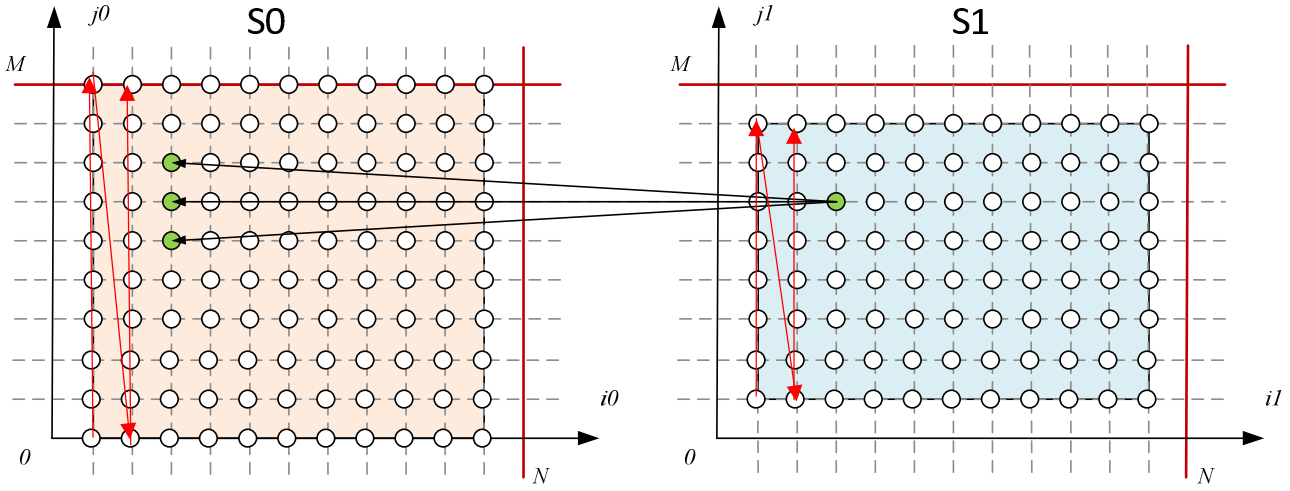
Exercise 1 (solution)

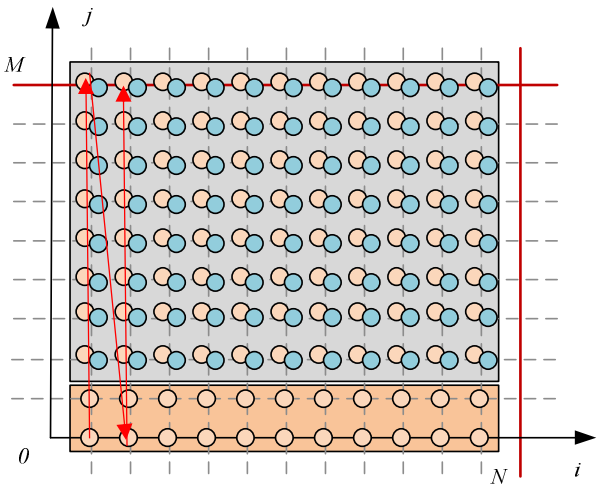
- Straight forward fusion is not possible, needs to be combined with loop shifting.

```
//illegal transformation
for(i=1;i<=N-2;i+=1){
  tmp[i][0] = 3*in[i][0]
    -in[i-1][0]-in[i+1][0];
  for(j=1;j<=M-2;j+=1){
    tmp[i][j] = 3*in[i][j]
      -in[i-1][j]-in[i+1][j];
    out[i][j] = 3*tmp[i][j]
      -tmp[i][j-1]-tmp[i][j+1];
  }
  tmp[i][M-1] = 3*in[i][M-1]
      -in[i-1][M-1]-in[i+1][M-1];
}
```



Exercise 1 (solution)






```
for (i = 1; i < N-1; i++)
  for (j = 0; j < 2; j++)
S0:  tmp[j%3] = f(in[i][j], in[i-1][j], in[1+i][j]);
  for (j = 2; j < M; j++)
S0:  tmp[j%3] = f(in[i][j], in[i-1][j], in[1+i][j]);
S1:  out[i][j-1] = f(tmp[(j-1)%3],tmp[(j-2)%3],tmp[j%3]);
```