J. DE FINE LICHT AND T. HOEFLER

Productive Parallel Programming on FPGA with High-Level Synthesis

Based on material from:

Transformations of High-Level Synthesis Codes for High-Performance Computing
https://arxiv.org/abs/1805.08288

Code examples found at:

https://github.com/spcl/hls_tutorial_examples

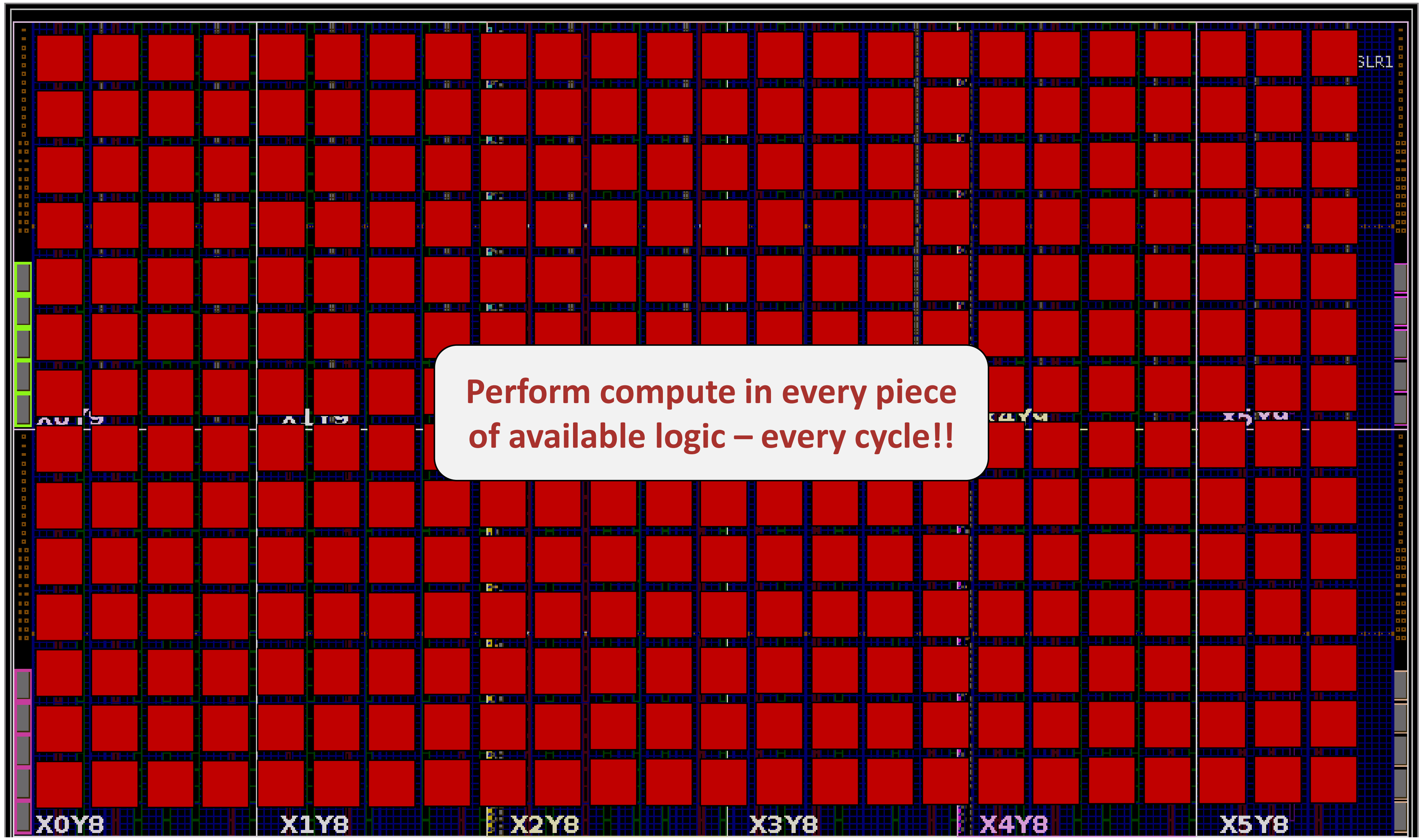
Virtual machine for emulation

http://spcl.inf.ethz.ch/~definelj/HLS_Tutorial.7z

Nimbix Alveo Trial

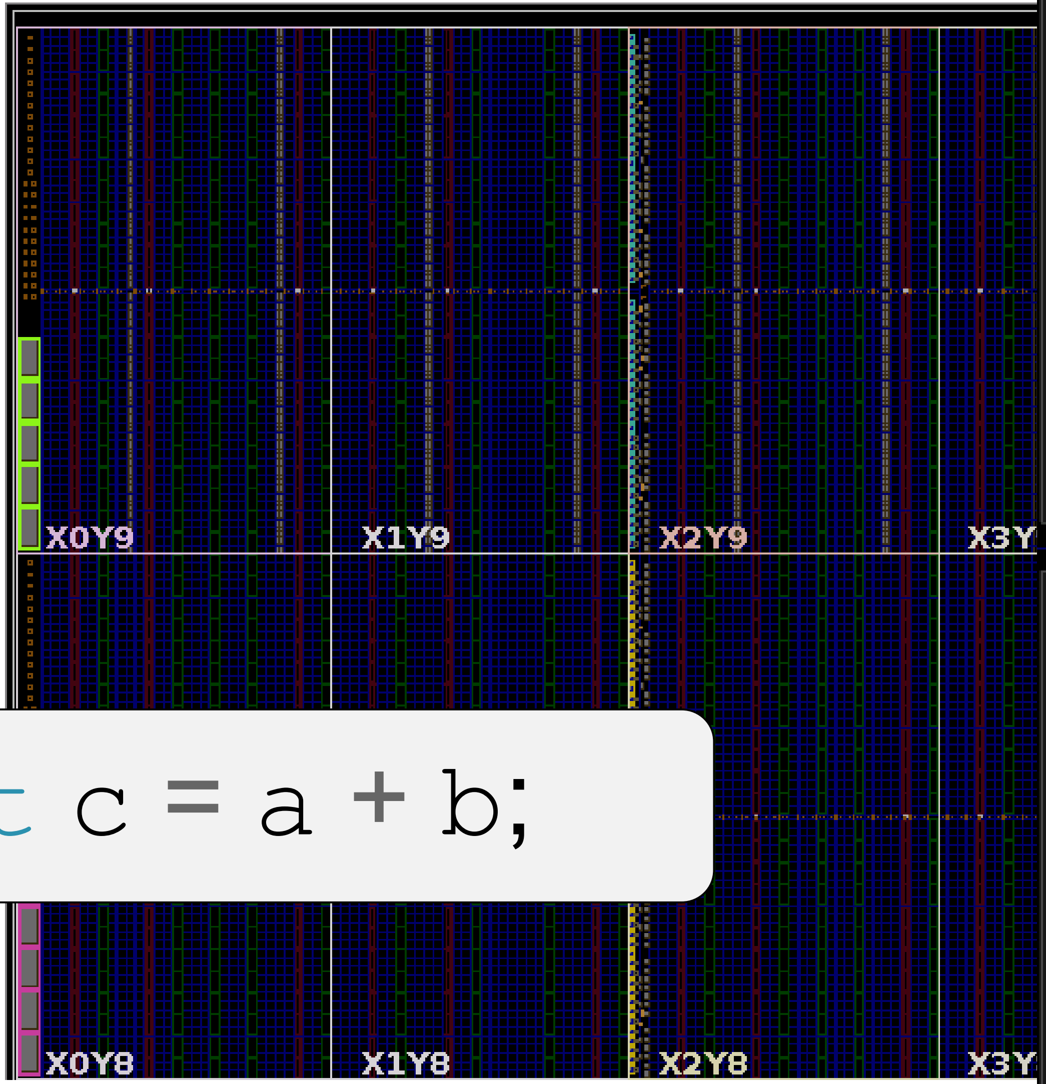
https://www.nimbix.net/alveotrial

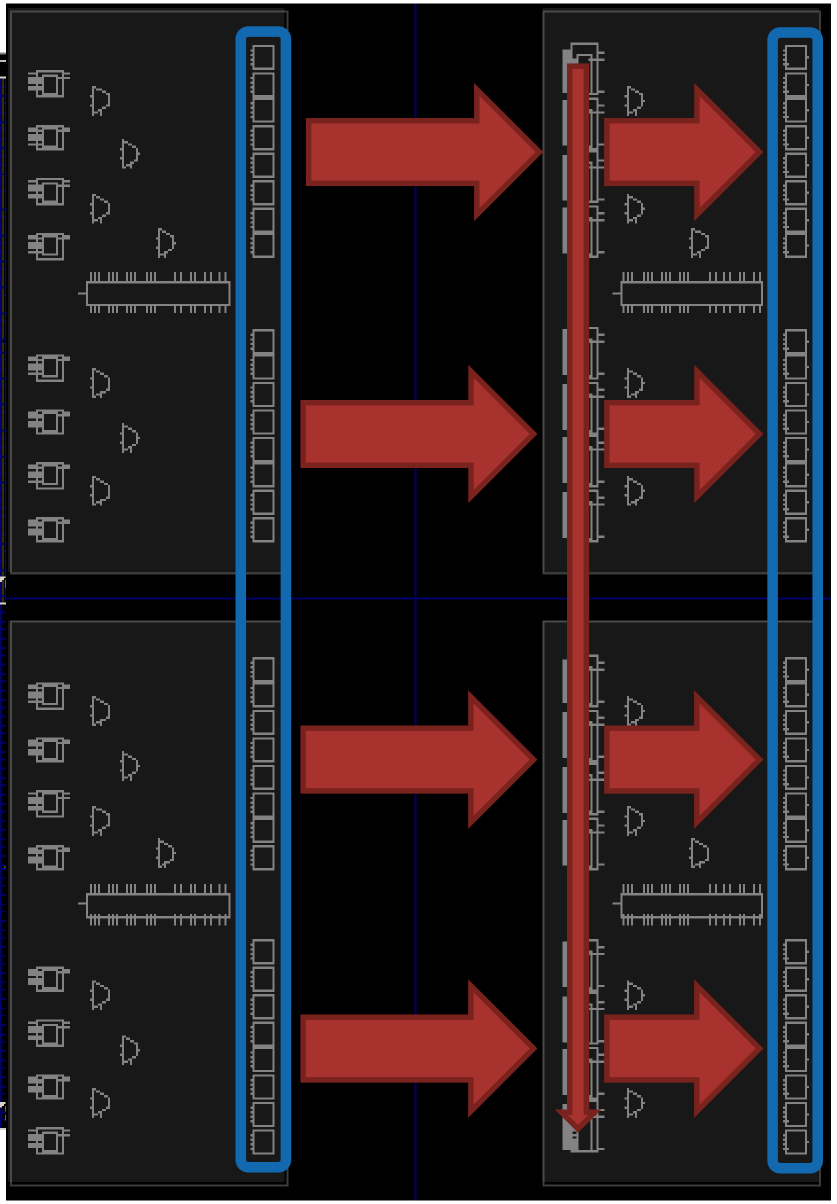
Our goal



Perform compute in every piece of available logic – every cycle!!

A single operation

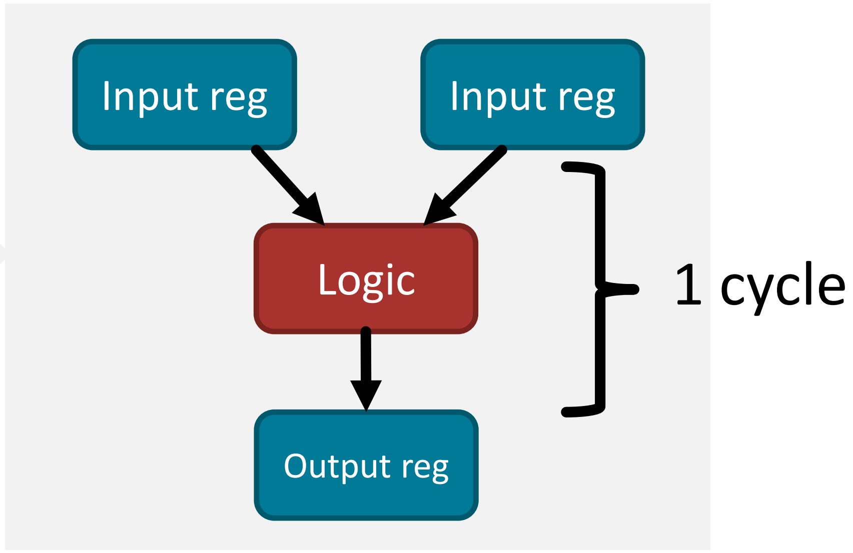




```
int c = a + b;
```

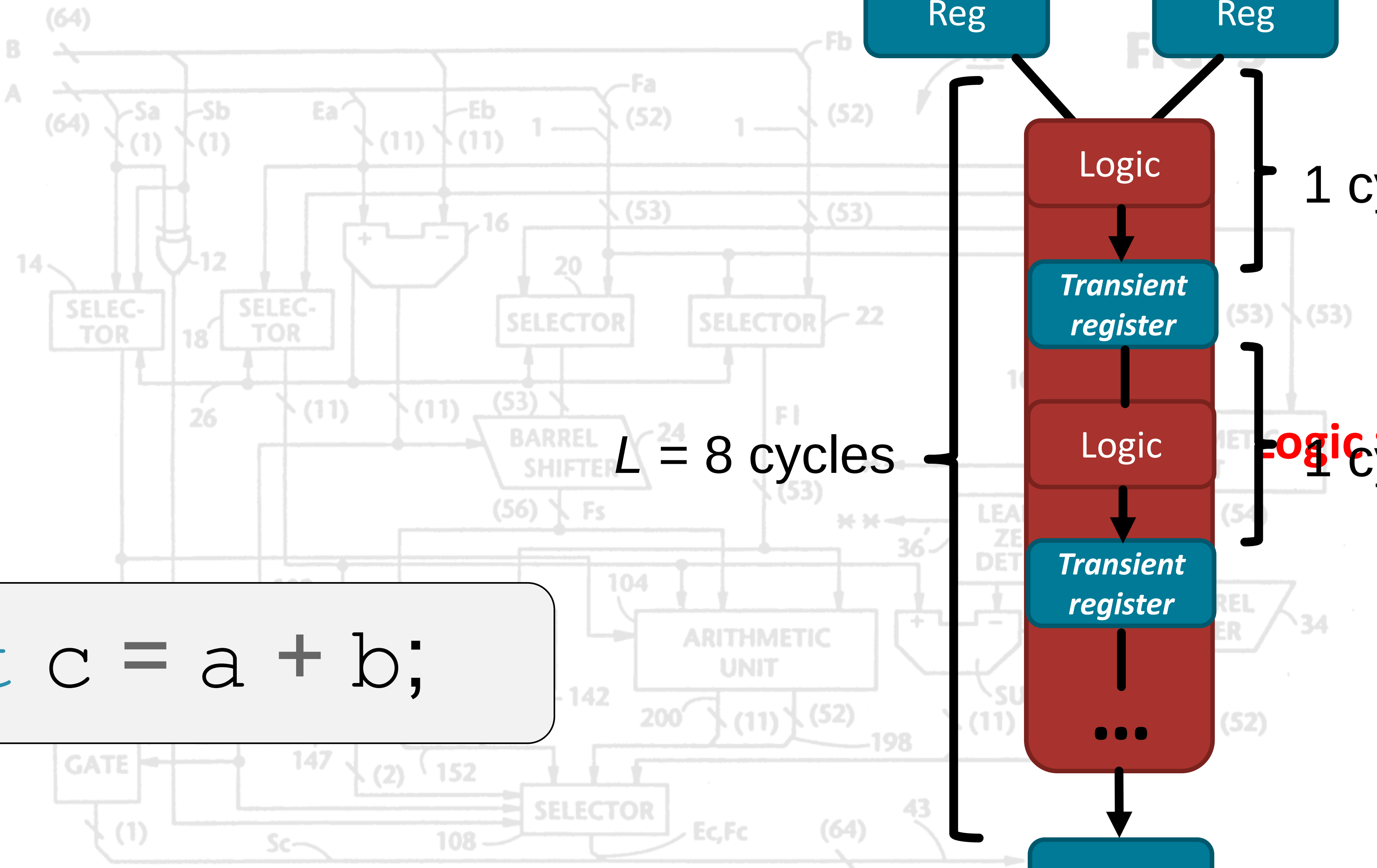
Register transfer level

```
always @(posedge clk)
  if (start) begin
      out <= in + 1;
  end
```



```
int c = a + b;
```

Single floating point operation



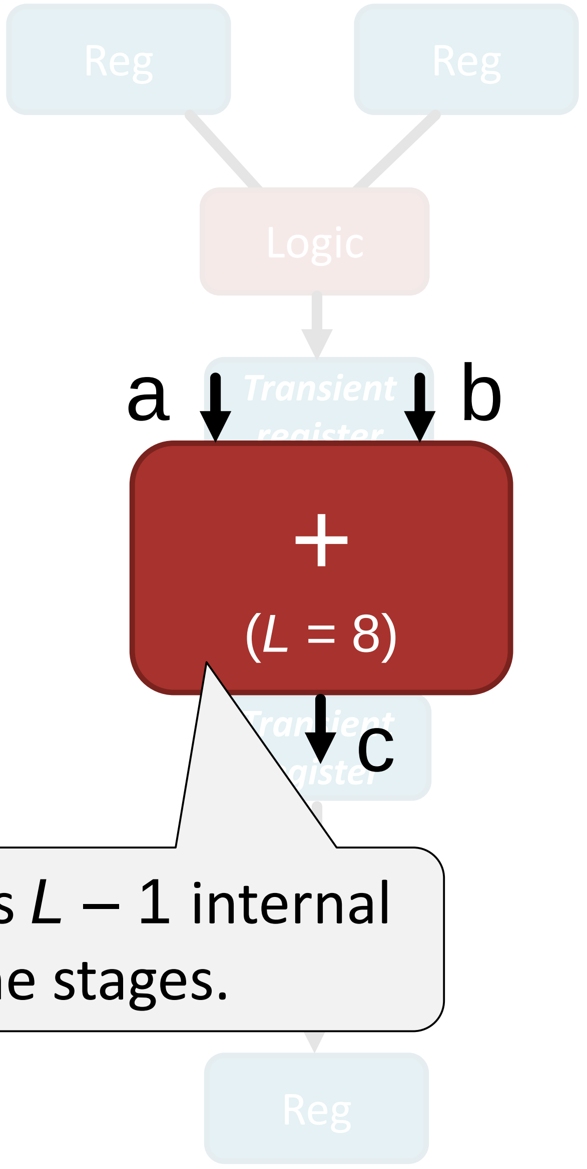
```
float c = a + b;
```

Nakayama, T. *Hardware arrangement for floating-point addition and subtraction*, 1993, US Patent 5,197,023.

Our point of view

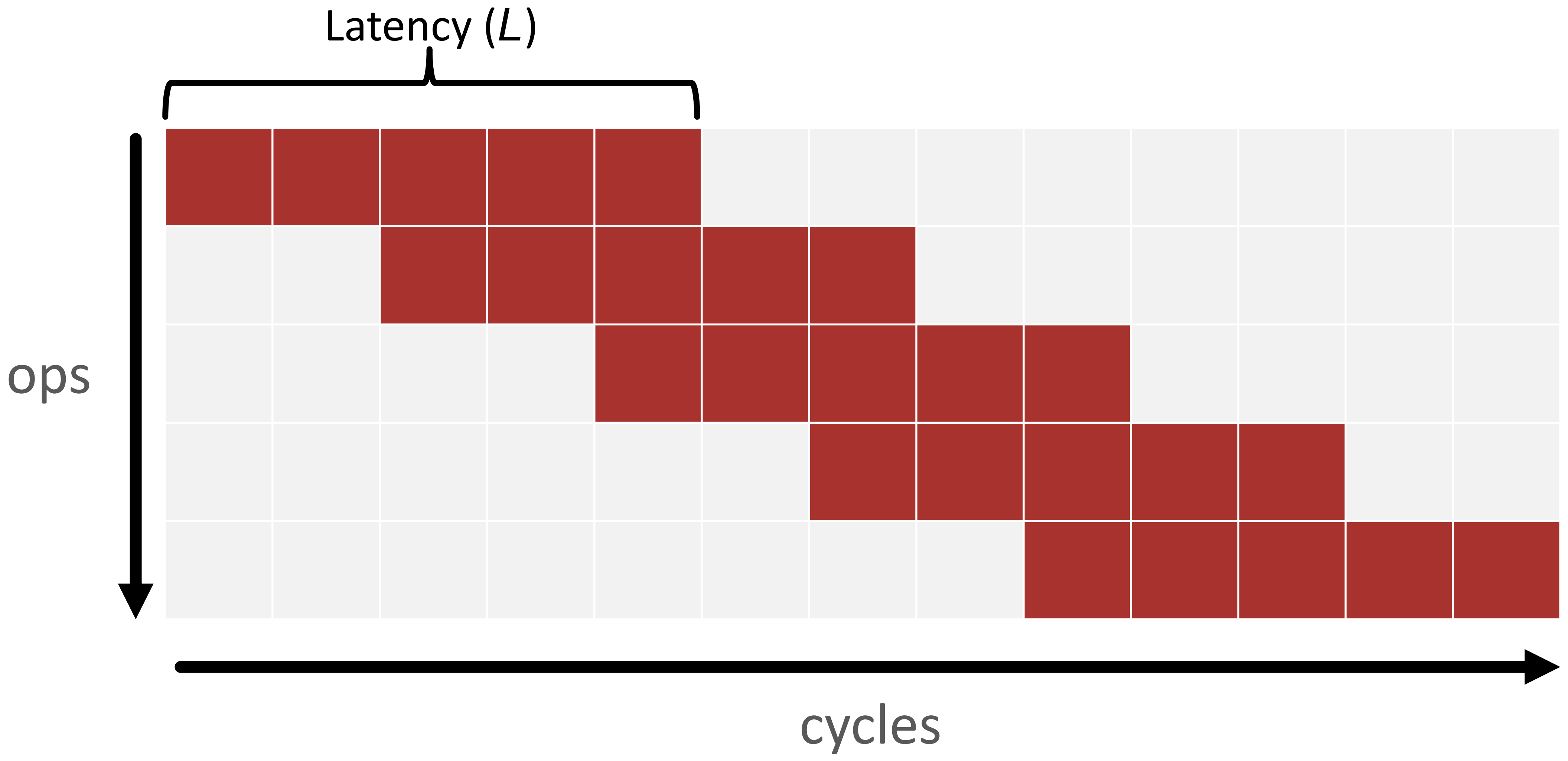
In HLS, we treat **pipelines**.

```
float c = a + b;
```

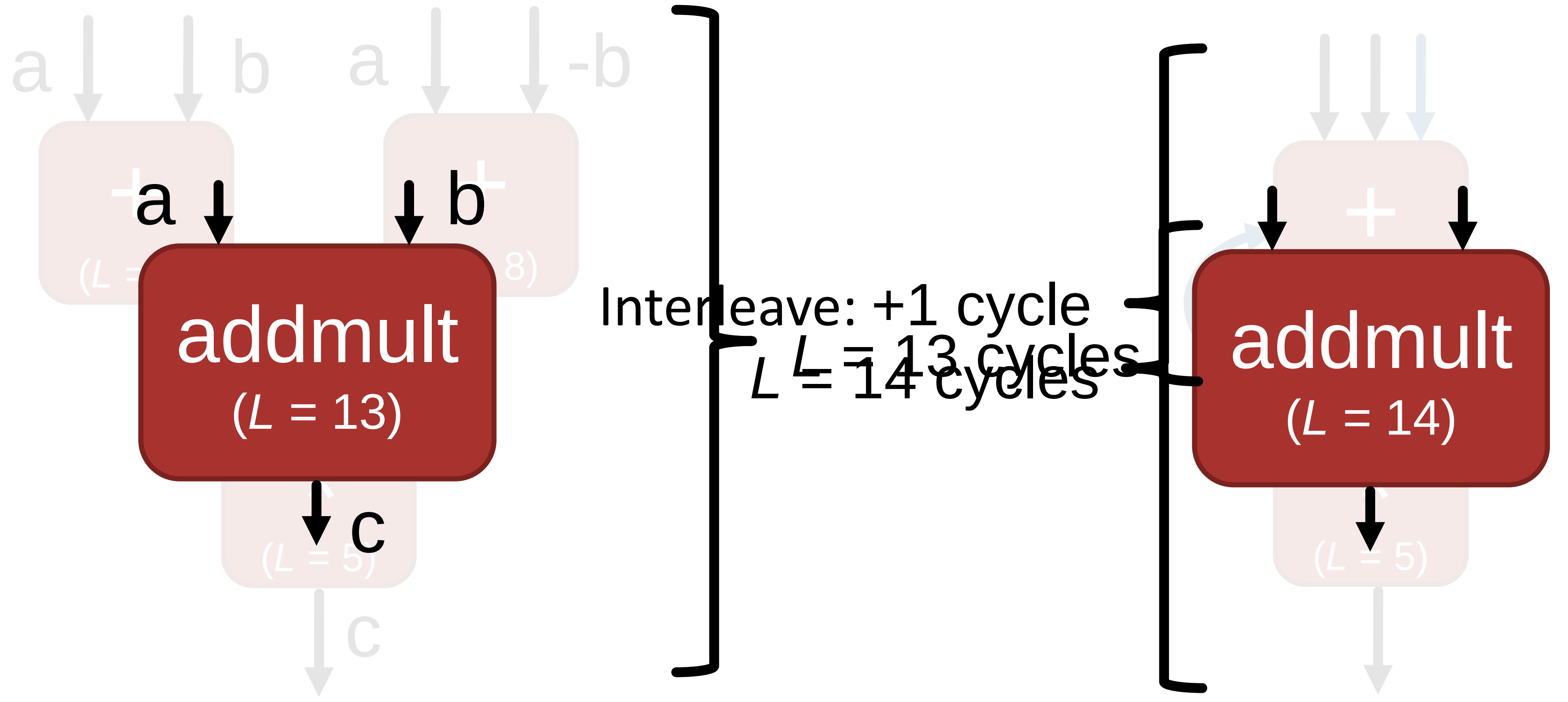


Implies $L - 1$ internal pipeline stages.

Pipelines



Multiple floating point operations



```
float c = (a + b) * (a - b);
```

≥ Two ways to implement this

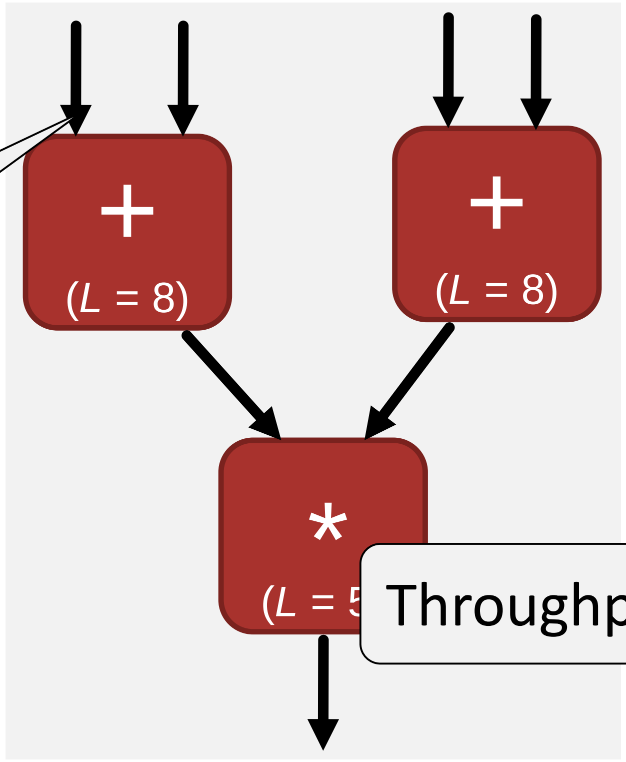
Initiation interval

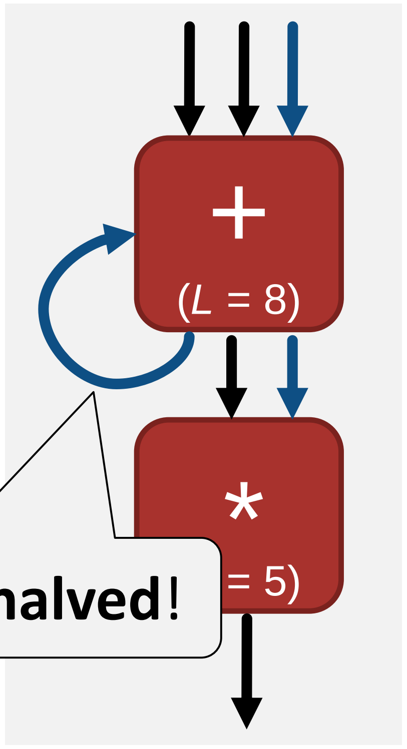
In addition to **latency** (L), we introduce the property **initiation interval** (“II”, here I).

Interpretations:

1. *No. of cycle*
   new inputs (
2. *Inverse throughput of the pipeline*
3. *Factor slowdown of your application* ☺

Can accept all four inputs in parallel



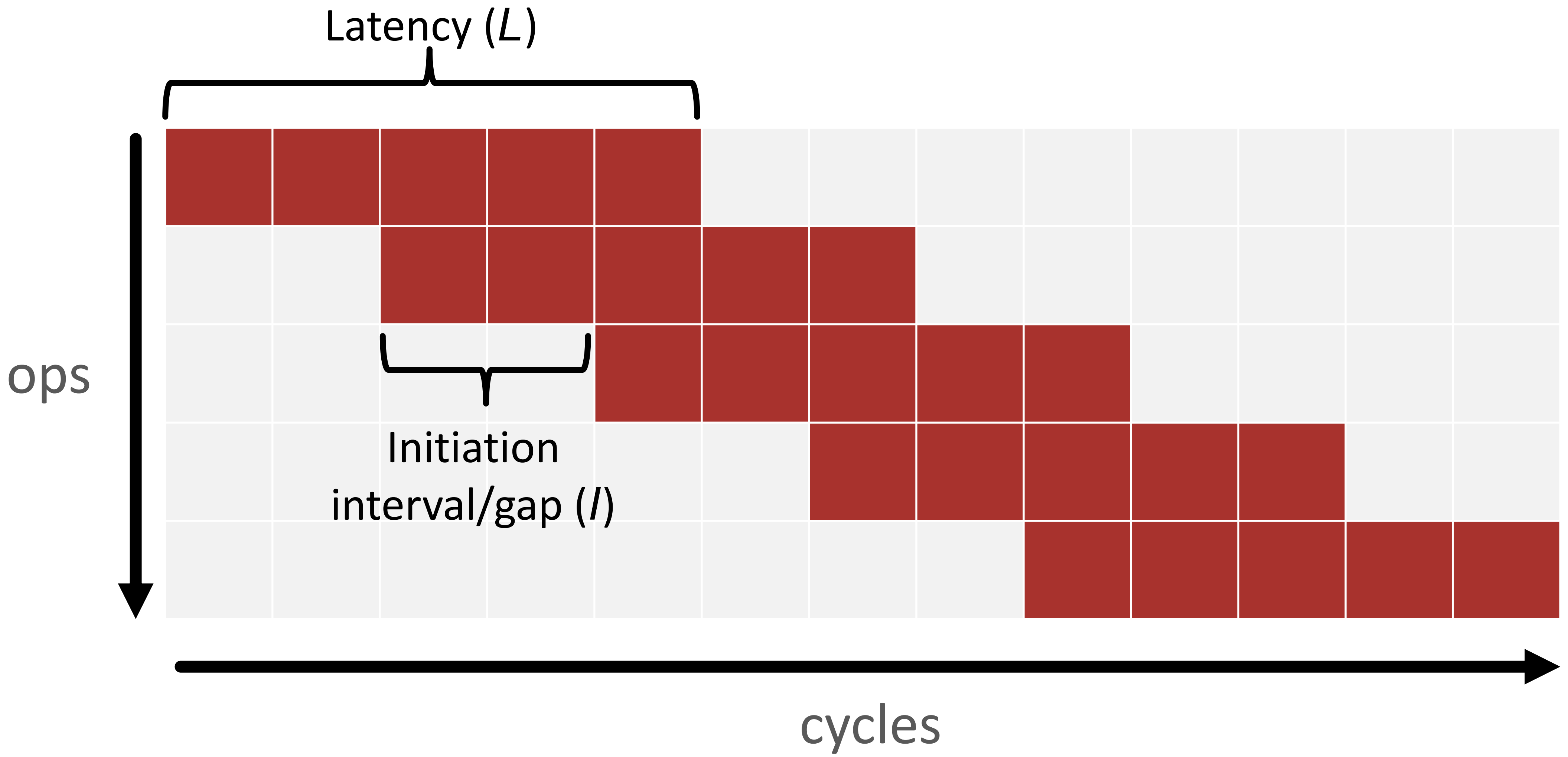


Throughput is **halved**!

L = 13 cycles
I = 1 cycle
2 adds, 1 mult
3 op/1 cycle

L = 14 cycles
I = 2 cycles
1 add, 1 mult
3 op/2 cycles

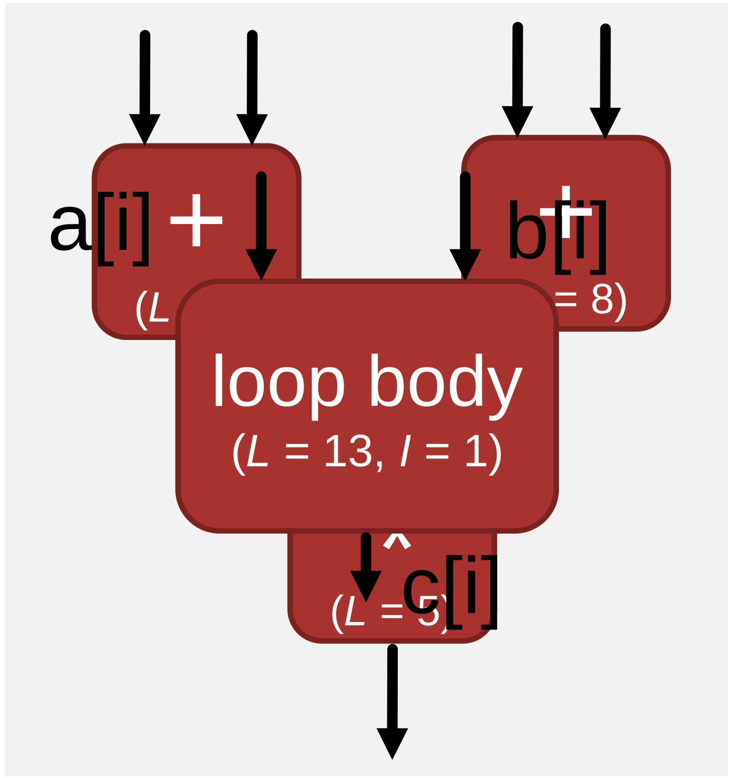
Pipelines vol. II



Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=1
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```

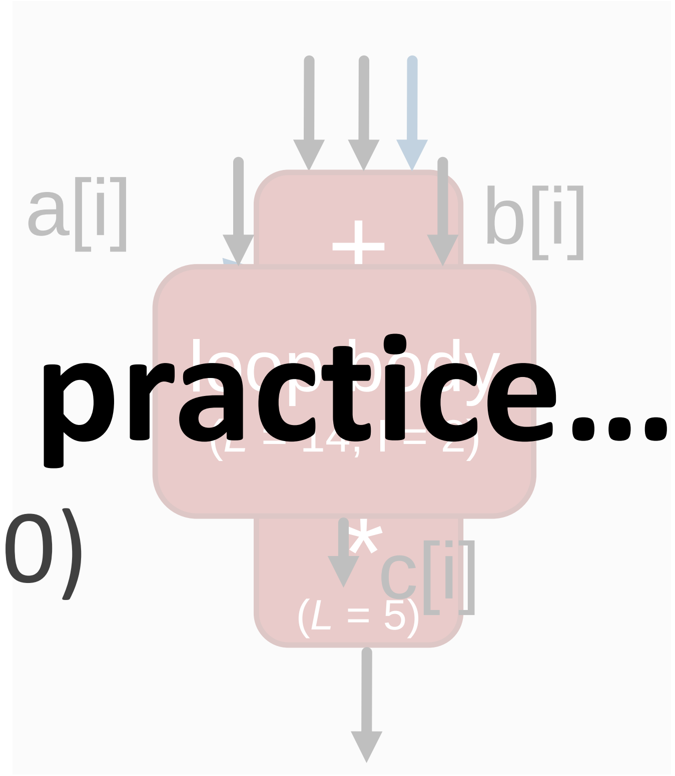
1 iteration	13 + 1 = 14 cycles
10 iterations	13 + 10 = 23 cycles
N iterations	13 + *N* cycles



Loop iterations affect the runtime **additively**, regardless of body content

Adding loops

```
for (int i = 0; i < N; ++i) {
  #pragma HLS PIPELINE II=2
  c[i] = (a[i] + b[i]) *
         (a[i] - b[i]);
}
```



Let's see this in practice…

(example 0)

Generally:

$$L_{\text{tot}} = L + I \cdot N$$

Initiation interval paid at *every iteration*

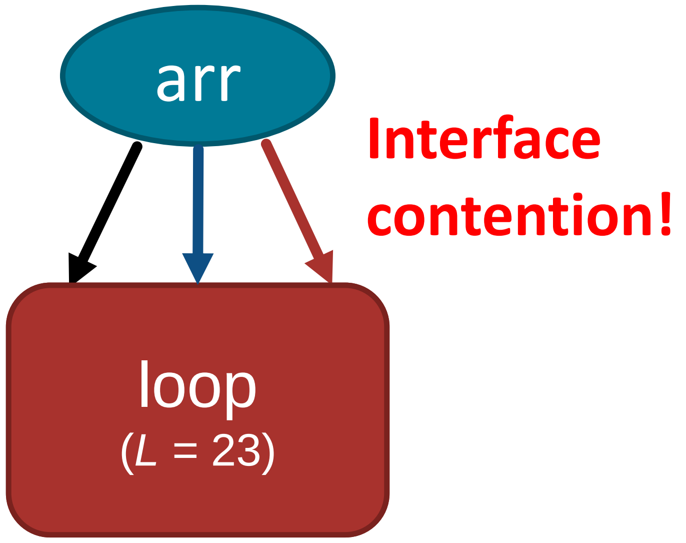
Pipeline stalls in practice

Why do we worry so much? Just set *I* = 1…

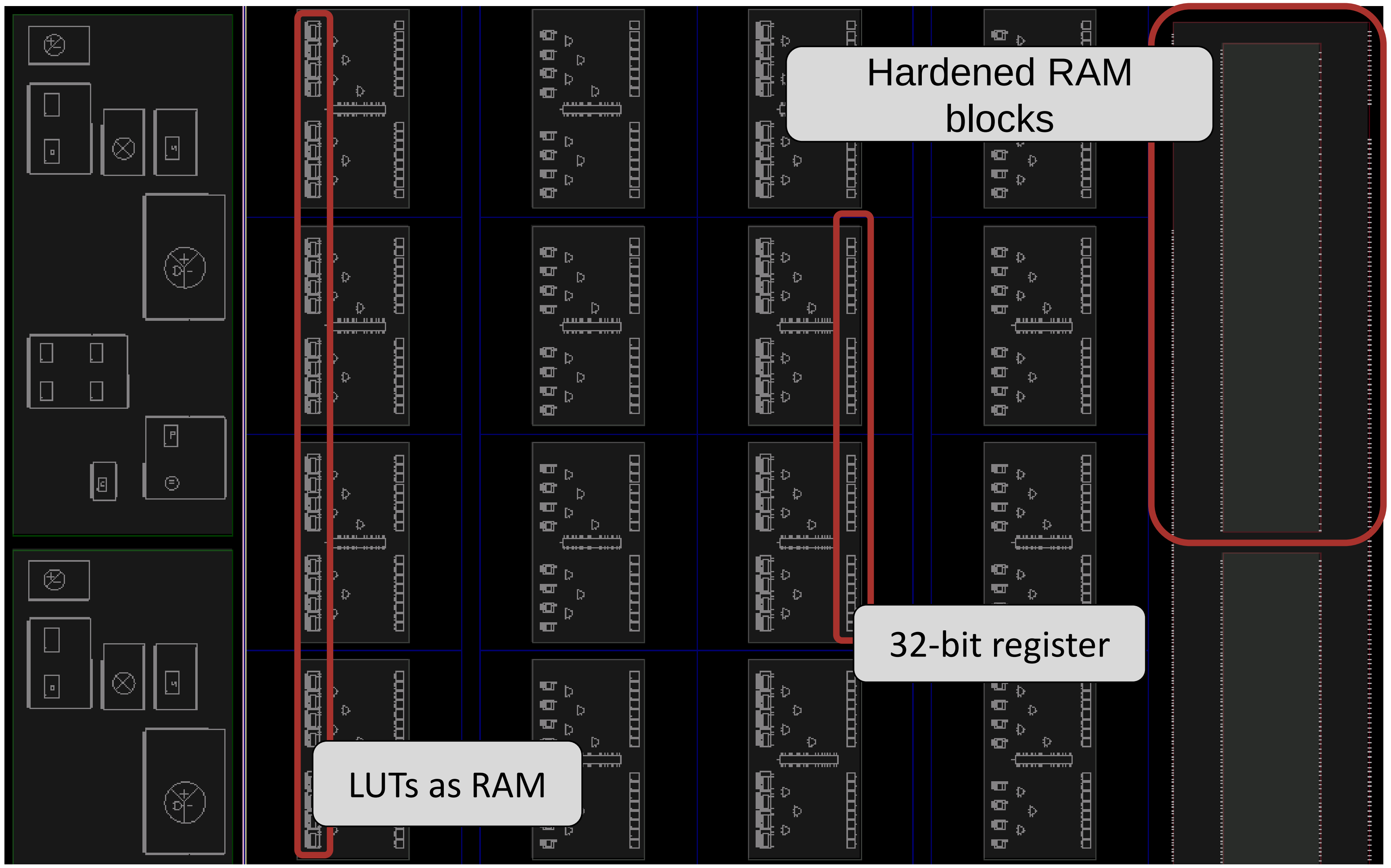
Increased II is commonly found in the wild:

1. Intra-iteration (now):
   - *Multiple accesses to the same interface*
2. Inter-iteration (later)
   - *Data dependencies*
3. Low throughput requirements (rare)
   - *e.g. input only received every 16 cycles*

```
for (int i = 1; i < N - 1; ++i) {
  #pragma HLS PIPELINE ~~II=1~~ II=3
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```

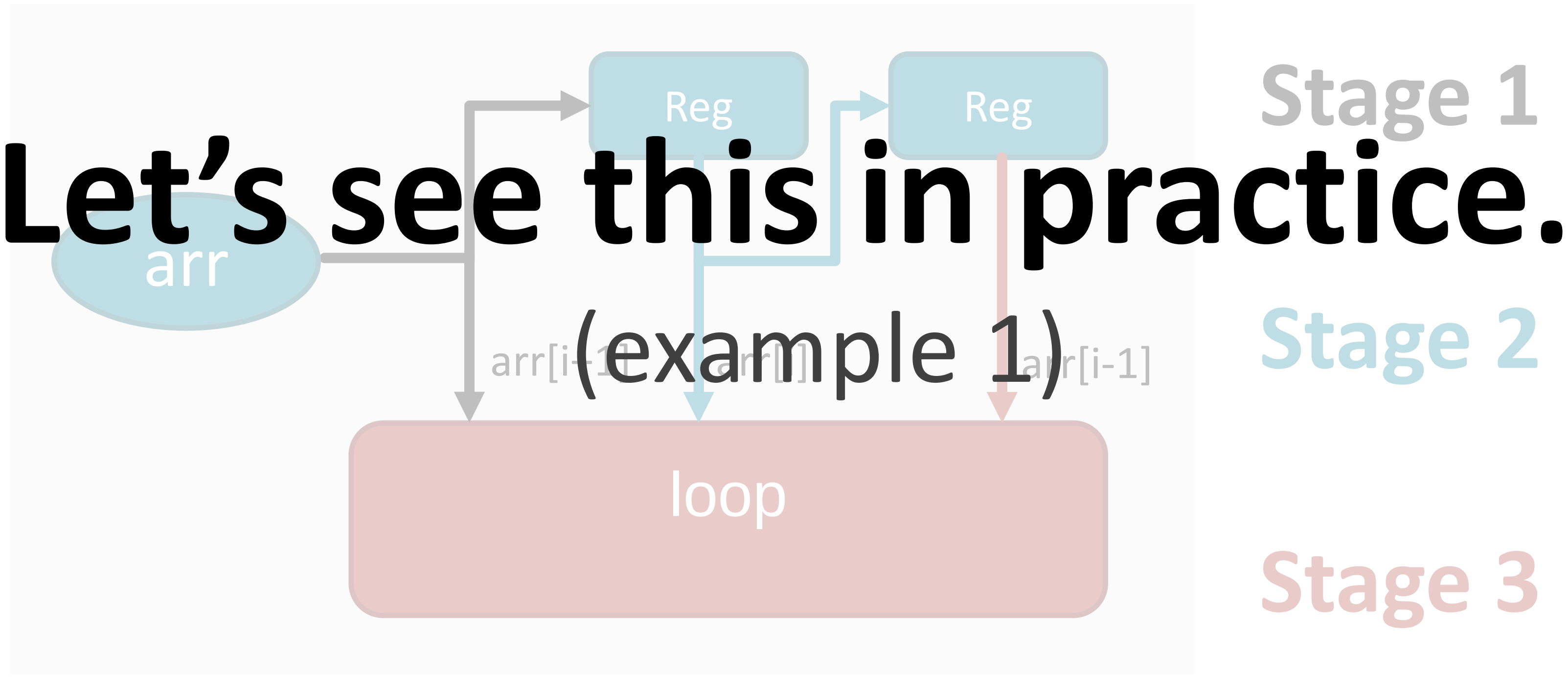


Fast memory everywhere



Inserting registers

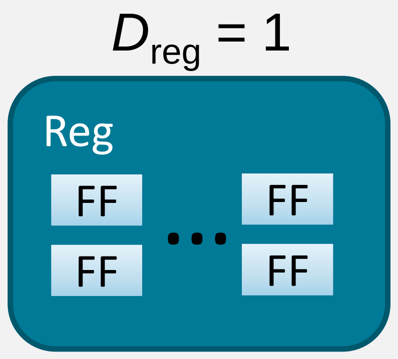
```
for (int i = 1; i < N - 1; ++i) {
  res[i] = 0.3333 *
    (arr[i-1] + arr[i] + arr[i+1]);
}
```



Let's see this in practice…

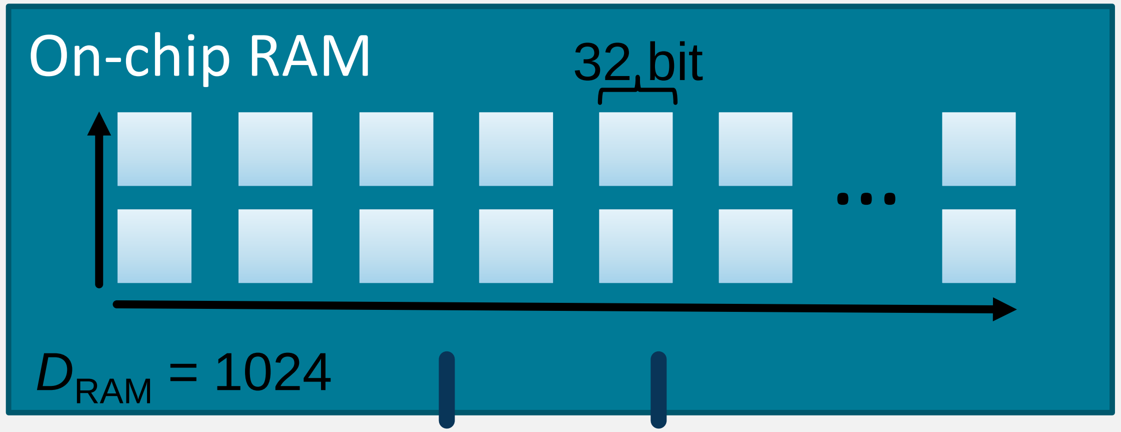
(example 1)

≥ Two classes of storage



$W_{\text{reg}} = 32$ bit

4 Byte = 32 bit · 1



$W_{\text{RAM}} = 32$ bit

4096 Byte = 32 bit · 1024
(example configuration of Xilinx BRAM)

Useful to think of memory in terms of **depth** (*D*) and **width** (*W*)

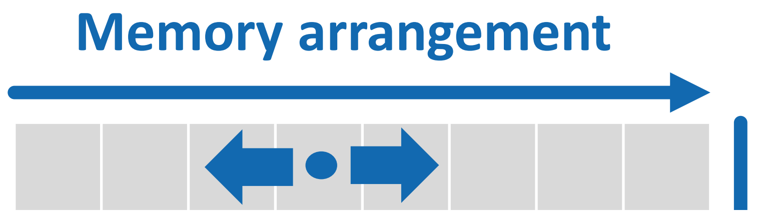
Buffer depth

1D stencil program:

$W = 2, D = 1$

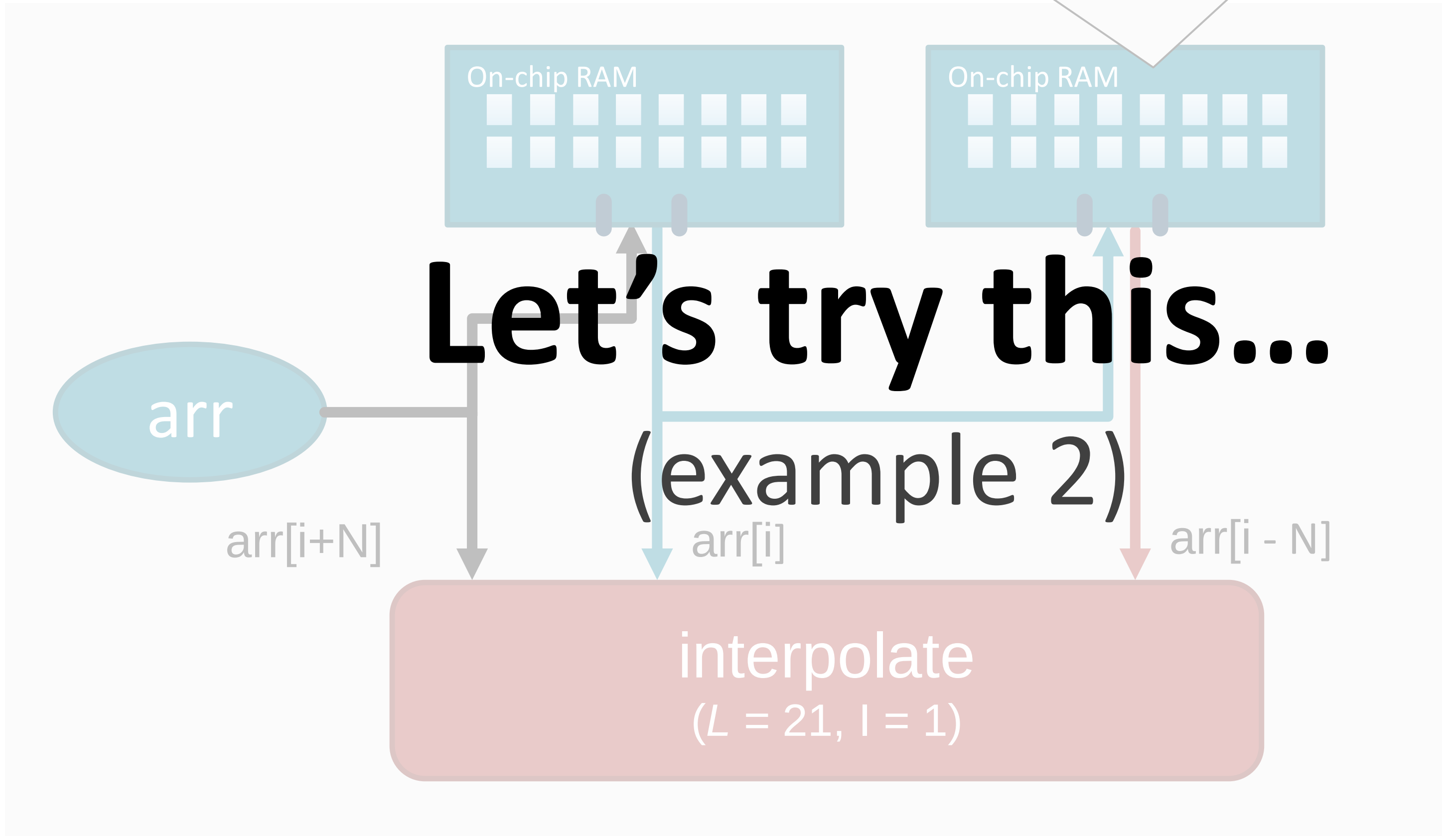
2D row-major:

$W = 2, D = N$



Modified example

W unchanged, *D* changed from 1 to *N*



Stage 1

Stage 2

Stage 3

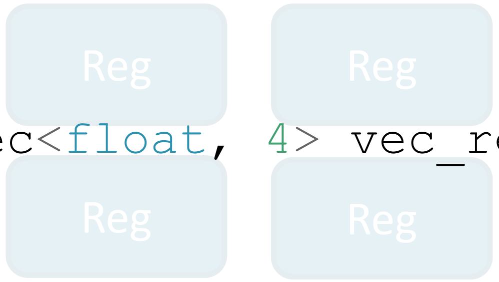
Let's try this…

(example 2)

Thinking in width and depth

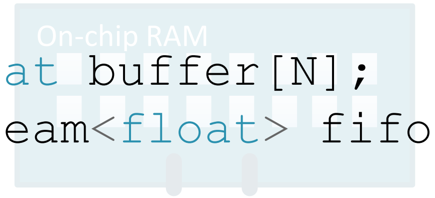
```
float reg;
```

W = 32 bit
D = 1



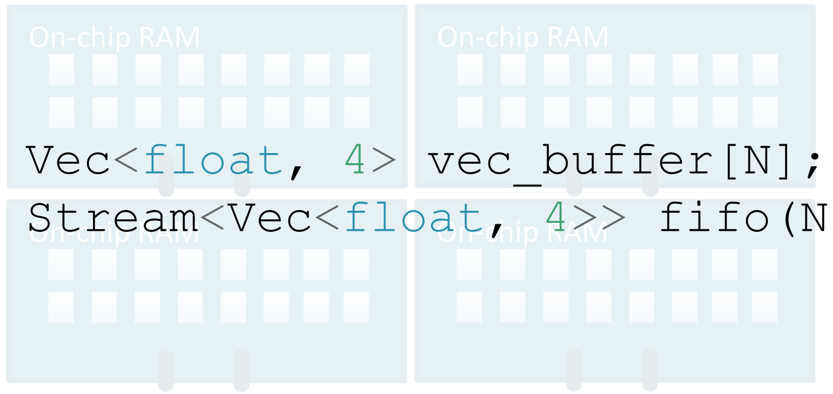
```
Vec<float, 4> vec_reg;
```

W = 128 bit
D = 1



```
float buffer[N];
Stream<float> fifo(N);
```

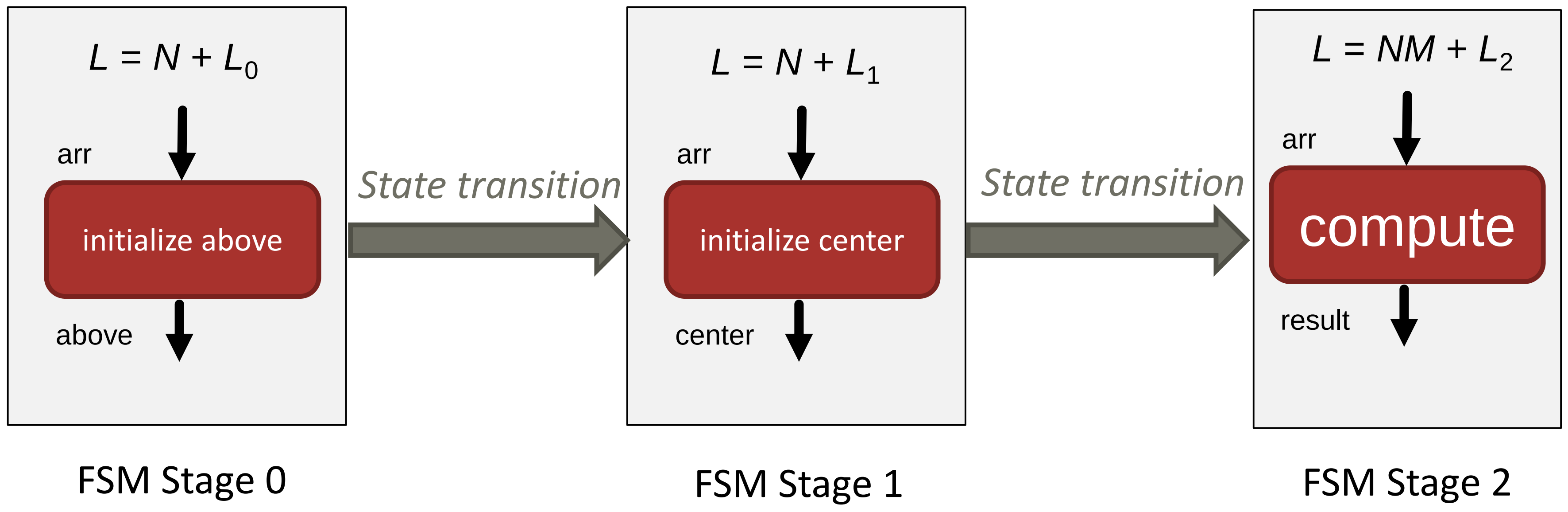
W = 32 bit
D = N



```
Vec<float, 4> vec_buffer[N];
Stream<Vec<float, 4>> fifo(N);
```

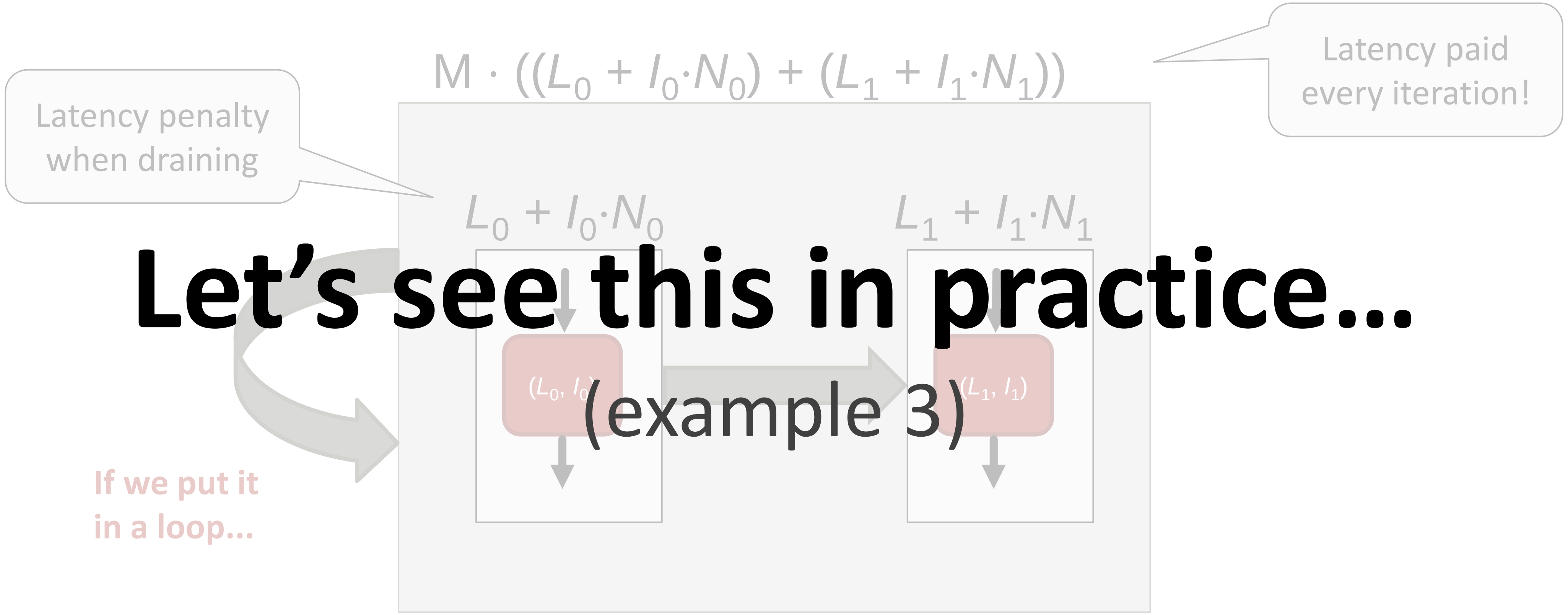
W = 128 bit
D = N

Finite state machine



Sequential execution of (pipeline-)**parallel** stages

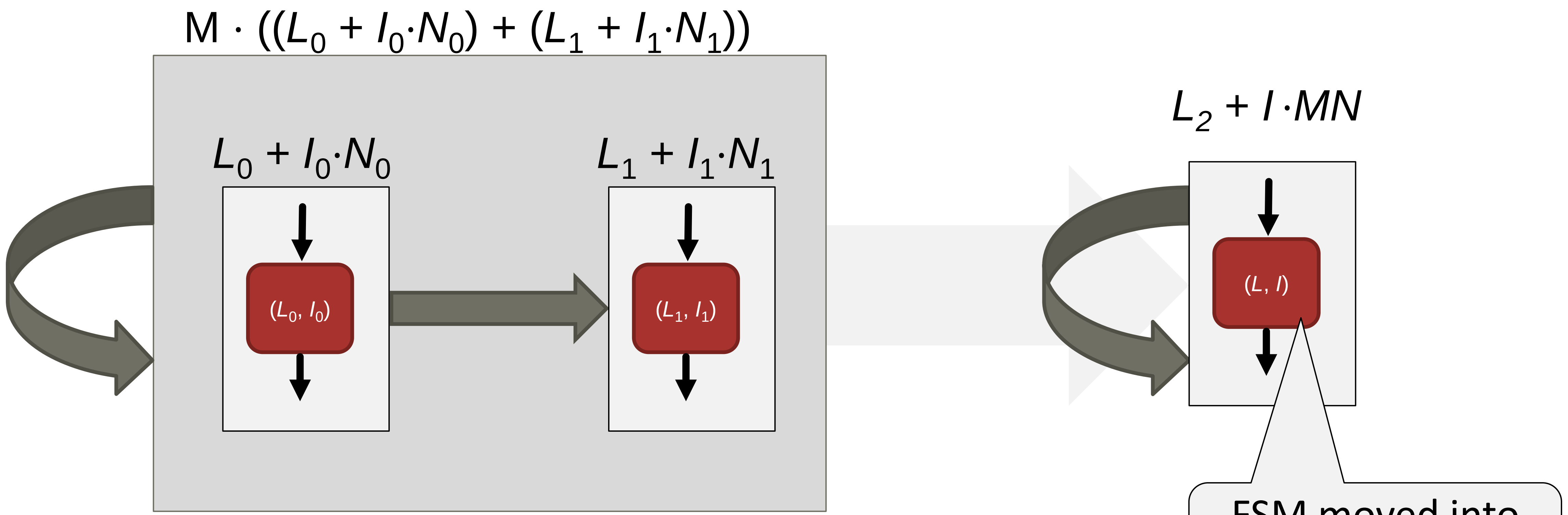
Draining and initialization phases


Latency penalty when draining
$M \cdot ((L_0 + I_0 \cdot N_0) + (L_1 + I_1 \cdot N_1))$
Latency paid every iteration!
$L_0 + I_0 \cdot N_0$
$L_1 + I_1 \cdot N_1$
(L_0, I_0)
(L_1, I_1)
If we put it in a loop…

Let’s see this in practice…

(example 3)

Flattening



FSM moved into the pipeline

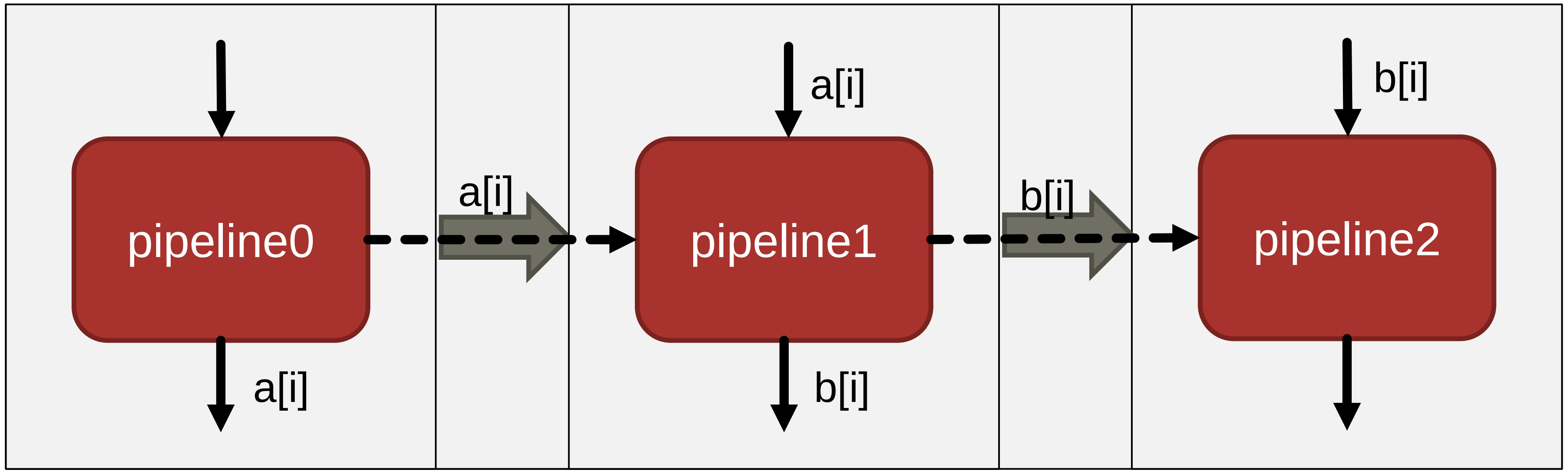
We will see an example of this shortly

Multiple concurrent pipelines

$$(L_0 \cdot N_0) + (L_1 \cdot N_1) + (L_2 \cdot N_2)$$

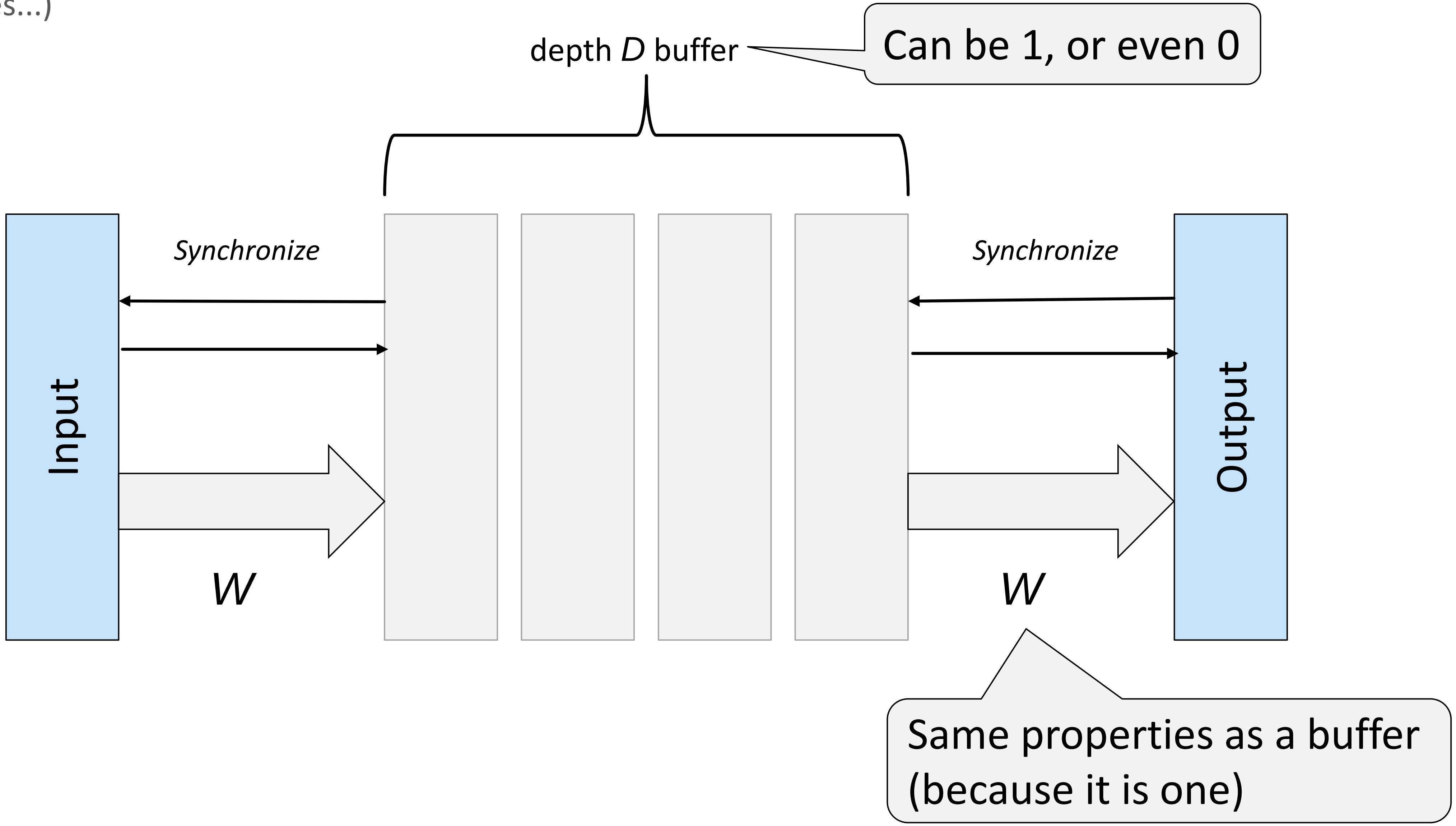
$$(L_0 + N_0) + (L_1 + N_1) + (L_2 + N_2)$$

$$(L_0 + L_1 + L_2 \cdot N_2)$$

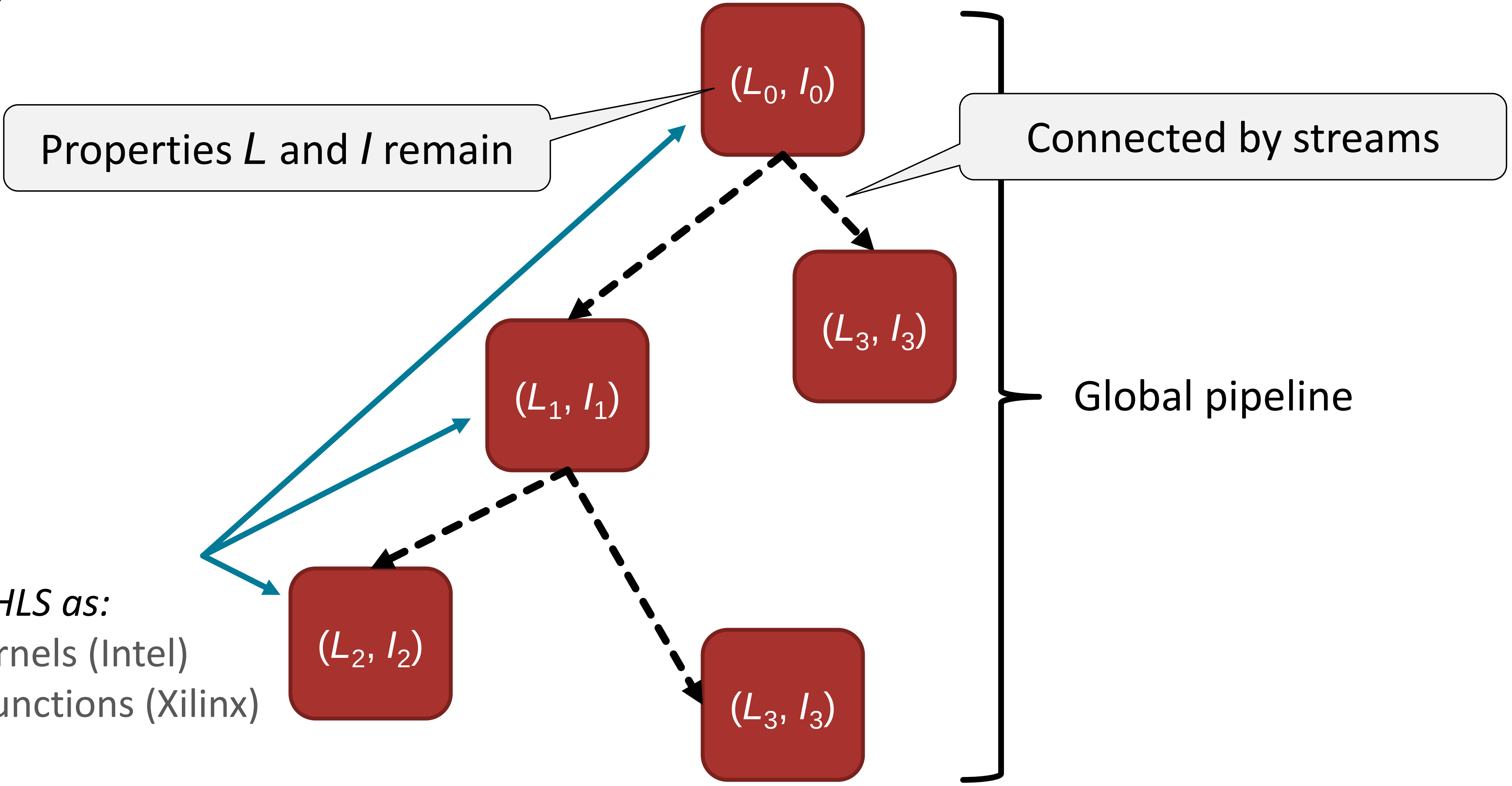


Streams

(FIFOs, queues...)

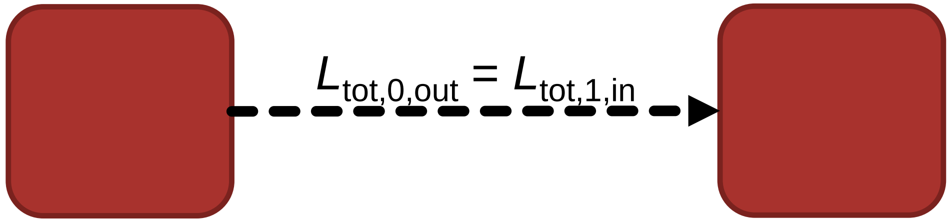


Processing elements

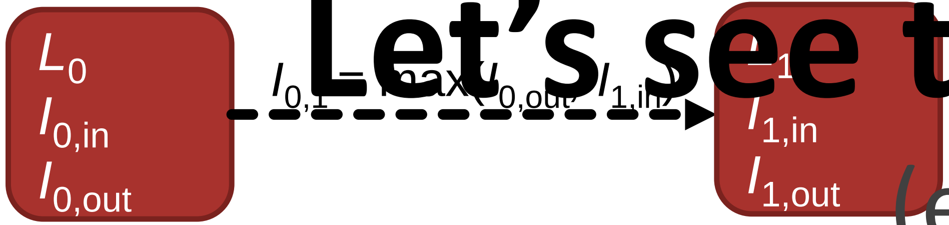


Expressed in HLS as:
- OpenCL kernels (Intel)
- Dataflow functions (Xilinx)

Properties of the global pipeline



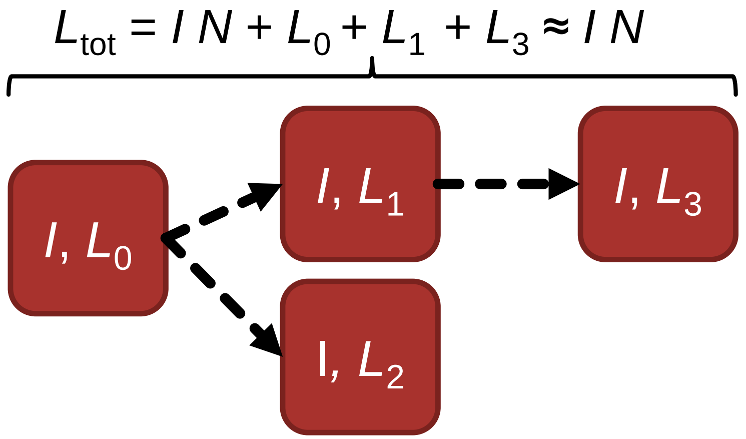
What goes in must come out:
Every stream write needs a corresponding read



Let's see this in practice…

(example 4)

Slow dominates:
Can only stream with the highest initiation interval (producer/consumer) at each interface



Depth is "free":
In a perfect pipeline for large *N*, the influence of pipeline latency is negligible w.r.t. the total runtime

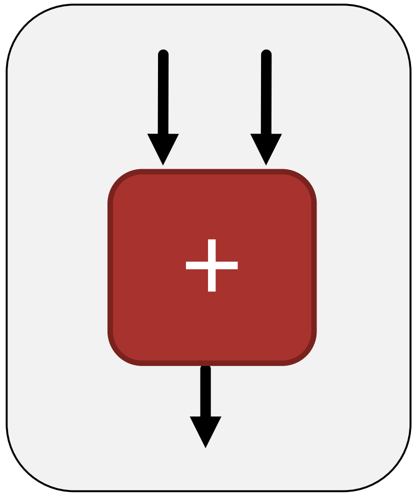
Unrolling

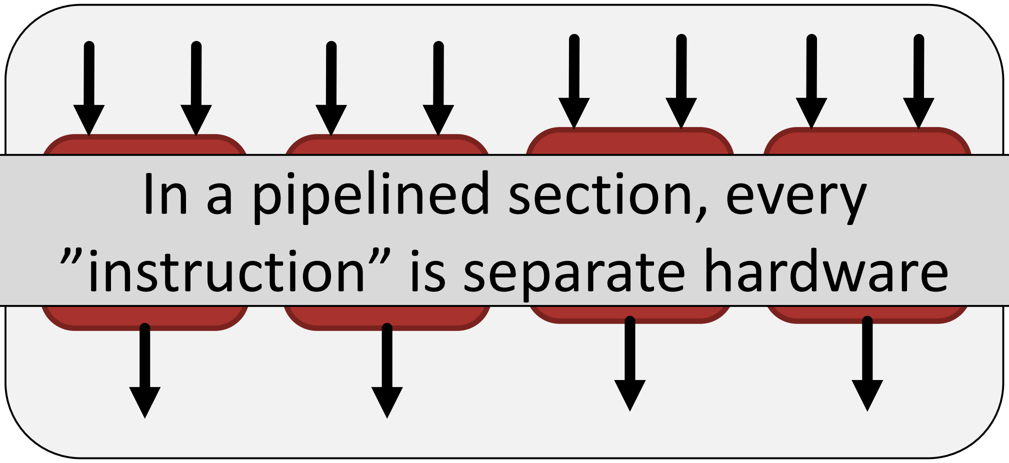
Much stronger meaning on FPGA than for an instruction-based architecture.

```
for (int w = 0; w < 4; ++w) {
  #pragma HLS PIPELINE II=1
  res[w] = a[w] + b[w];
}
```

```
#pragma HLS PIPELINE II=1
for (int w = 0; w < 4; ++w) {
  #pragma HLS UNROLL
  res[w] = a[w] + b[w];
}
```

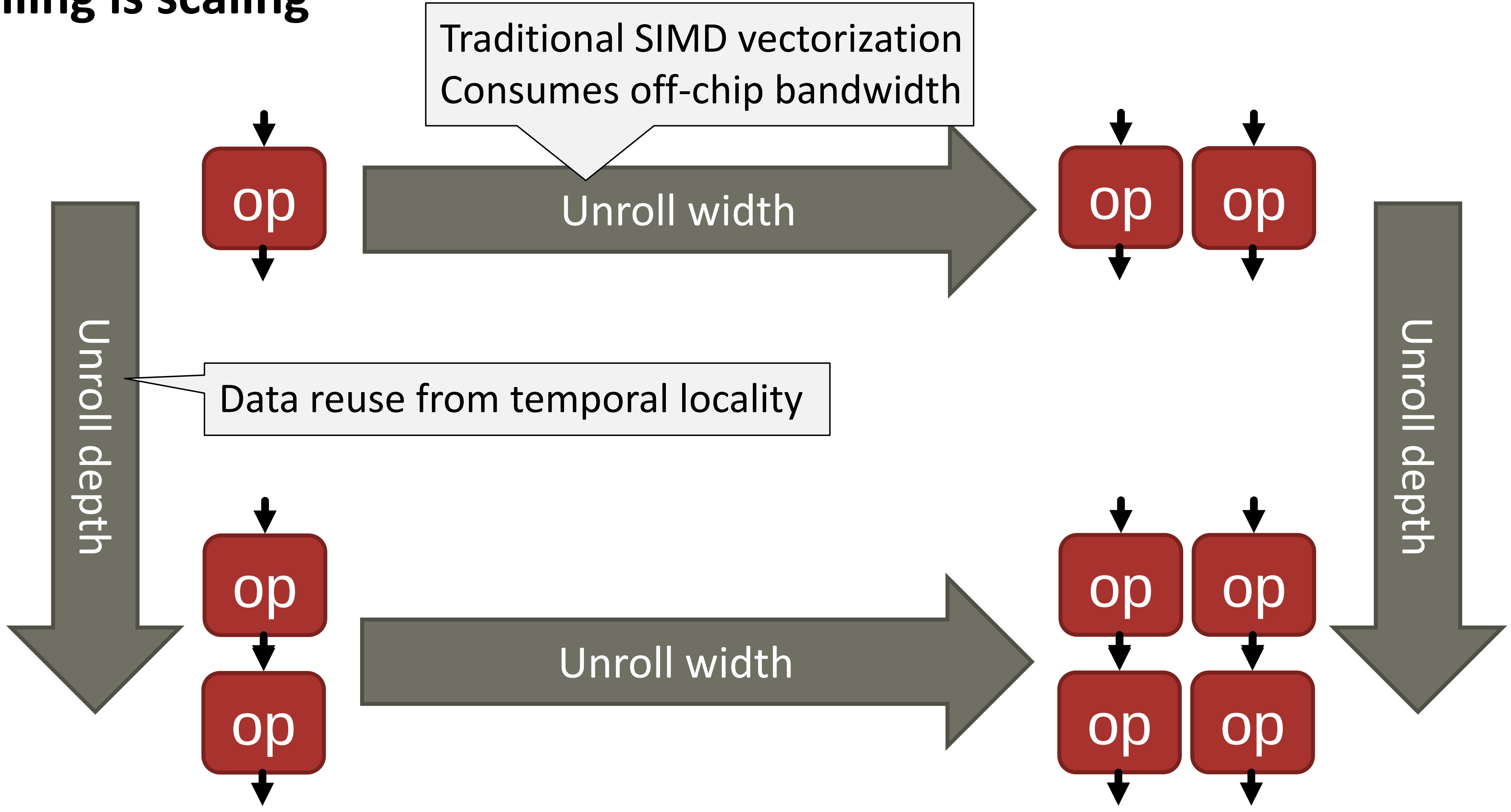
```
#pragma HLS PIPELINE II=1
res[0] = a[0] + b[0];
res[1] = a[1] + b[1];
res[2] = a[2] + b[2];
res[3] = a[3] + b[3];
```





In a pipelined section, every ”instruction” is separate hardware

Unrolling is scaling


Traditional SIMD vectorization
Consumes off-chip bandwidth
Unroll width
Unroll depth
Data reuse from temporal locality
Unroll width
Unroll depth
op
op
op
op
op
op
op
op
op

Matrix-matrix multiplication

```
for (int n = 0; n < N; ++n)
  for (int p = 0; p < P; ++p)
    for (int m = 0; m < M; ++m)
      C[n, p] += A[n, m] * B[m, p];
```

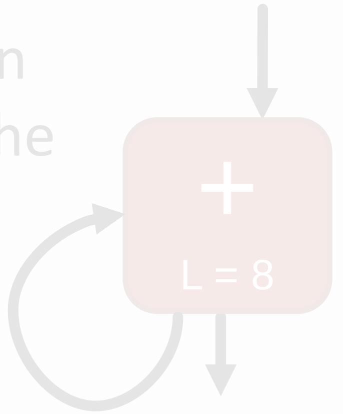
Our intuition of temporal locality does not work here!

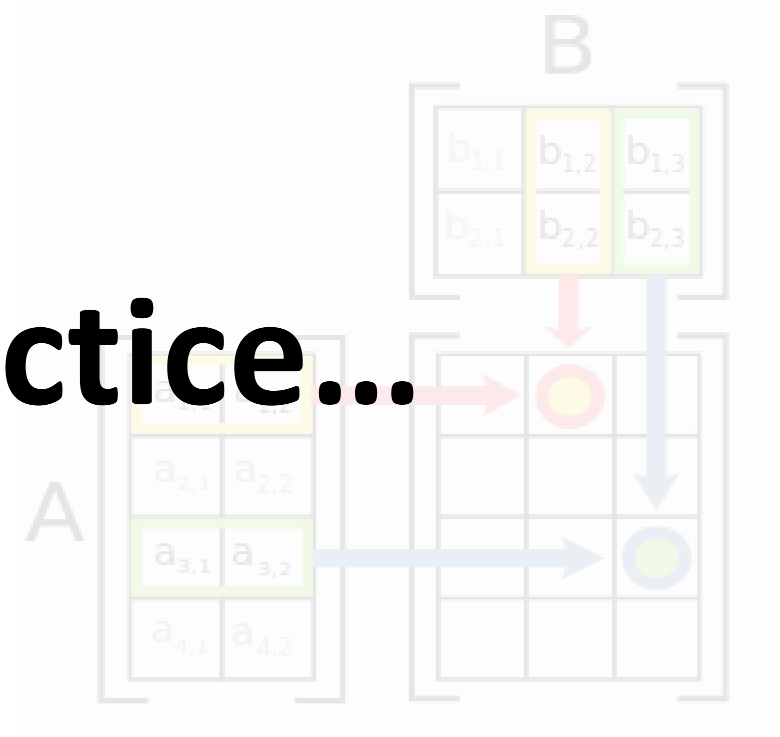
Let's see this in practice…

(example 5)

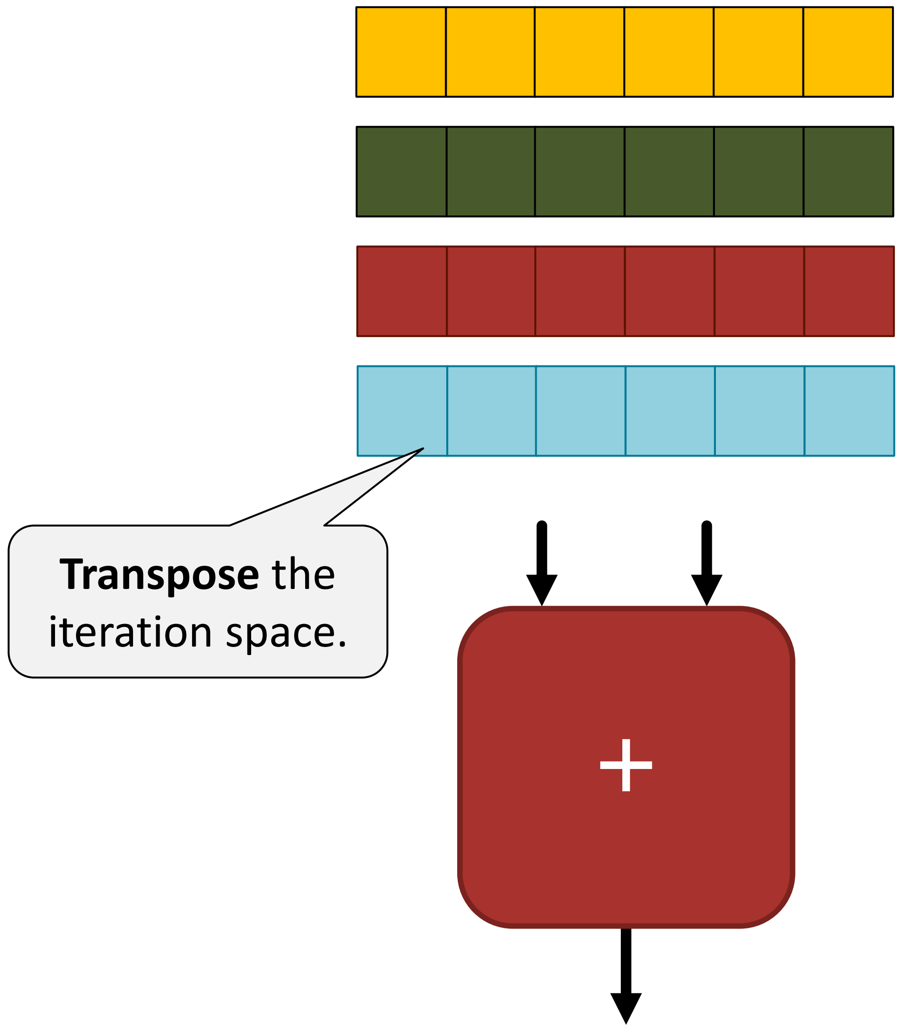
Every iteration depends on the previous:

$I = 8$





Solving loop-carried data dependencies



```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j = 0; j < M; ++j) {
    acc += ...
  }
  out[i*M+j] = acc;
```

Comes at the cost of buffer space.

```
float acc[N];
for (int j = 0; j < M; ++j) {
  for (int i = 0; i < N; ++i) {
    acc[i] += ...
  }
}
for (int i = 0; i < N; ++i) {
  out[i*M+i] = acc[i];
}
```

Only needs to be larger than the latency

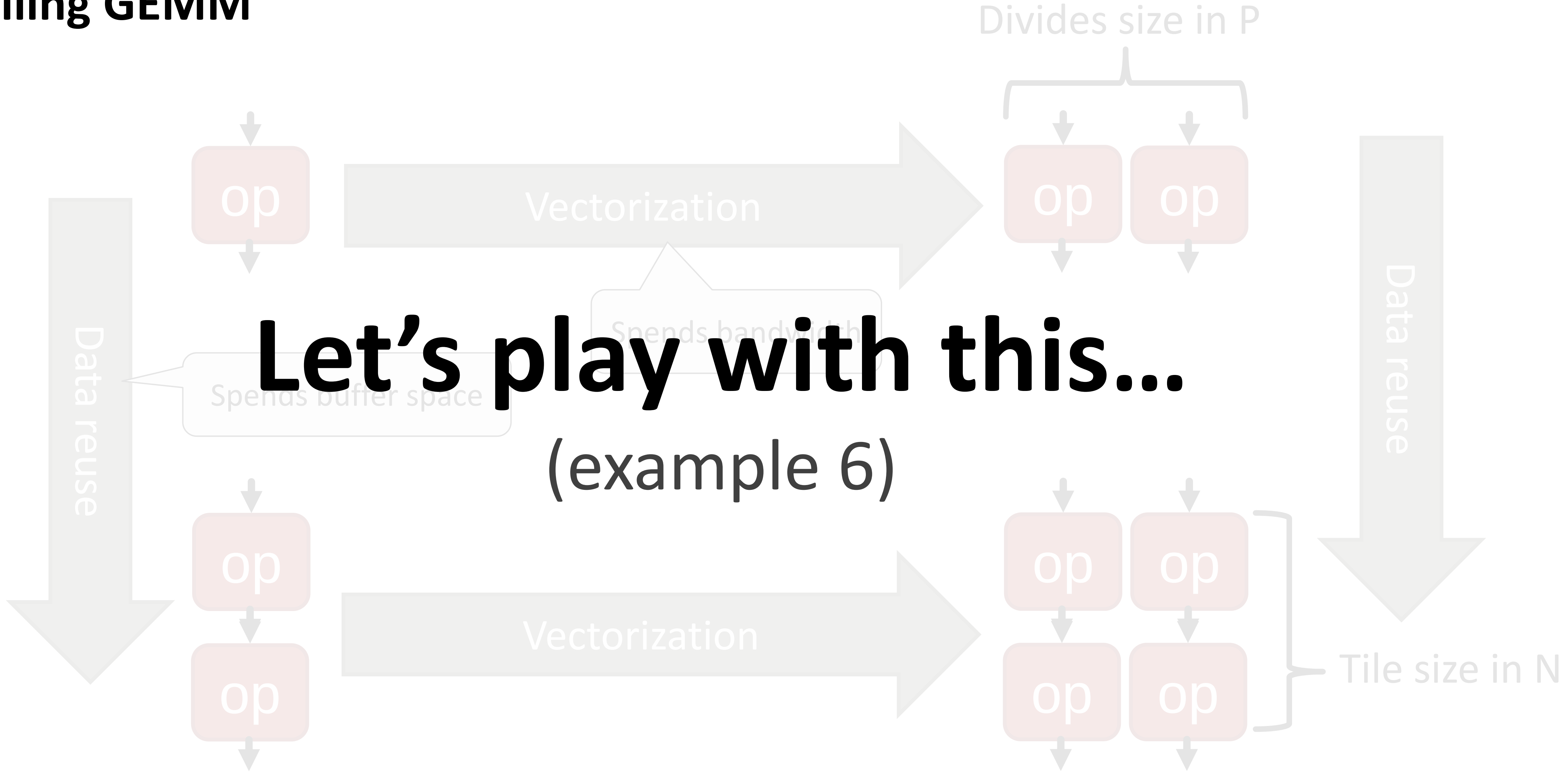
Locality in the program

```
for (int n = 0; n < N; ++n) {
  float acc[P];

  for (int k = 0; k < K; ++k) {
    const auto a = A[n*K + k];

    for (int m = 0; m < M; ++m) {
      #pragma HLS PIPELINE II=1
      const float prev = (k == 0) ? 0 : acc[m];
      acc[m] = prev + a * B[k*M + m];
    }
  }
  // ...
}
```

Temporal locality: Reused P times. Load more of these and tile N!

Spatial locality: Vectorizable.

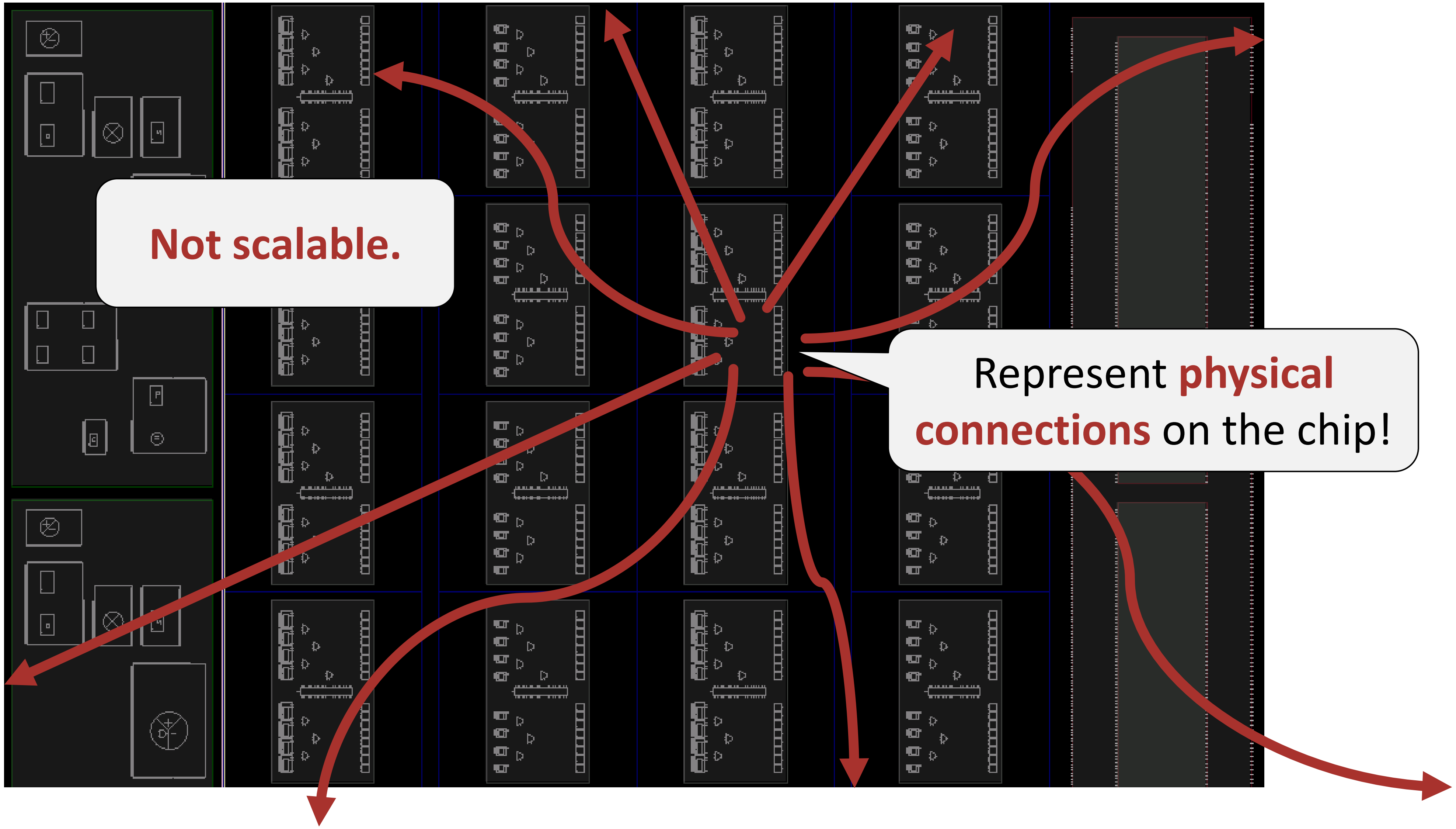
Unrolling GEMM



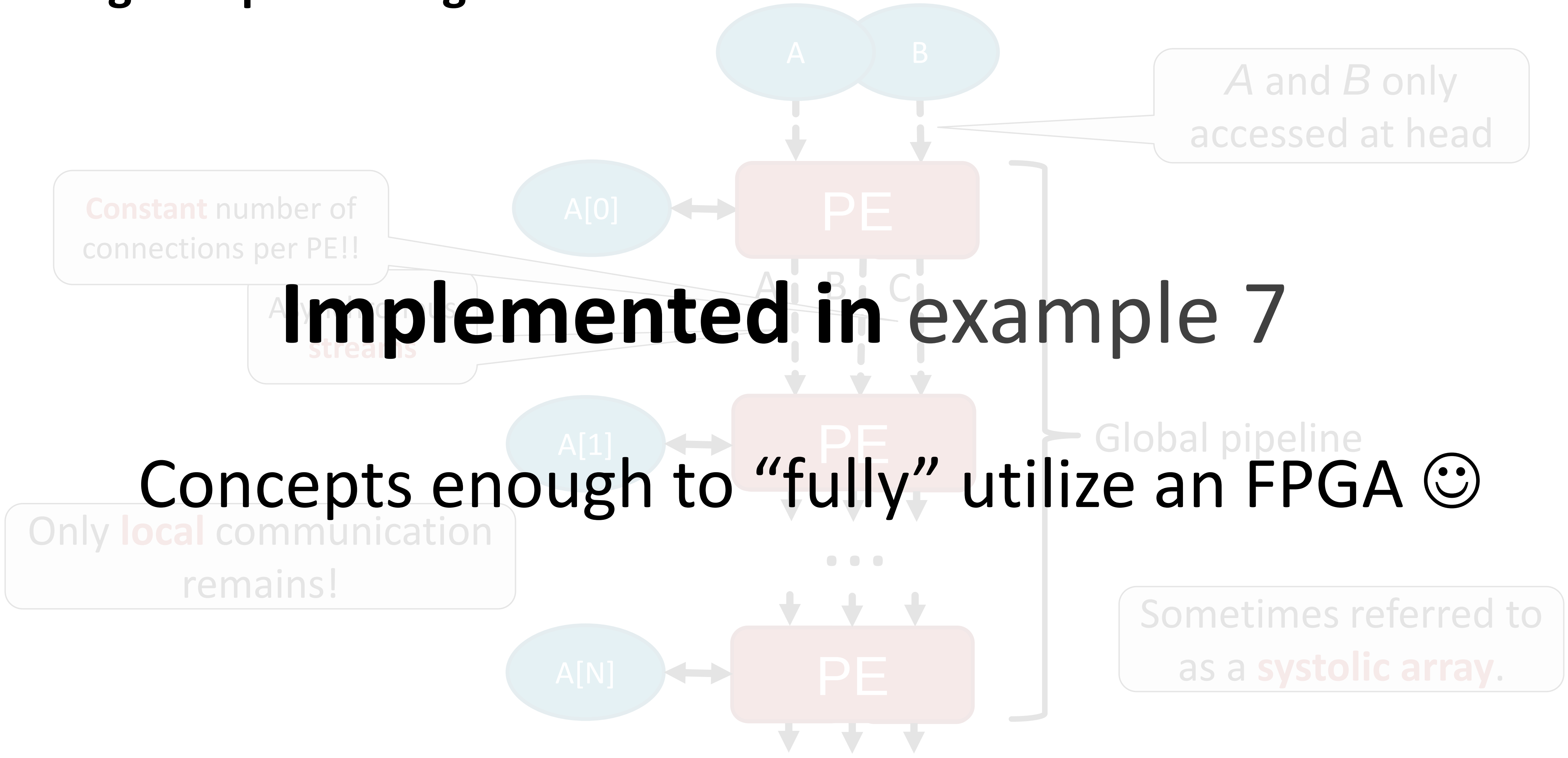
Let’s play with this…

(example 6)

Fanout issue



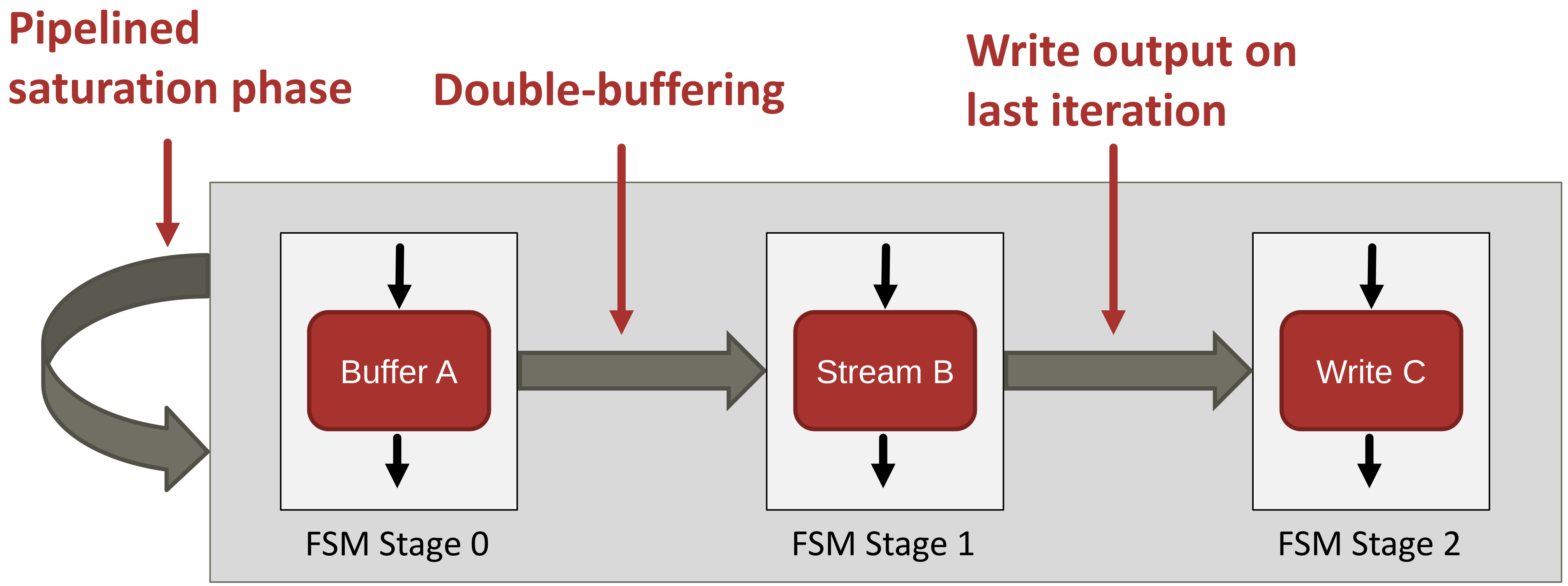
Breaking into processing elements


A
B
A and B only accessed at head
A[0]
PE
Constant number of connections per PE!!
A
B
C
streams
A[1]
PE
Global pipeline
Only local communication remains!
A[N]
PE
Sometimes referred to as a systolic array.

Implemented in example 7

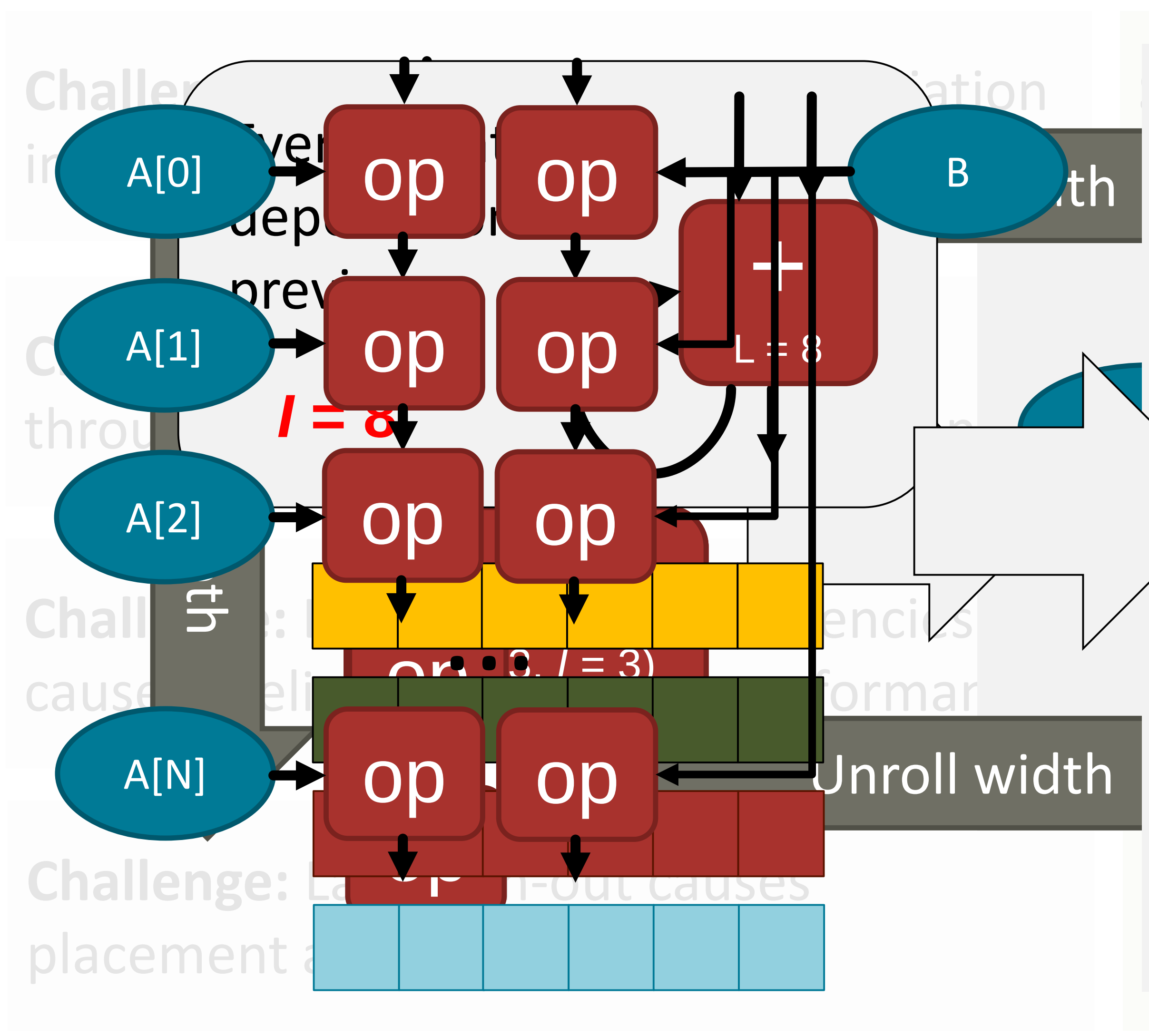
Concepts enough to “fully” utilize an FPGA ☺

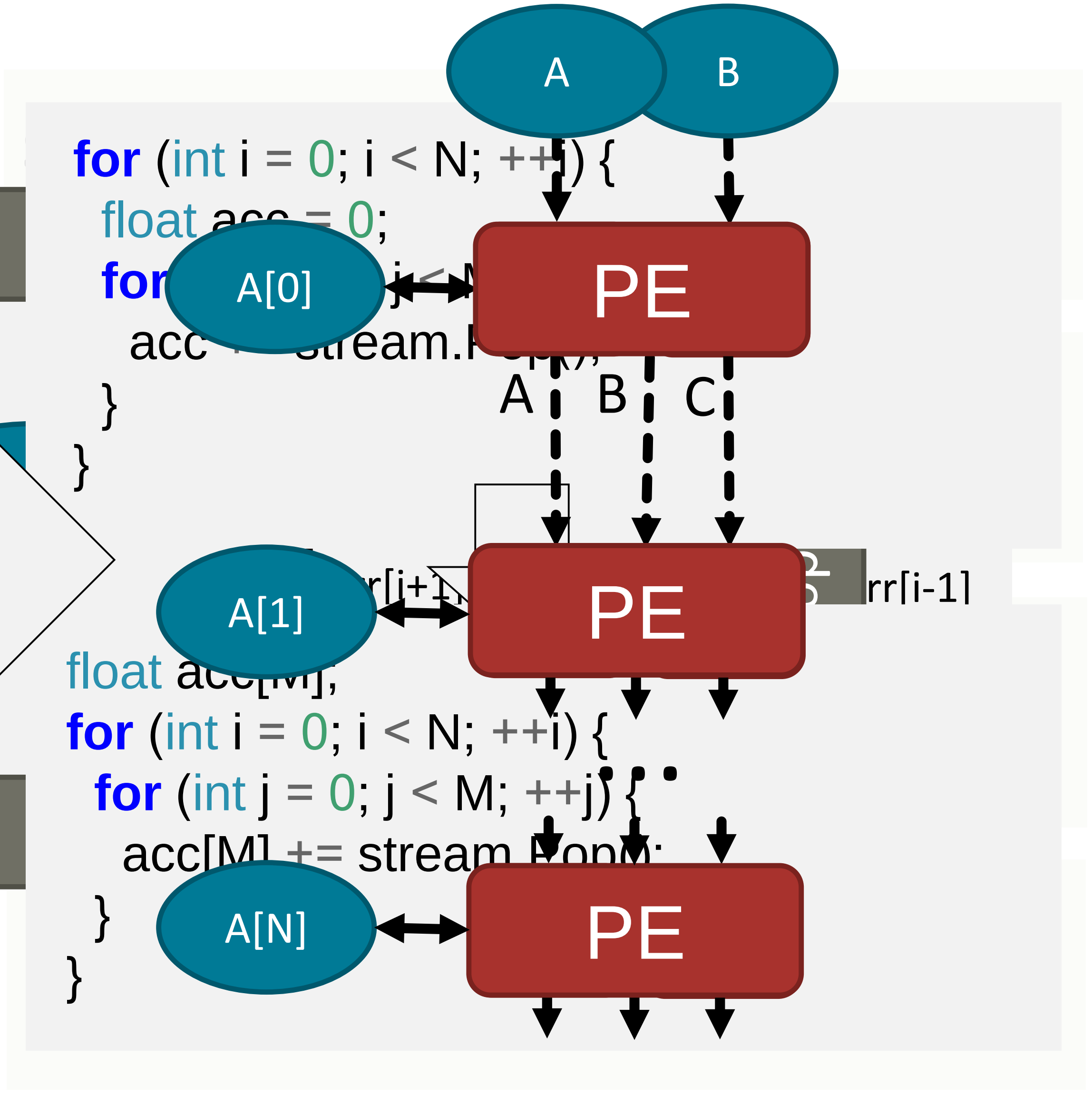
Remaining optimization potential



Fully optimized implementation at: https://github.com/spcl/gemm_hls

Summary





```
for (int i = 0; i < N; ++i) {
  float acc = 0;
  for (int j ...
    acc += stream.Pop();
  }
}
```

```
float acc[M];
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < M; ++j) {
    acc[M] += stream.Pop();
  }
}
```

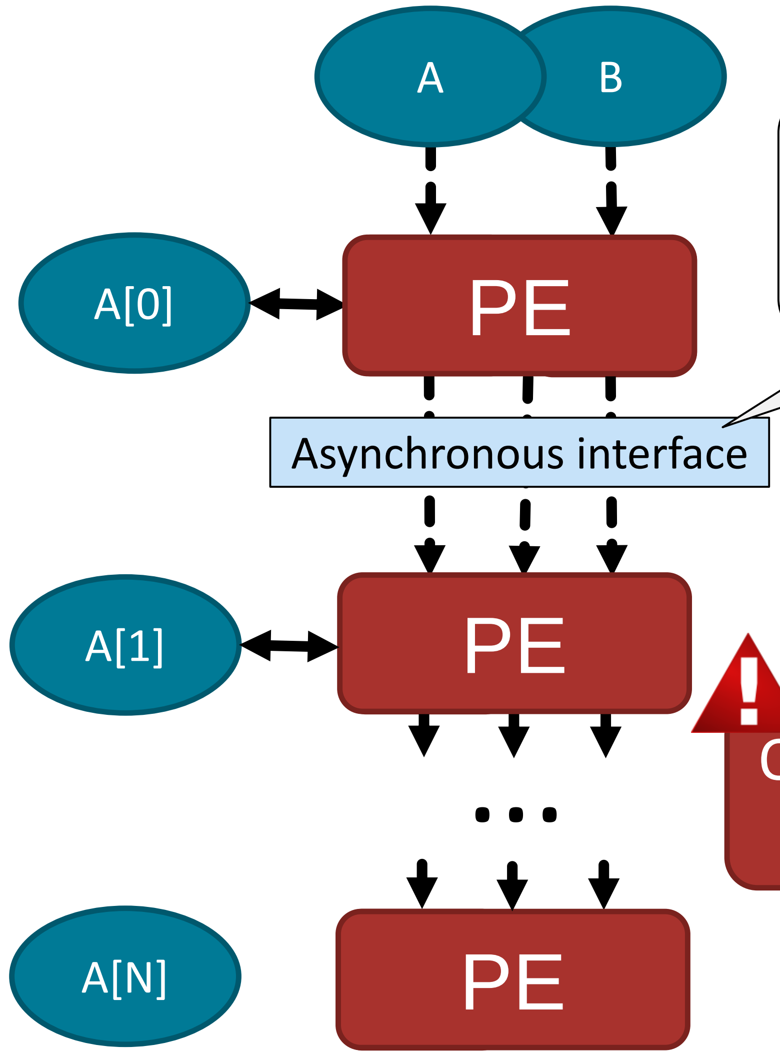
Thank you for your attention ☺

Reach out at: definelicht@inf.ethz.ch

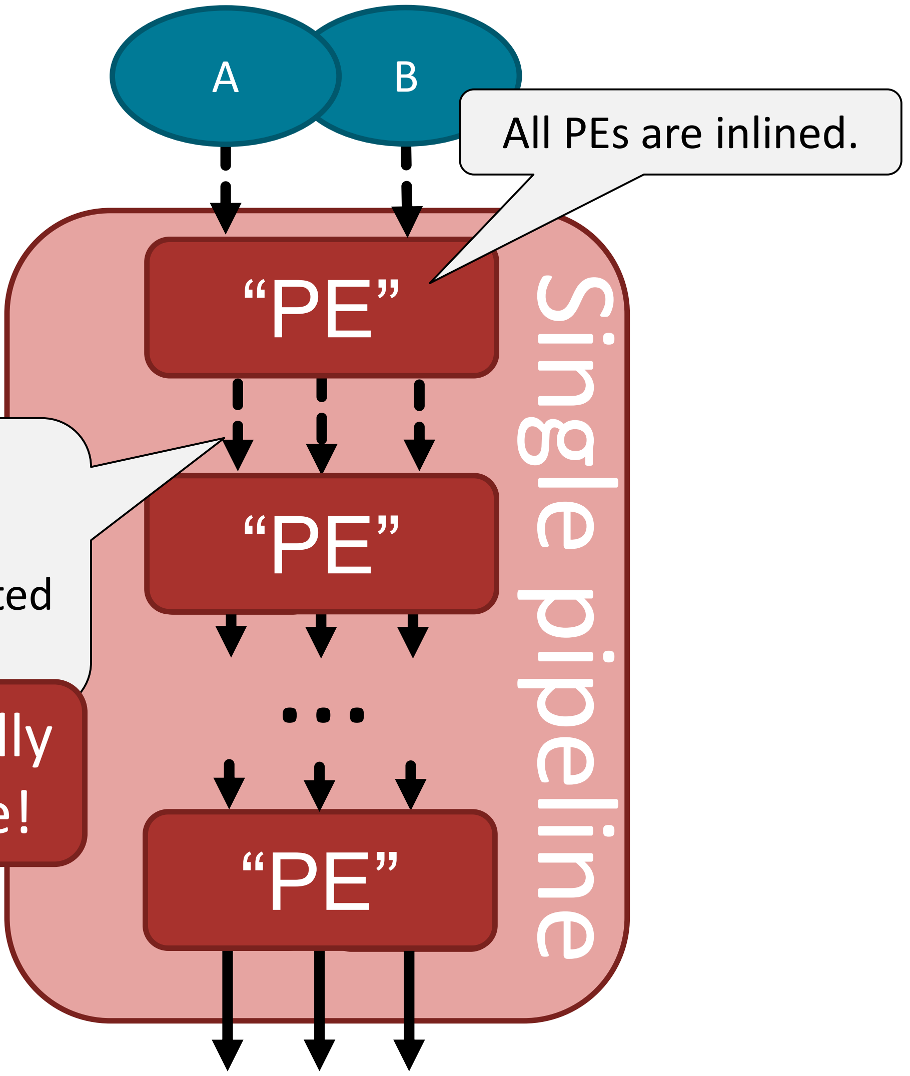
For a detailed description of HLS transformations, see:

Transformations of High-Level Synthesis Codes for High-Performance Computing
https://arxiv.org/abs/1805.08288

Two flavors of systolic arrays



FIFOs offset initialization and draining phases: *easier to program*.



All PEs are inlined.

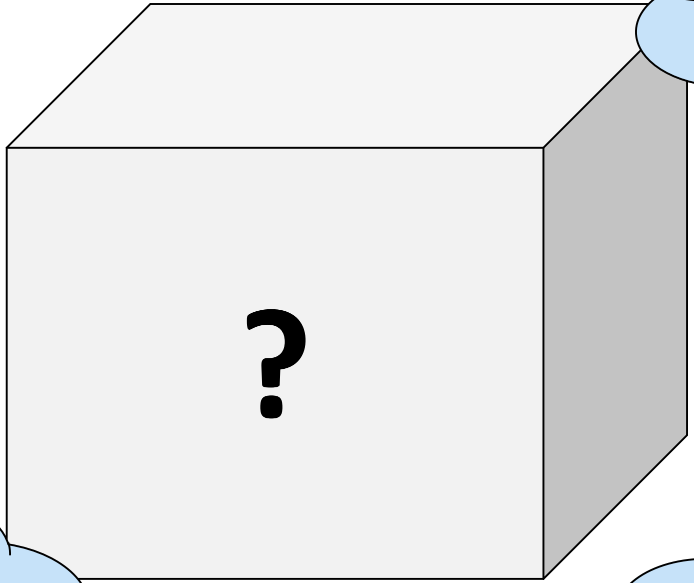
Use a synchronous **distribution network**. Requires careful predicated orchestration of loops.

Only attempt this if you fully understand your schedule!

Approaching a new problem

Scalable source of temporal locality?

Can we tile the problem to fit on the FPGA?



Where are the memory accesses?

Can we partition into PEs?

When HDL should be involved

```
always @(posedge ACLK) begin
  if (~ARESETN | system_reset) begin
    write_state <= WRITE_IDLE;
    write_addr <= 0;
    start_kernel_signal <= 1'b0;
    SC_reset <= 1'b0;
    icap_wr <= 1'b0;
    host2device_wr <= 1'b0;
    host2device <= 32'h0;
    system_reset <= 1'b0;
  end
  else begin
   AWREADY <= 1'b1;
   WREADY <= 1'b0;
   BVALID <= 1'b0;
   system_reset <= 1'b0;
```

```
for (int i = 1; i < N - 1; ++i) {
  for (int j = 0; j < M; ++j) {
    #pragma HLS PIPELINE II=1

                                    ();
                                    ();
                                    M + j];

                                  0.3333;

    memory_out[i * M + j] = average;
  }
}
```

Latency critical optimizations

HDL and HLS can (and do often) happily co-exist!

Interfacing

Claim

FPGAs are more energy efficient than GPUs

What does this actually mean?

How do we measure it?

Compute throughput

$$\frac{F}{P} \quad \left[\frac{\mathrm{Op/s}}{\mathrm{J/s}} = \boxed{\frac{\mathrm{Op}}{\mathrm{J}}}\right]$$

Power draw

Count these...

...and integrate the instantaneous power.

Claim

FPGAs are more energy efficient than GPUs

Compute throughput

At $F_{FPGA} = F_{GPU}$, we are looking at a ... improvement in energy efficiency.

Verdict

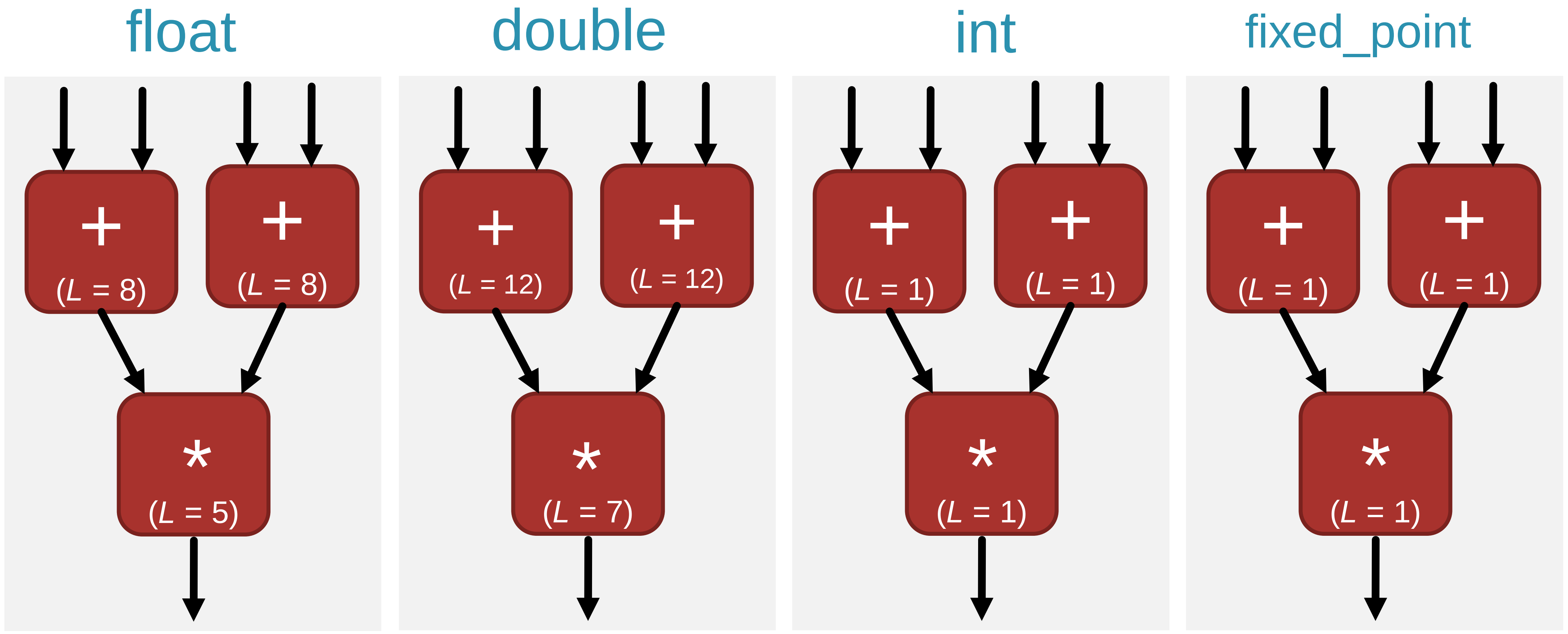
Only if the performance is competitive

For a 10× slowdown, we **lose** a factor 2× in energy efficiency

Might seem trivial... F matters!

(for now, accept these numbers for the sake of argument)

Types on FPGA



Same concepts, different latencies (some problems go away at $L = 1$).