SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis

Julian Oppermann[1], Lukas Sommer[1], Lukas Weber[1], Melanie Reuter-Oppermann[2], Andreas Koch[1], Oliver Sinnen[3]

1: TECHNISCHE UNIVERSITÄT DARMSTADT

2: KIT Karlsruhe Institute of Technology

3: THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND

A Common Problem

A Common Problem

- Given: a kernel, an HLS tool, and an FPGA

A Common Problem

- Given: a kernel, an HLS tool, and an FPGA
  - *What's the fastest mircoarchitecture that still fits on the device?*

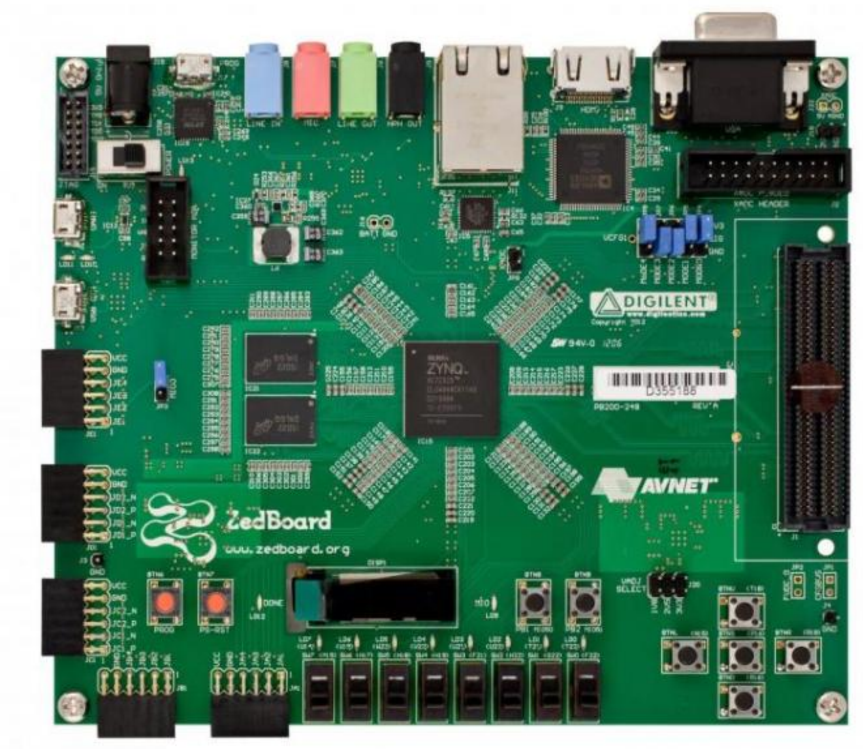
A Common Problem

- Given: a kernel, an HLS tool, and an FPGA
  - *What's the fastest mircoarchitecture that still fits on the device?*

```
{ /∗ 10 FP mul, 1 FP add ∗/ }
{ /∗  8 FP mul, 1 FP add ∗/ }
```

```
double spn(...)           { /∗ 10 FP mul, 1 FP add ∗/ }
double spn_marginal(...)  { /∗  8 FP mul, 1 FP add ∗/ }

double top(char i1, char i2, char i3, char i4) {
  // most probable explanation for "i5"
  char maxClause = -1;  double maxProb = -1.0;
  MPE: for (char x = 0; x < 0xFF; x += 4) {
         double p0 = spn(i1, i2, i3, i4, x);
         double p1 = spn(i1, i2, i3, i4, x+1);
         double p2 = spn(i1, i2, i3, i4, x+2);
         double p3 = spn(i1, i2, i3, i4, x+3);
         maxProb   = ... // max(maxProb, p0, p1, p2, p3);
         maxClause = ... // argument value for i5 that
                         // yielded new value for maxProb
  }
  double pM = spn_marginal(i2, i3, i4, maxClause);
  return maxProb / pM;
}
```



A Common Problem

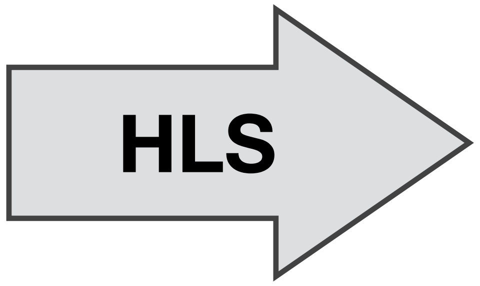
- Given: a kernel, an HLS tool, and an FPGA
  - *What's the fastest mircoarchitecture that still fits on the device?*

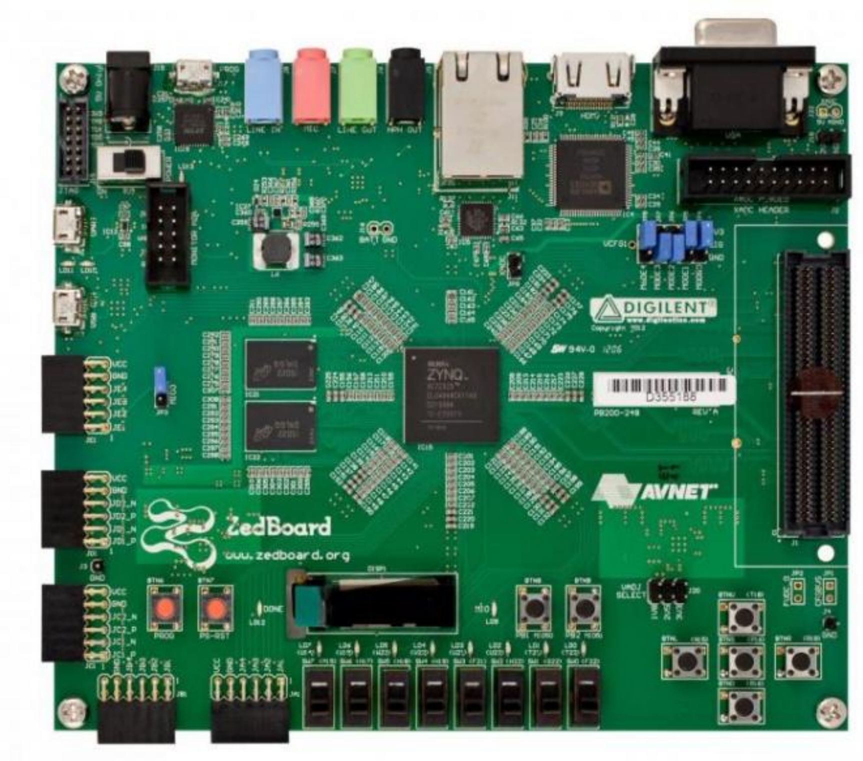
```
{ /∗ 10 FP mul, 1 FP add ∗/ }
{ /∗  8 FP mul, 1 FP add ∗/ }
```

```
double spn(...)          { /∗ 10 FP mul, 1 FP add ∗/ }
double spn_marginal(...) { /∗  8 FP mul, 1 FP add ∗/ }

double top(char i1, char i2, char i3, char i4) {
  // most probable explanation for "i5"
  char maxClause = -1;  double maxProb = -1.0;
  MPE: for (char x = 0; x < 0xFF; x += 4) {
         double p0 = spn(i1, i2, i3, i4, x);
         double p1 = spn(i1, i2, i3, i4, x+1);
         double p2 = spn(i1, i2, i3, i4, x+2);
         double p3 = spn(i1, i2, i3, i4, x+3);
         maxProb   = ... // max(maxProb, p0, p1, p2, p3);
         maxClause = ... // argument value for i5 that
                         // yielded new value for maxProb
  }
  double pM = spn_marginal(i2, i3, i4, maxClause);
  return maxProb / pM;
}
```

ZedBoard with XC7Z020:
220 DSP blocks





src: Xilinx

A Common Problem

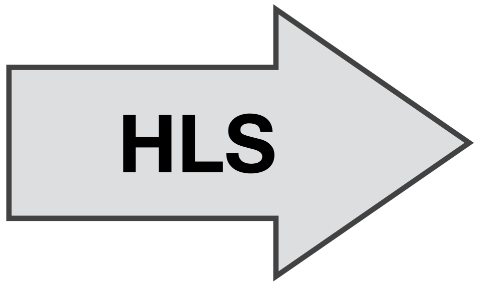
- Given: a kernel, an HLS tool, and an FPGA
  - *What's the fastest mircoarchitecture that still fits on the device?*

```
{ /* 10 FP mul, 1 FP add */ }
{ /*  8 FP mul, 1 FP add */ }
```

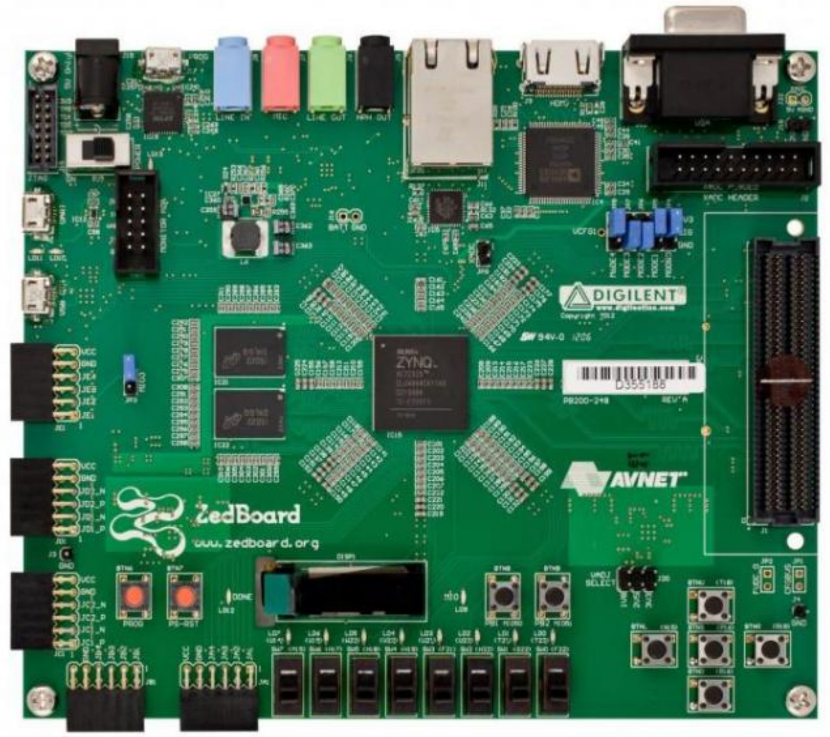
```
double spn(...)          { /* 10 FP mul, 1 FP add */ }
double spn_marginal(...) { /*  8 FP mul, 1 FP add */ }

double top(char i1, char i2, char i3, char i4) {
  // most probable explanation for "i5"
  char maxClause = -1;  double maxProb = -1.0;
  MPE: for (char x = 0; x < 0xFF; x += 4) {
         double p0 = spn(i1, i2, i3, i4, x);
         double p1 = spn(i1, i2, i3, i4, x+1);
         double p2 = spn(i1, i2, i3, i4, x+2);
         double p3 = spn(i1, i2, i3, i4, x+3);
         maxProb   = ... // max(maxProb, p0, p1, p2, p3);
         maxClause = ... // argument value for i5 that
                         // yielded new value for maxProb
  }
  double pM = spn_marginal(i2, i3, i4, maxClause);
  return maxProb / pM;
}
```

ZedBoard with XC7Z020:
220 DSP blocks



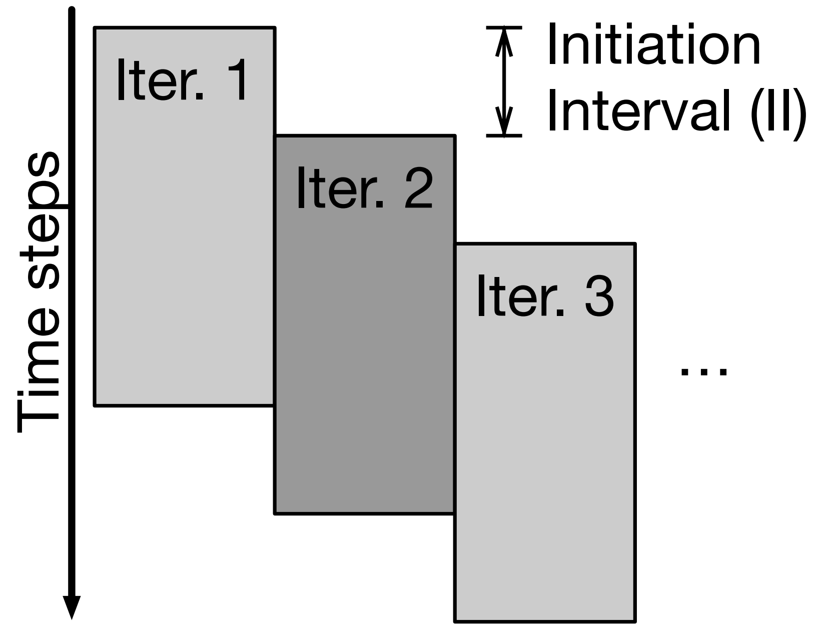
Fastest µ-arch (II=1)
499 DSP blocks

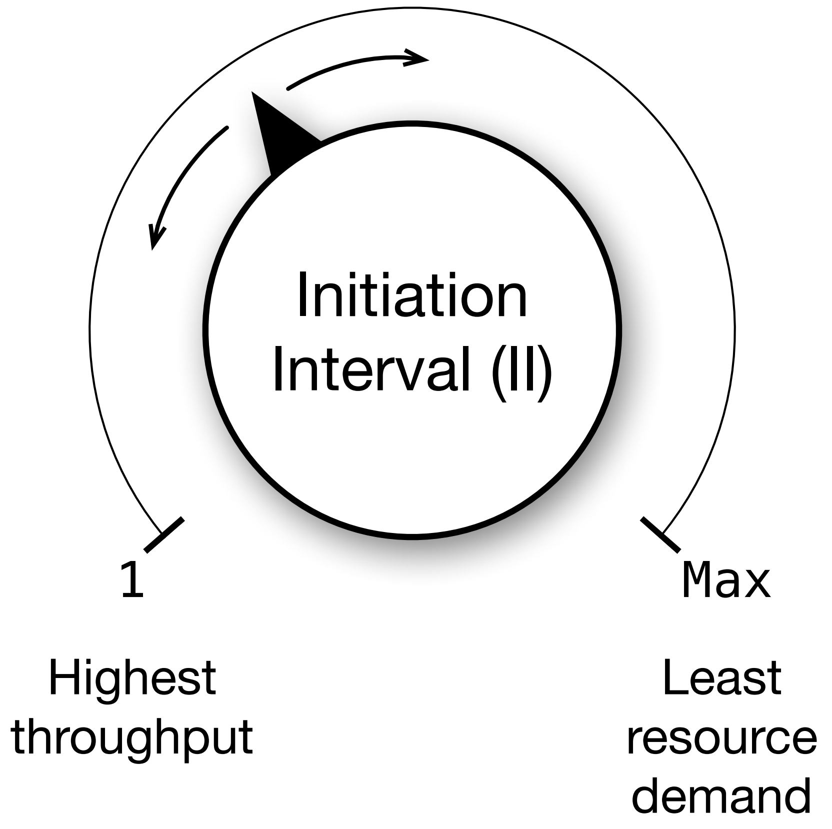


src: Xilinx

Fitting a Kernel

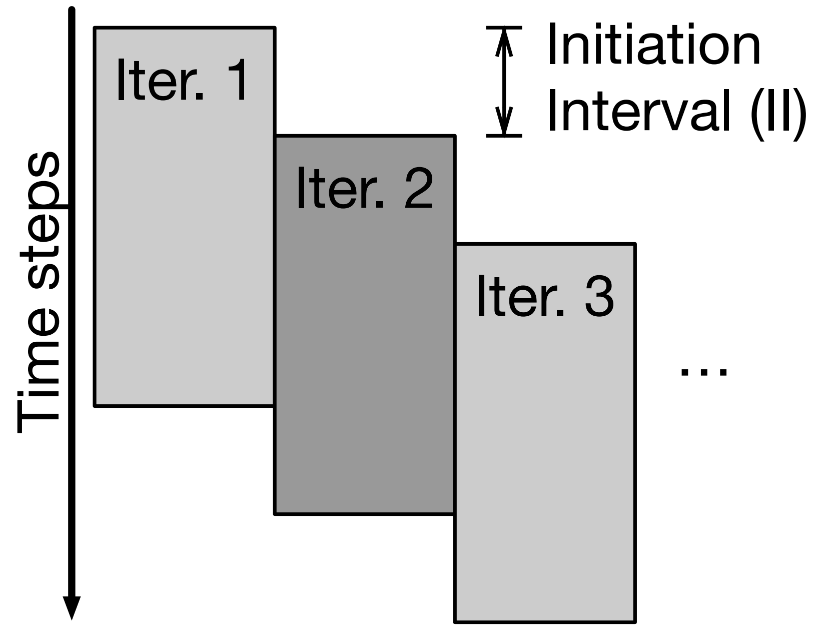
- Most influential control „knob“: amount of **(loop) pipelining**

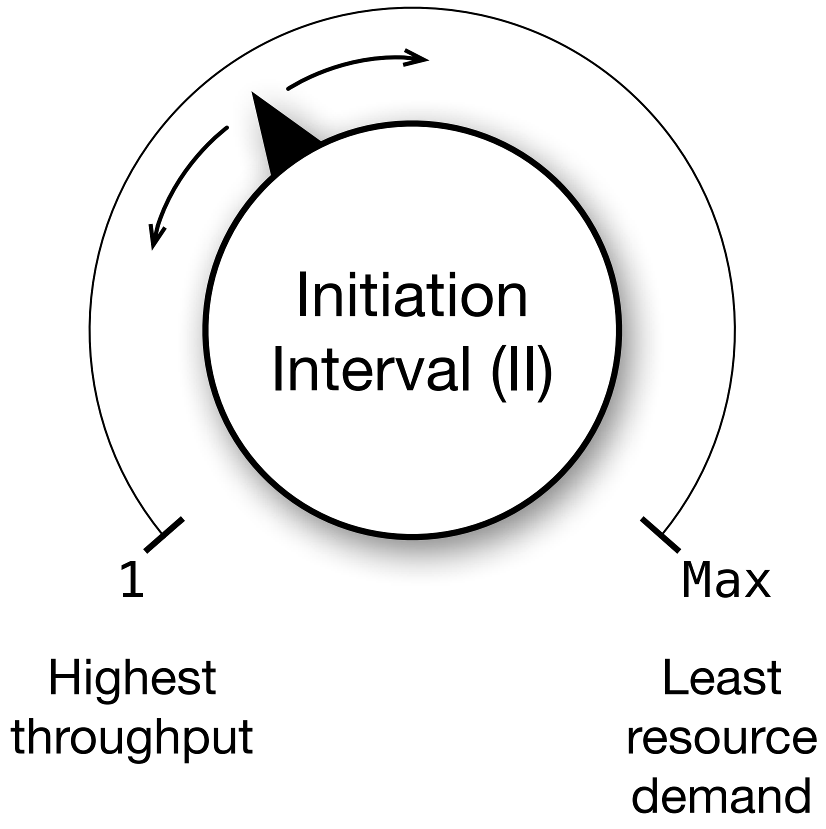




Fitting a Kernel

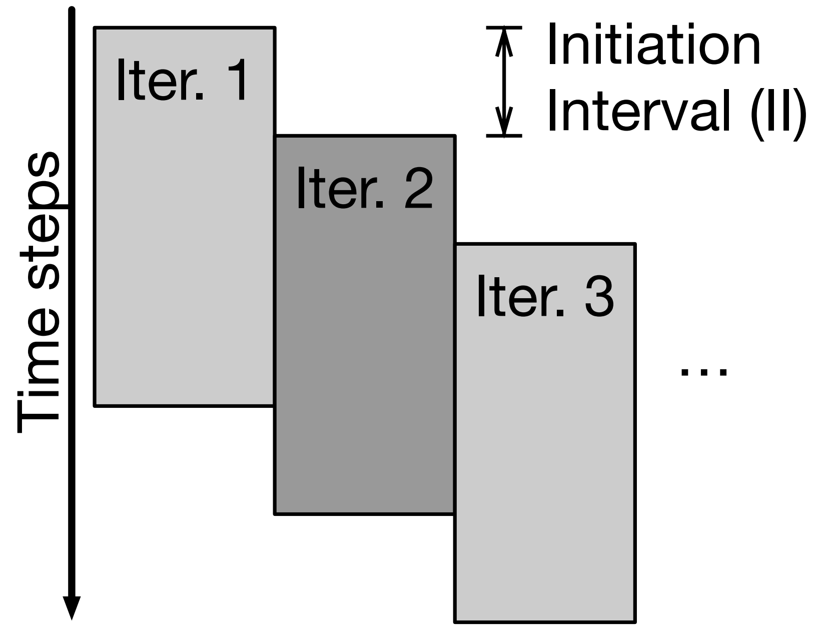
- Most influential control „knob“: amount of **(loop) pipelining**
- Tweak manually?

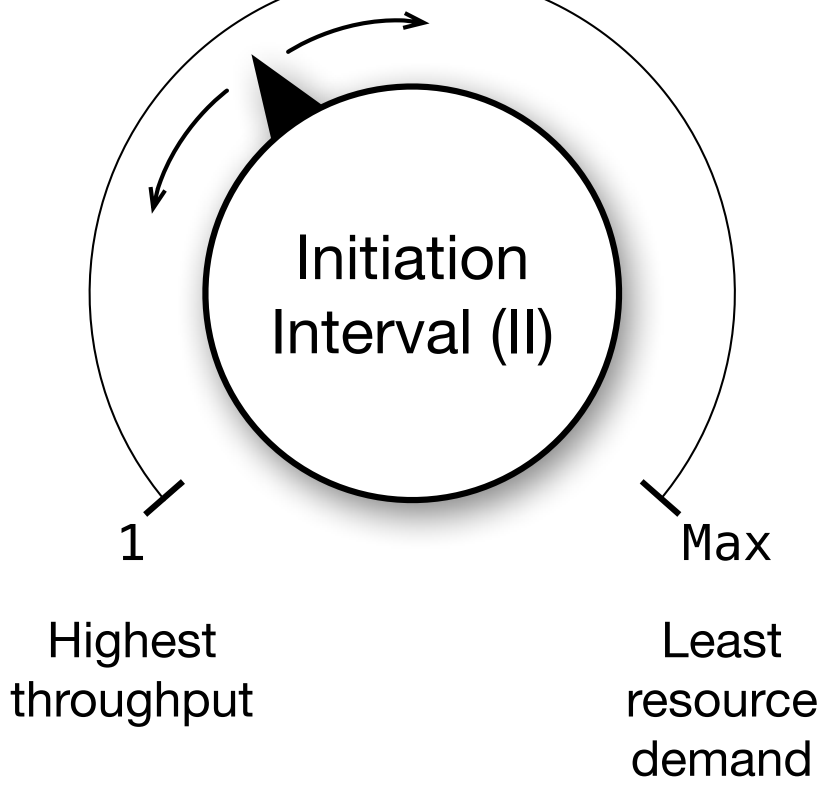




Fitting a Kernel

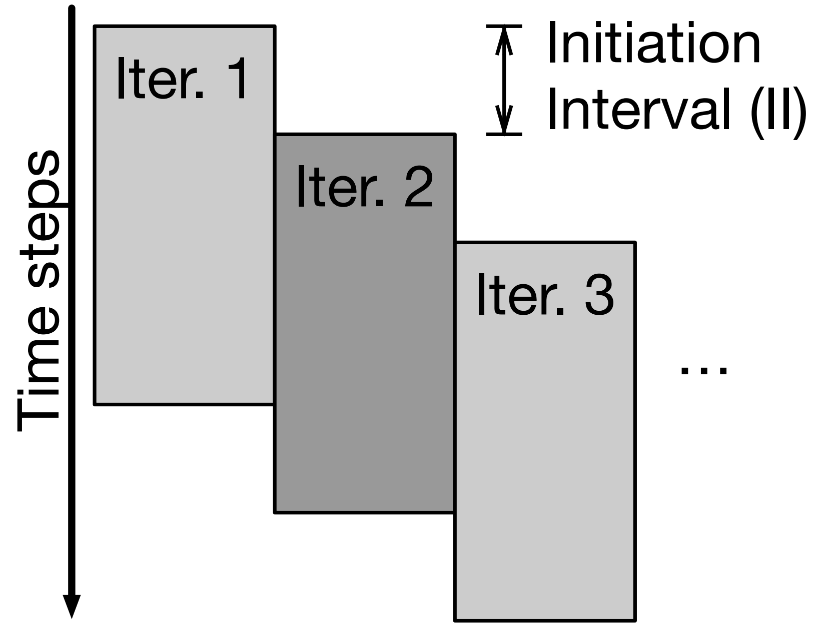
- Most influential control „knob“: amount of **(loop) pipelining**
- Tweak manually?
- Use external exploration tool?

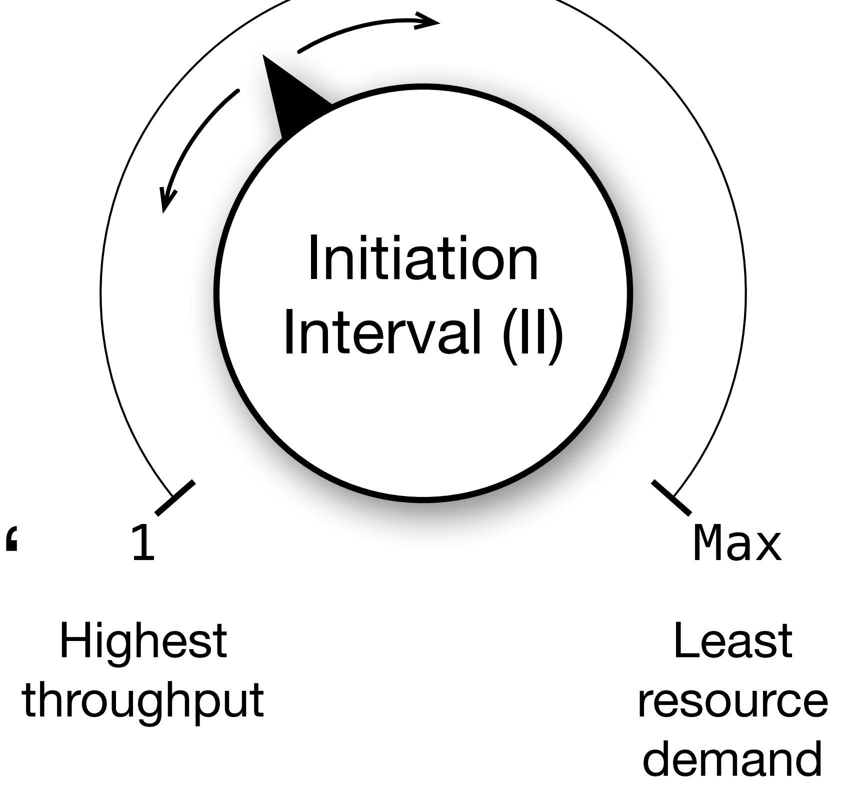




Fitting a Kernel

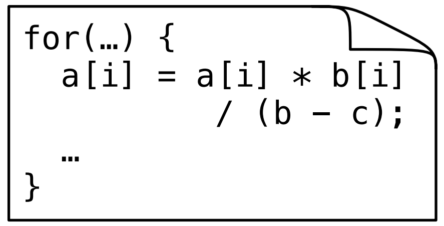
- Most influential control „knob“: amount of **(loop) pipelining**
- Tweak manually?
- Use external exploration tool?
- Integrate into core HLS algorithms!
  - Optimisation problem:
    maximise „performance“
    subject to „resource constraints“





High-Level Synthesis

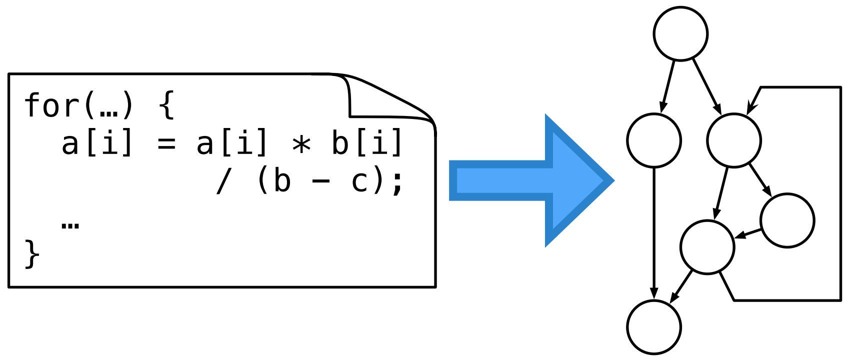
High-Level Synthesis



```
for(…) {
  a[i] = a[i] * b[i]
          / (b - c);
  …
}
```

- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

High-Level Synthesis

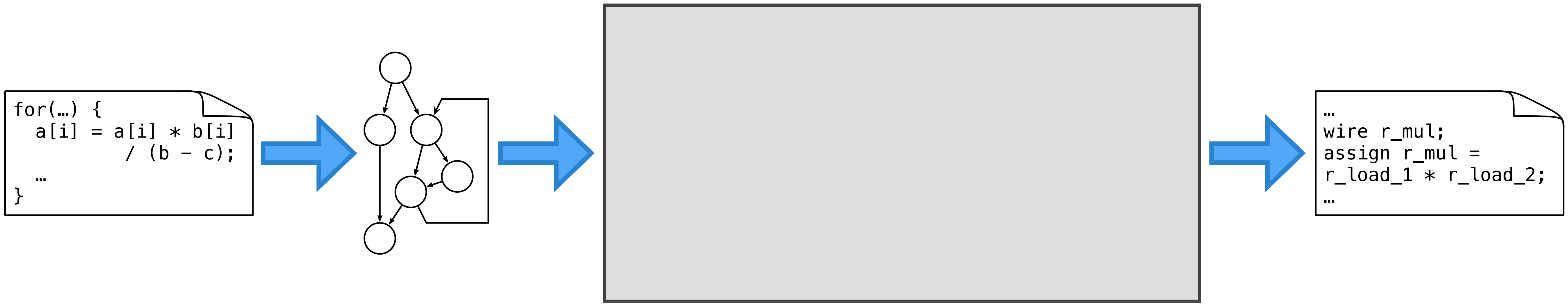


```
for(…) {
  a[i] = a[i] * b[i]
          / (b - c);
  …
}
```

- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

High-Level Synthesis



```
for(…) {
  a[i] = a[i] * b[i]
          / (b - c);
  …
}
```

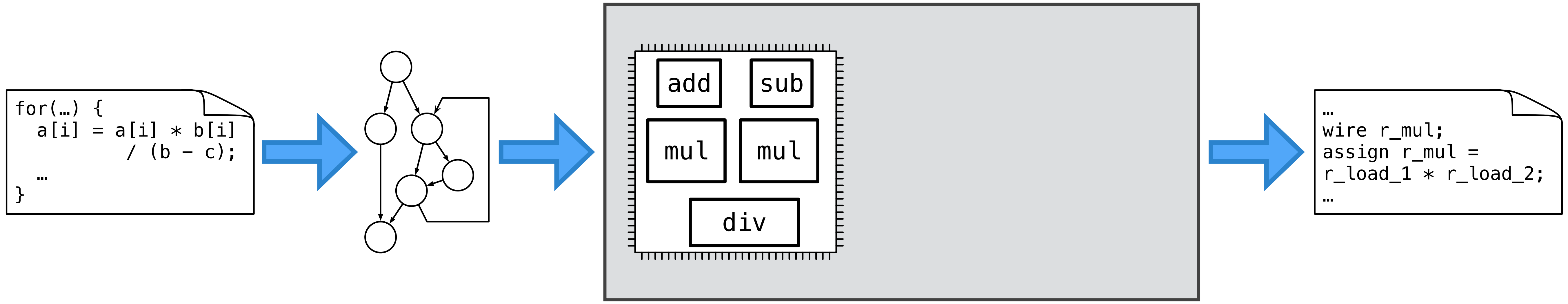
```
…
wire r_mul;
assign r_mul =
r_load_1 * r_load_2;
…
```

- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

- Algorithmic steps

High-Level Synthesis

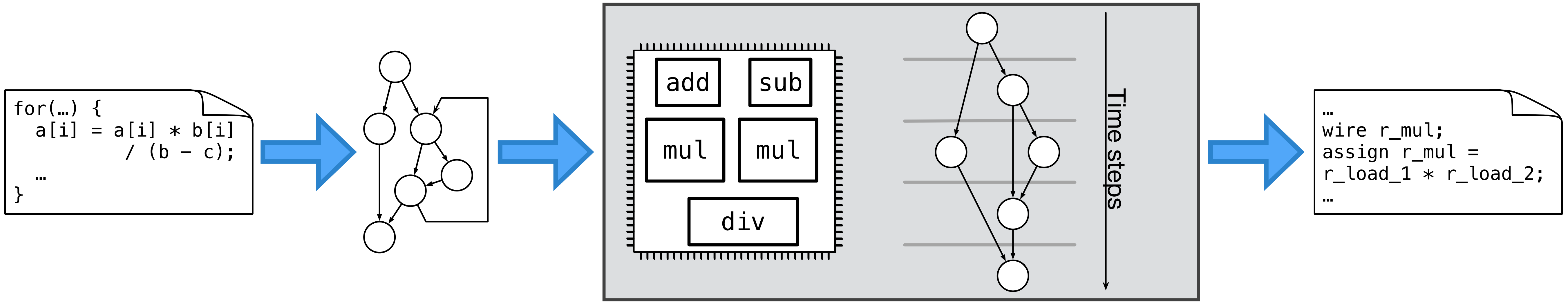


- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

- Algorithmic steps
  - Allocation — *how many operators?*

High-Level Synthesis

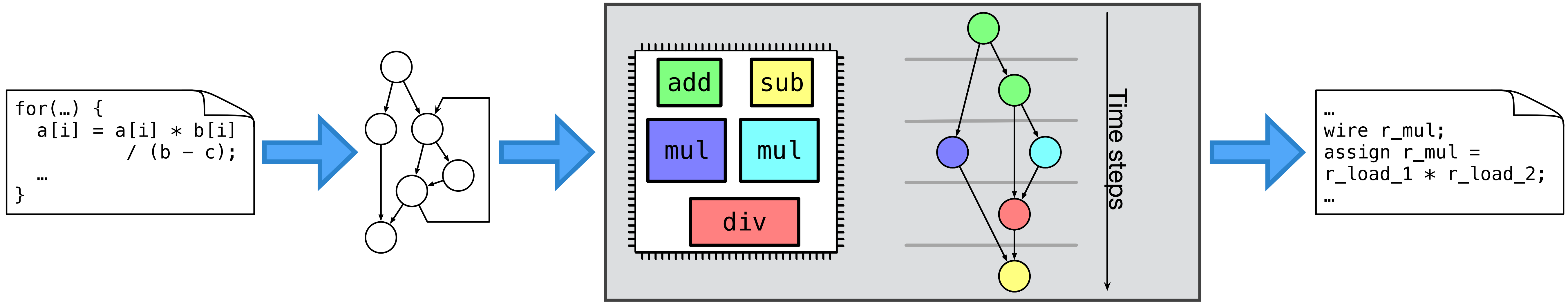


- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

- Algorithmic steps
  - Allocation — *how many operators?*
  - Scheduling — *when is an operation executed?*

High-Level Synthesis



```
for(…) {
  a[i] = a[i] * b[i]
          / (b - c);
  …
}
```

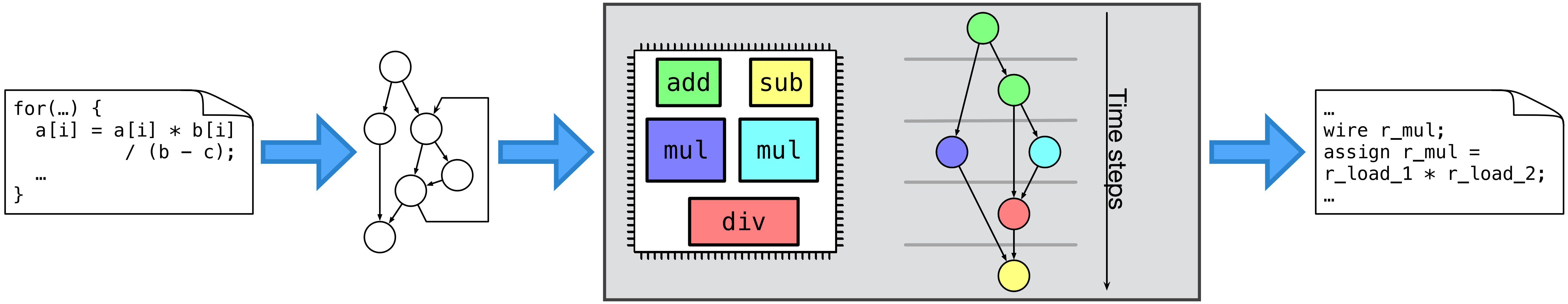
```
…
wire r_mul;
assign r_mul =
r_load_1 * r_load_2;
…
```

- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

- Algorithmic steps
  - Allocation — *how many operators?*
  - Scheduling — *when is an operation executed?*
  - Binding — *where is an operation executed?*

High-Level Synthesis



```
for(…) {
  a[i] = a[i] * b[i]
          / (b - c);
  …
}
```

```
…
wire r_mul;
assign r_mul =
r_load_1 * r_load_2;
…
```

- HLS = Automatic microarchitecture construction from a behavioural description
  think: C code

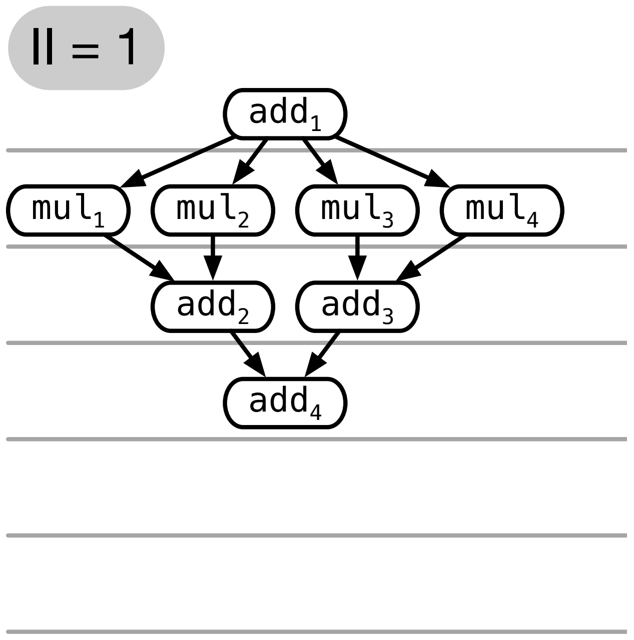
- Terminology
  - Loops (and other regions) are transformed to control-data-flow-graphs comprised of **operations** and **dependence edges**
  - Operations require **operators** to perform intended function (e.g. floating-point addition)
  - Operators occupy **resources** on the FPGA device (e.g. DSP blocks)

- Algorithmic steps
  - Allocation — *how many operators?*
  - Scheduling — *when is an operation executed?*
  - Binding — *where is an operation executed?*

Modulo Scheduling
enables pipelining

Trade-offs

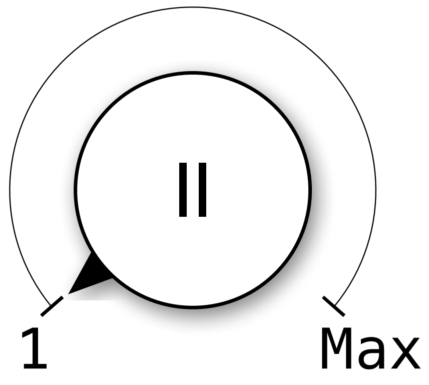
- For one loop, trade-offs can be computed with a **resource-aware** modulo scheduler [Euro-Par'19]



Allocation:

ADD	ADD	ADD	ADD
MUL	MUL	MUL	MUL

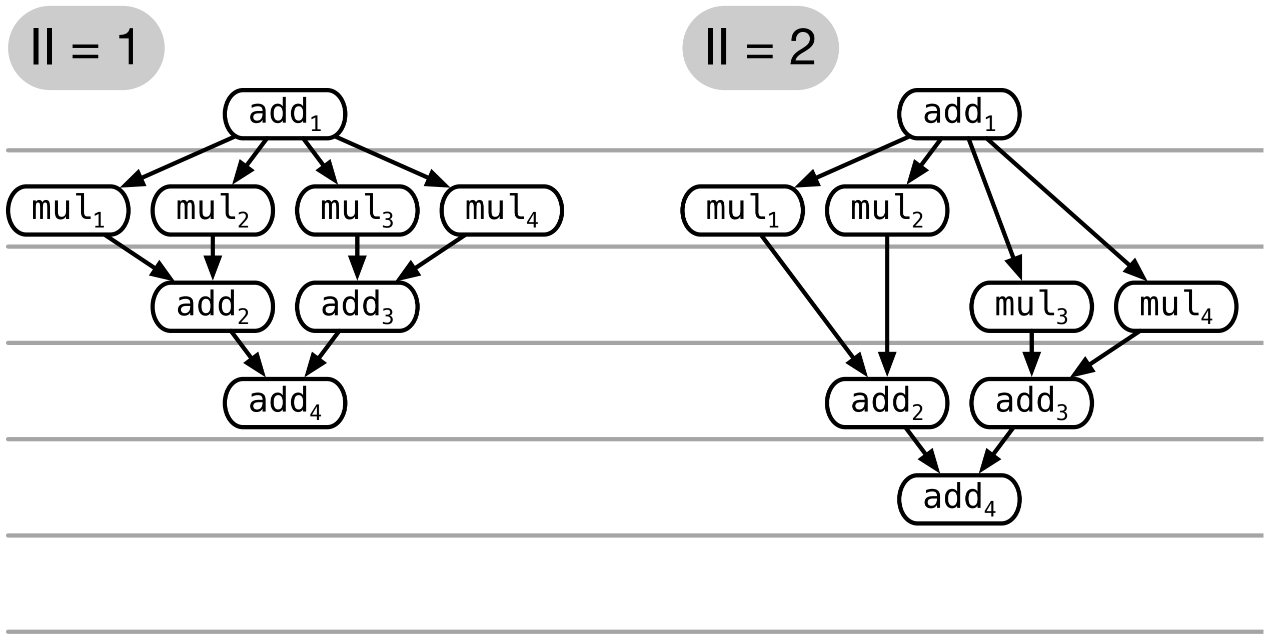
Highest throughput

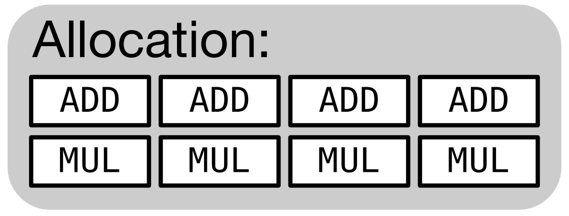


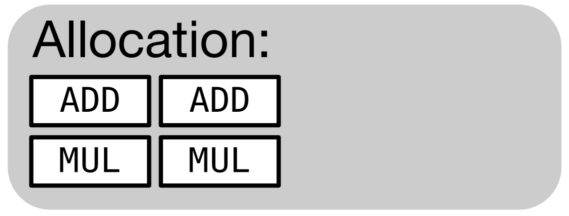
Least resource demand

Trade-offs

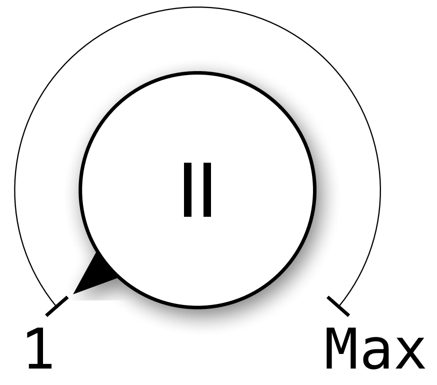
- For one loop, trade-offs can be computed with a **resource-aware** modulo scheduler [Euro-Par'19]

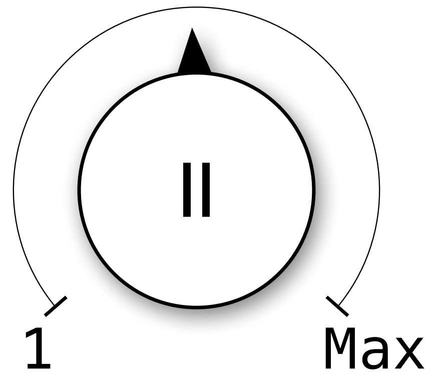






Highest throughput

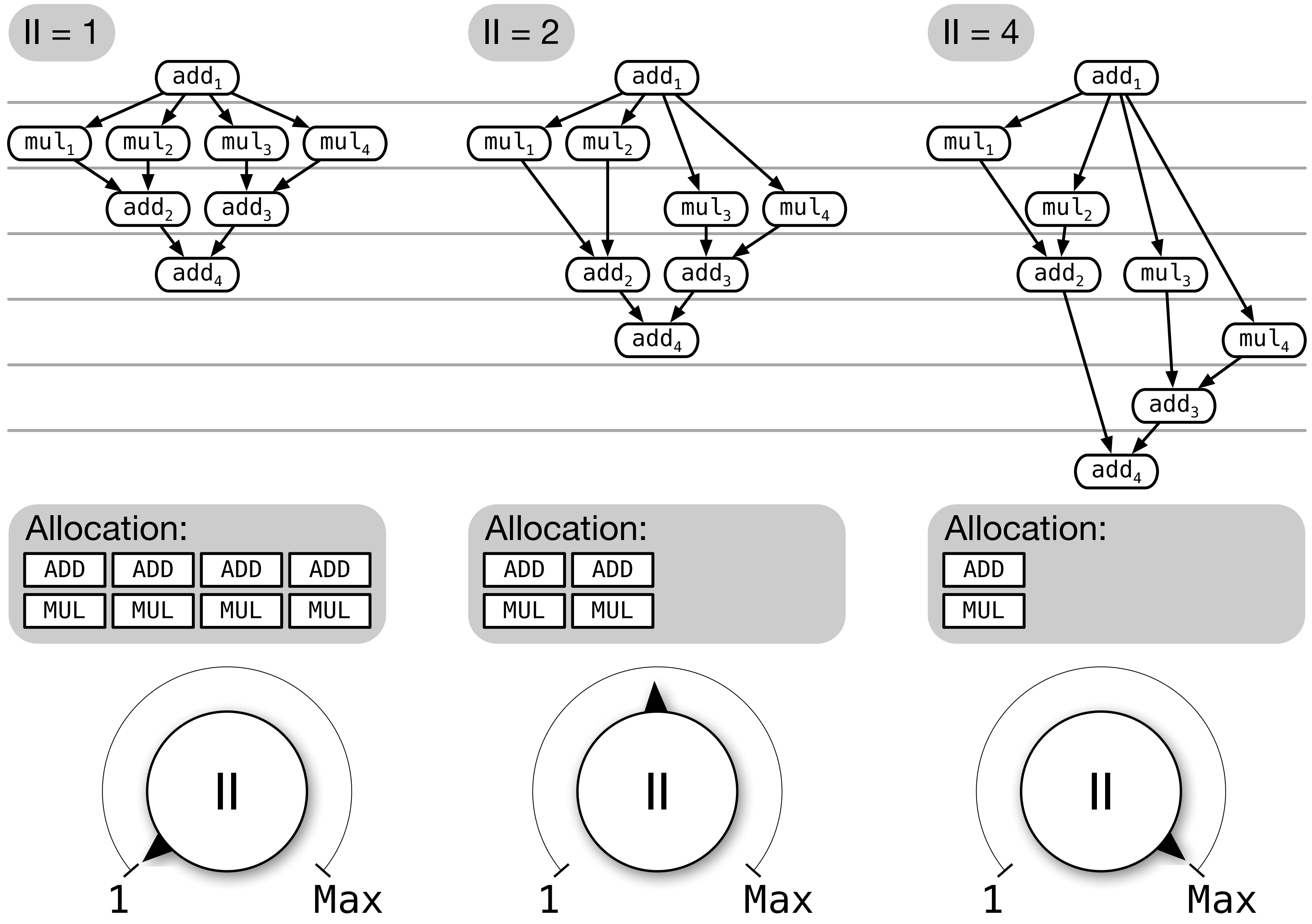




Least resource demand

Trade-offs

- For one loop, trade-offs can be computed with a **resource-aware** modulo scheduler [Euro-Par'19]



In Reality…

- Typical HLS kernels have:
  - More than one loop
  - Non-pipelined parts

In Reality...

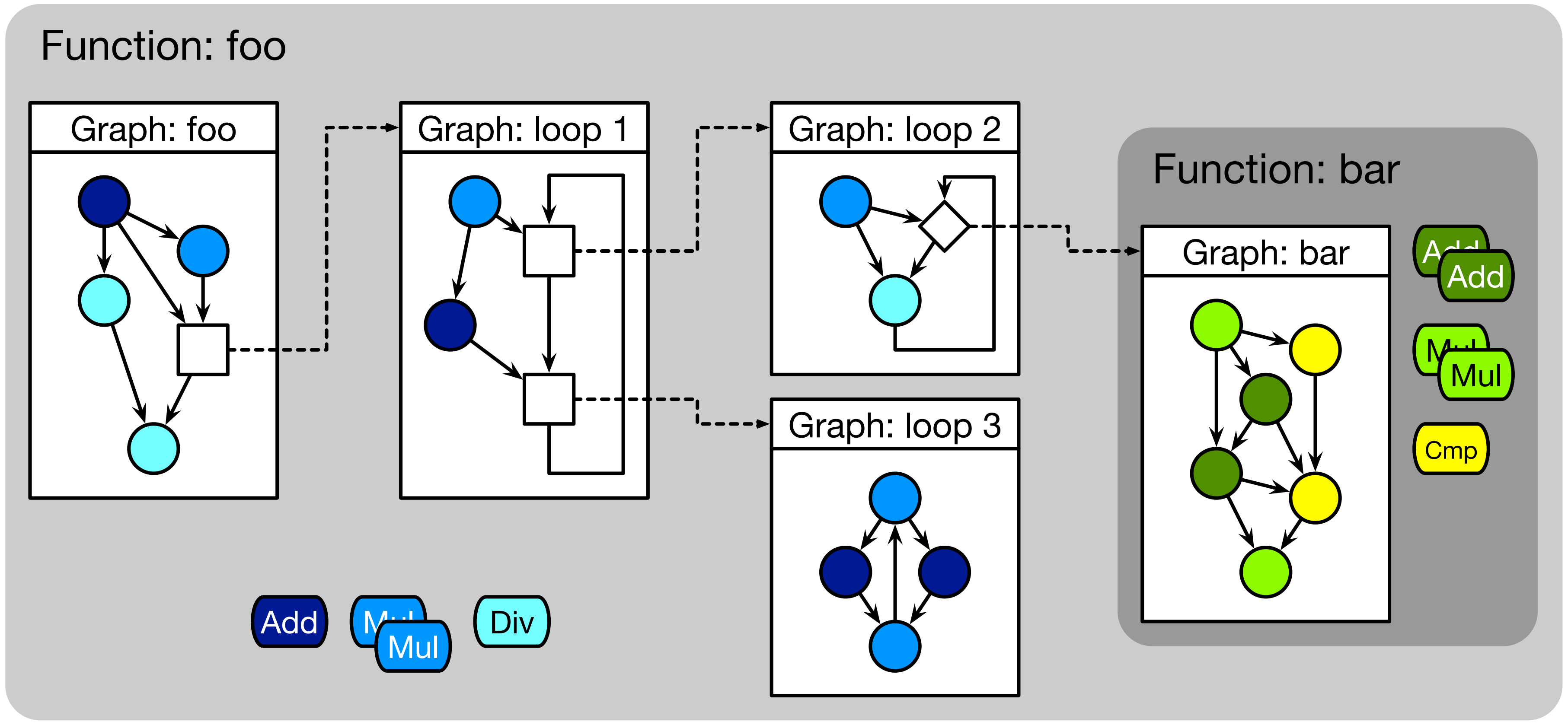
- Typical HLS kernels have:
  - More than one loop
  - Non-pipelined parts
- Typical HLS tools share operators between loops

In Reality...

- Typical HLS kernels have:
  - More than one loop
  - Non-pipelined parts
- Typical HLS tools share operators between loops

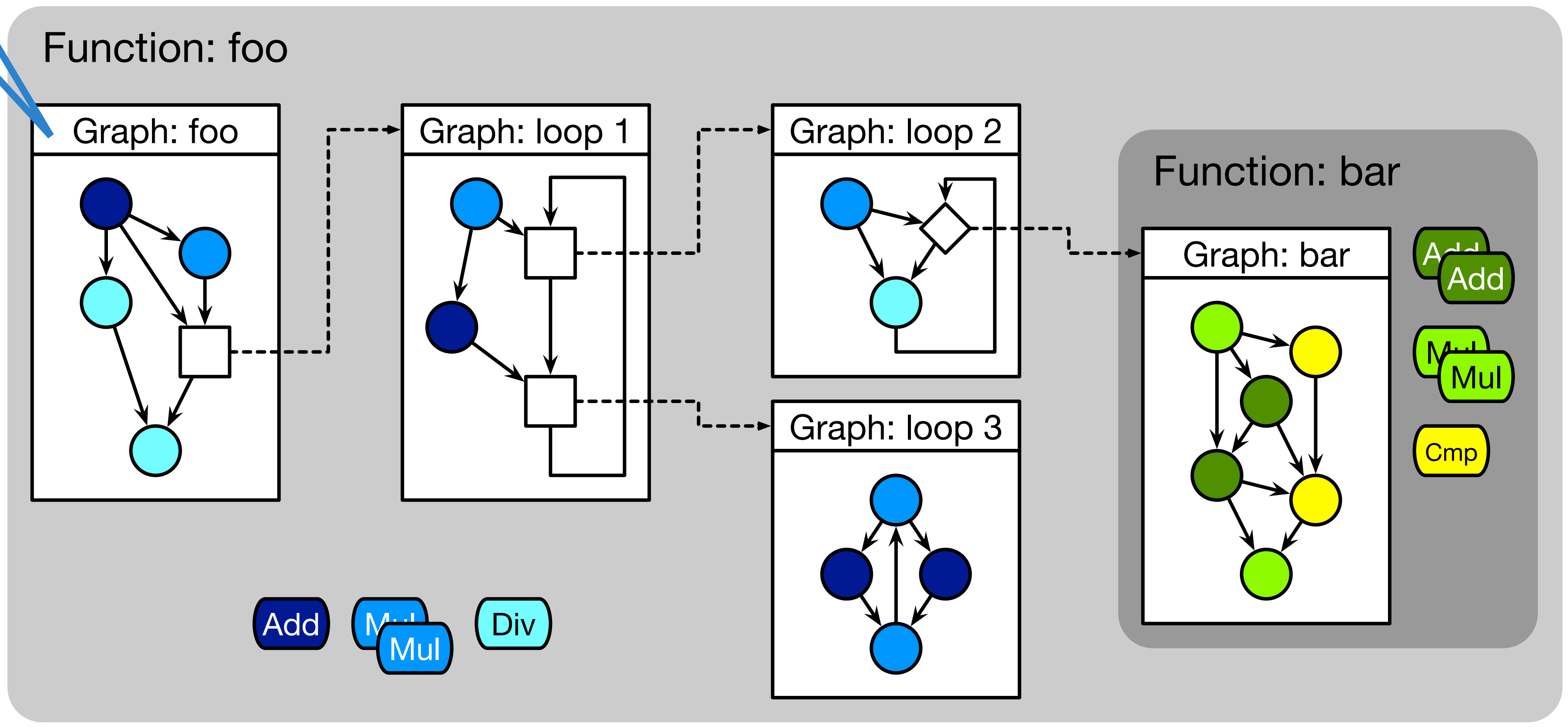
▶ Need a formal model for <u>that</u>!

Multi-Loop Scheduling Problem

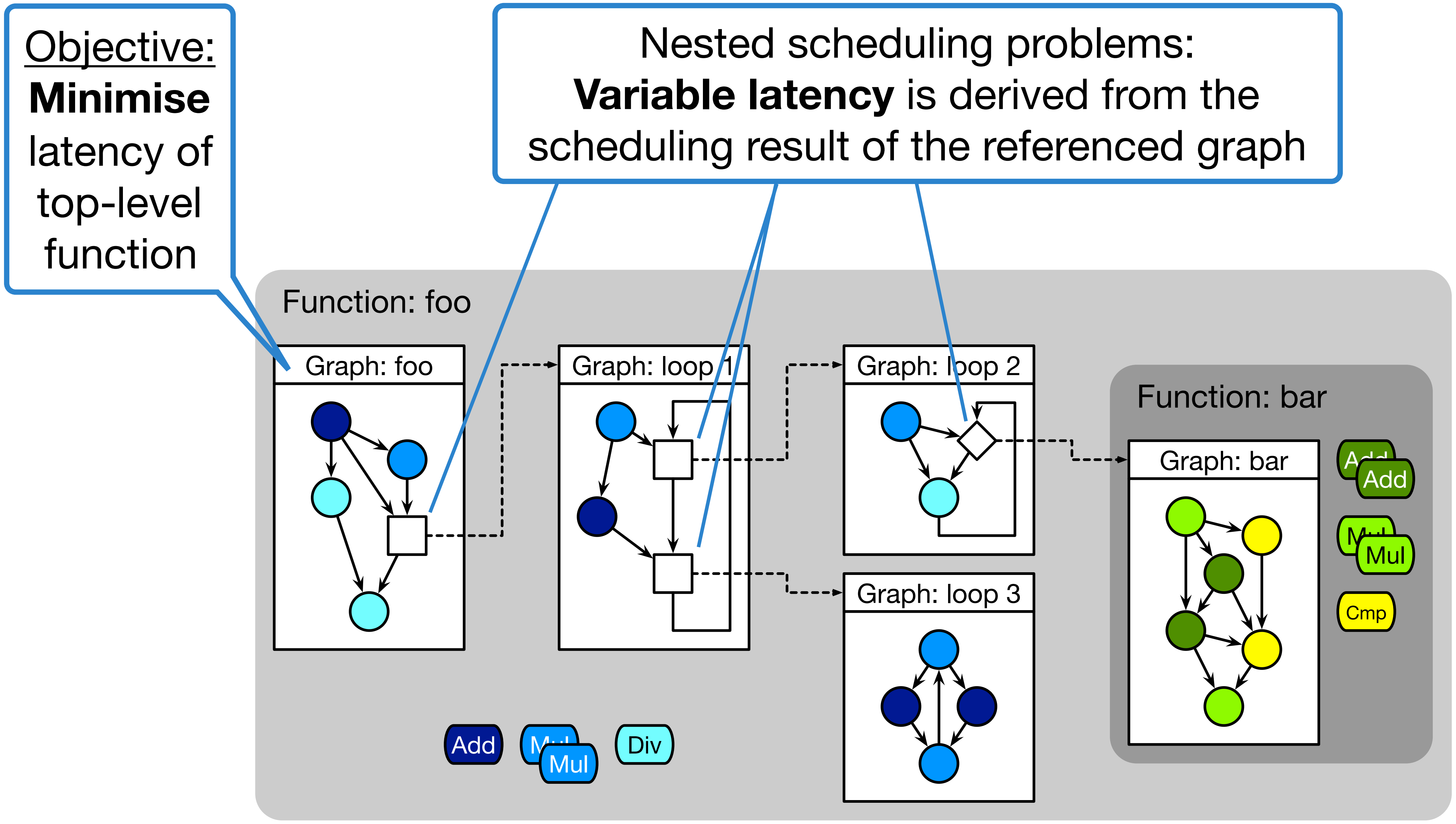


Multi-Loop Scheduling Problem

Objective:
Minimise
latency of
top-level
function



Multi-Loop Scheduling Problem

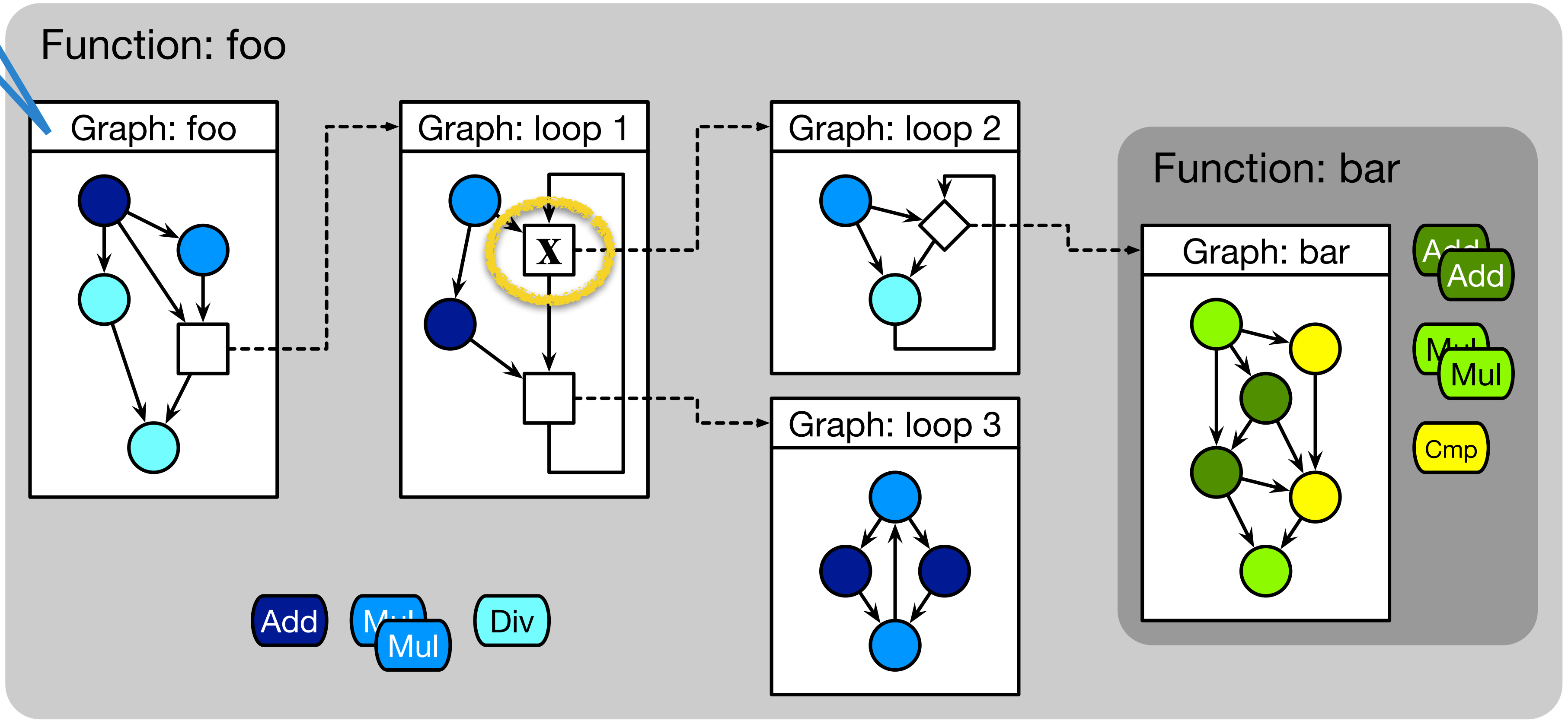

Objective:
Minimise latency of top-level function
Nested scheduling problems:
Variable latency is derived from the scheduling result of the referenced graph
Function: foo
Graph: foo
Graph: loop 1
Graph: loop 2
Graph: loop 3
Function: bar
Graph: bar
Add
Add
Mul
Mul
Cmp
Add
Mul
Mul
Div

Multi-Loop Scheduling Problem

<u>Objective:</u>
Minimise latency of top-level function

Nested scheduling problems:
Variable latency is derived from the scheduling result of the referenced graph

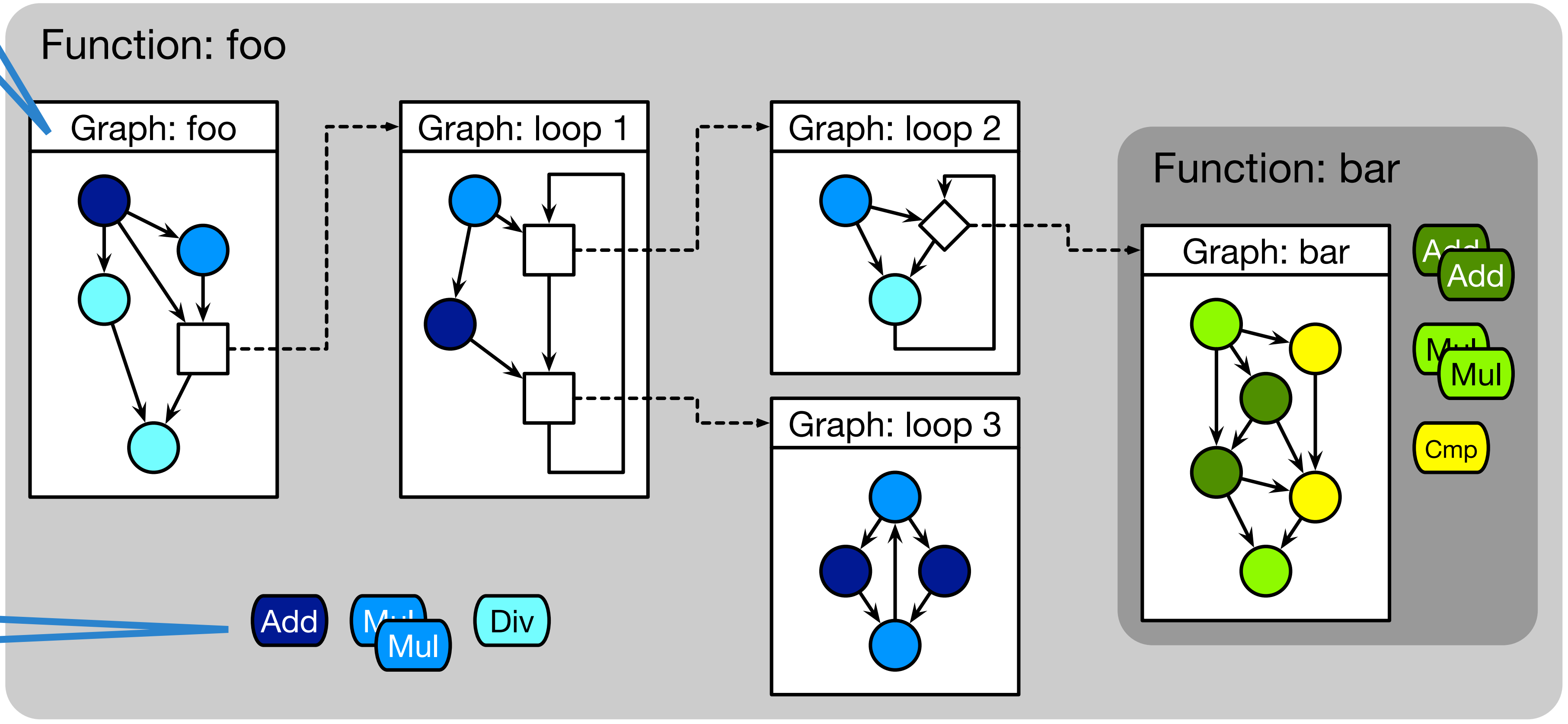
$$latency_x = (trip\_count_{loop2} - 1) \times II_{loop2} + schedule\_length_{loop2}$$



Multi-Loop Scheduling Problem

Objective:
Minimise latency of top-level function

Nested scheduling problems:
Variable latency is derived from the scheduling result of the referenced graph

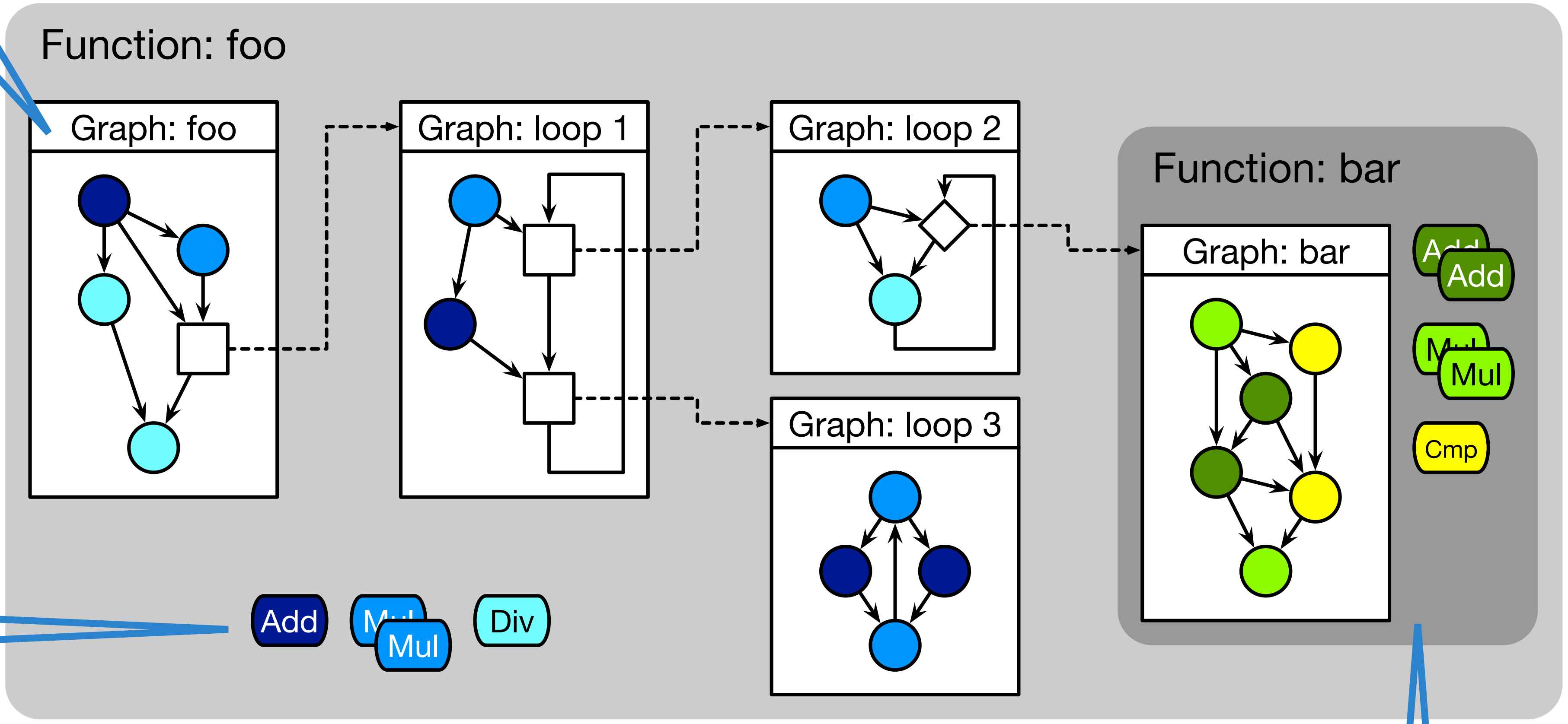


Variable allocation, shared among the scheduling problems in function „foo“

Multi-Loop Scheduling Problem

Objective:
Minimise latency of top-level function

Nested scheduling problems:
Variable latency is derived from the scheduling result of the referenced graph



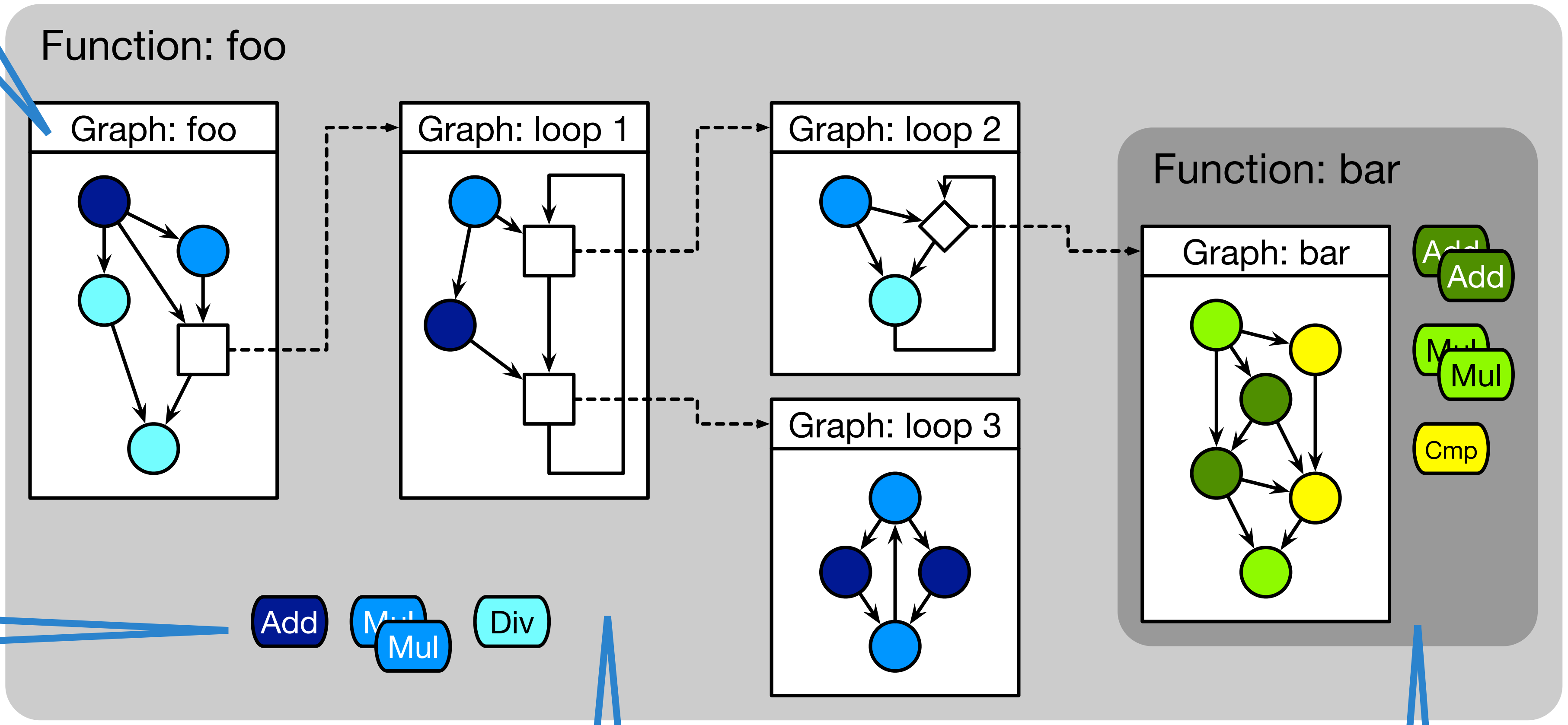
Variable allocation, shared among the scheduling problems in function „foo“

Operator type with **variable latency**, **variable II**, and **variable resource demands**, derived from scheduling „bar“

Multi-Loop Scheduling Problem

Objective: **Minimise** latency of top-level function

Nested scheduling problems: **Variable latency** is derived from the scheduling result of the referenced graph



Variable allocation, shared among the scheduling problems in function „foo“

Constraint: „foo“'s accumulated resource demand ≤ available resources!

Operator type with **variable latency**, **variable II**, and **variable resource demands**, derived from scheduling „bar“

Targeting Vivado HLS

- **Operator sharing** at the function level
  - Implicit in the formal model
  - Other schemes also possible

Targeting Vivado HLS

- **Operator sharing** at the <u>function</u> level
  - Implicit in the formal model
  - Other schemes also possible
- Pipelined regions **cannot** contain **loops**

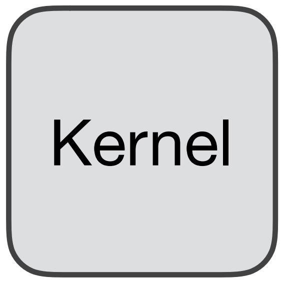
Targeting Vivado HLS

- **Operator sharing** at the function level
  - Implicit in the formal model
  - Other schemes also possible
- Pipelined regions **cannot** contain **loops**
- Pipelined regions **may** contain **calls** to pipelined functions
  - Callee's II must divide II of caller's graph

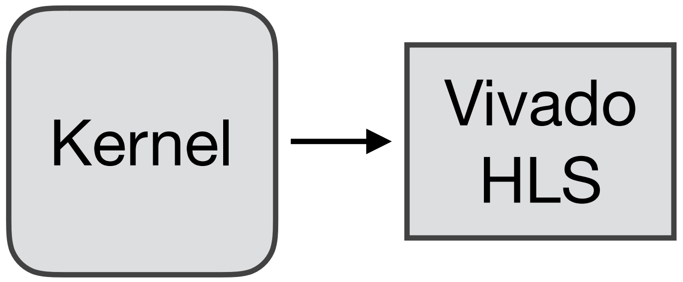
Targeting Vivado HLS

- **Operator sharing** at the function level
  - Implicit in the formal model
  - Other schemes also possible
- Pipelined regions **cannot** contain **loops**
- Pipelined regions **may** contain **calls** to pipelined functions
  - Callee’s II must divide II of caller’s graph
- Closed tool, no interface to influence HLS steps
  - Faithful reproduction of the scheduling and allocation problems inside of Vivado HLS

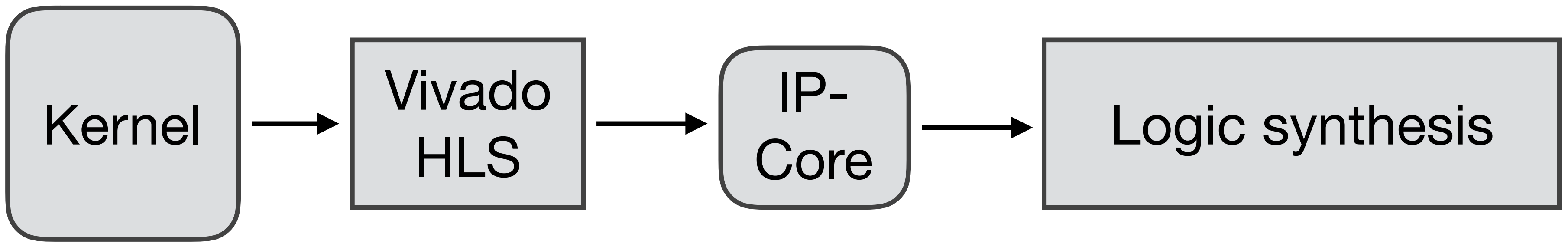
Proposed Flow



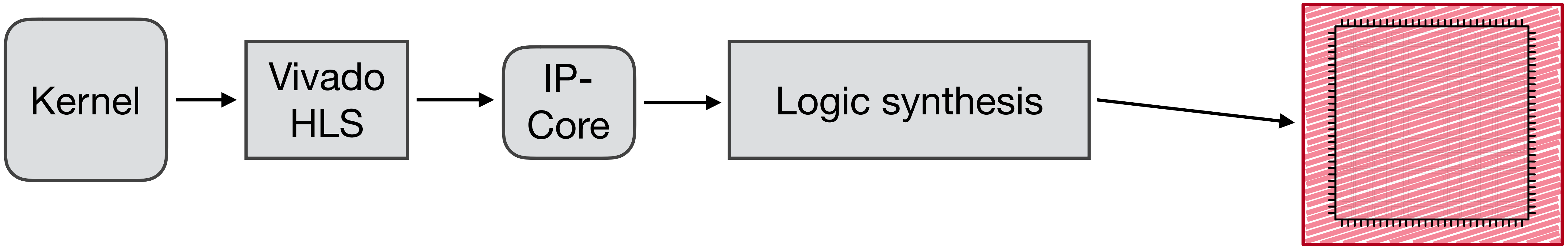
Proposed Flow



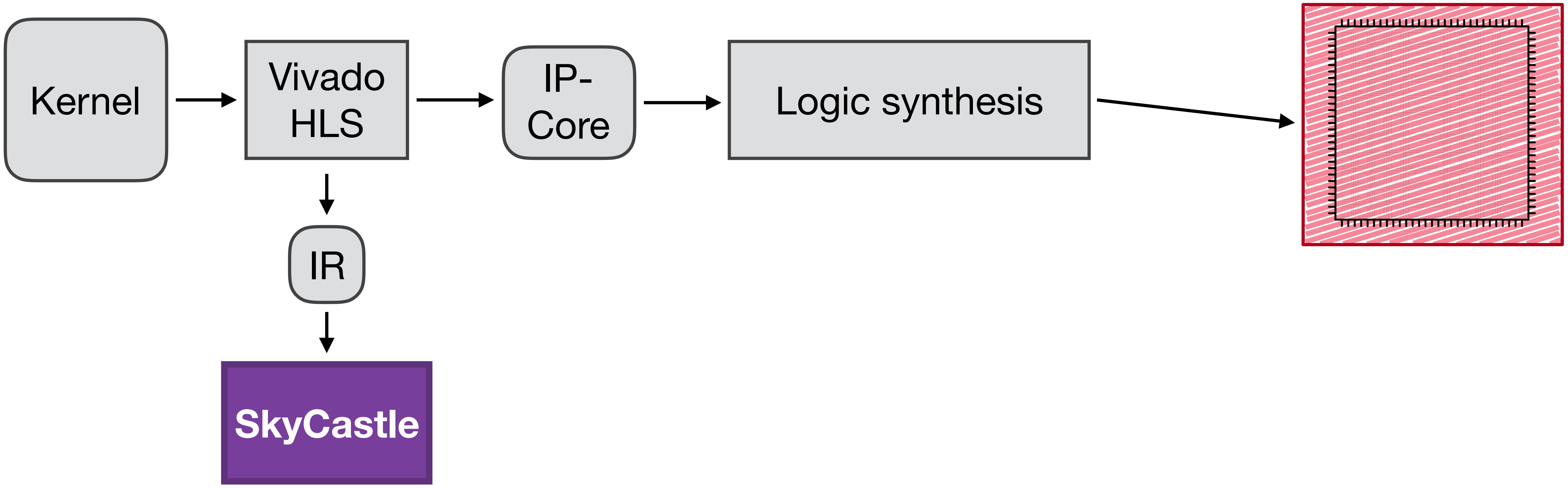
Proposed Flow



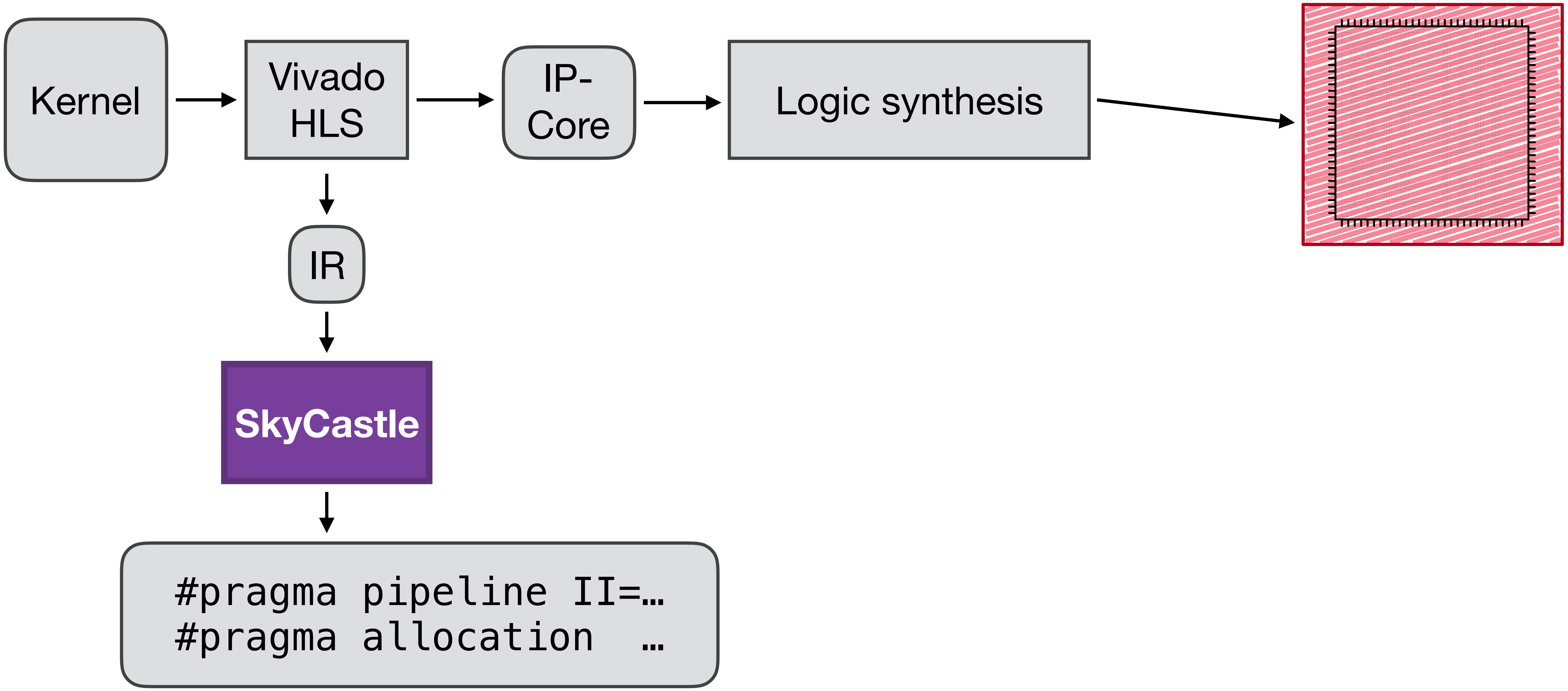
Proposed Flow



Proposed Flow

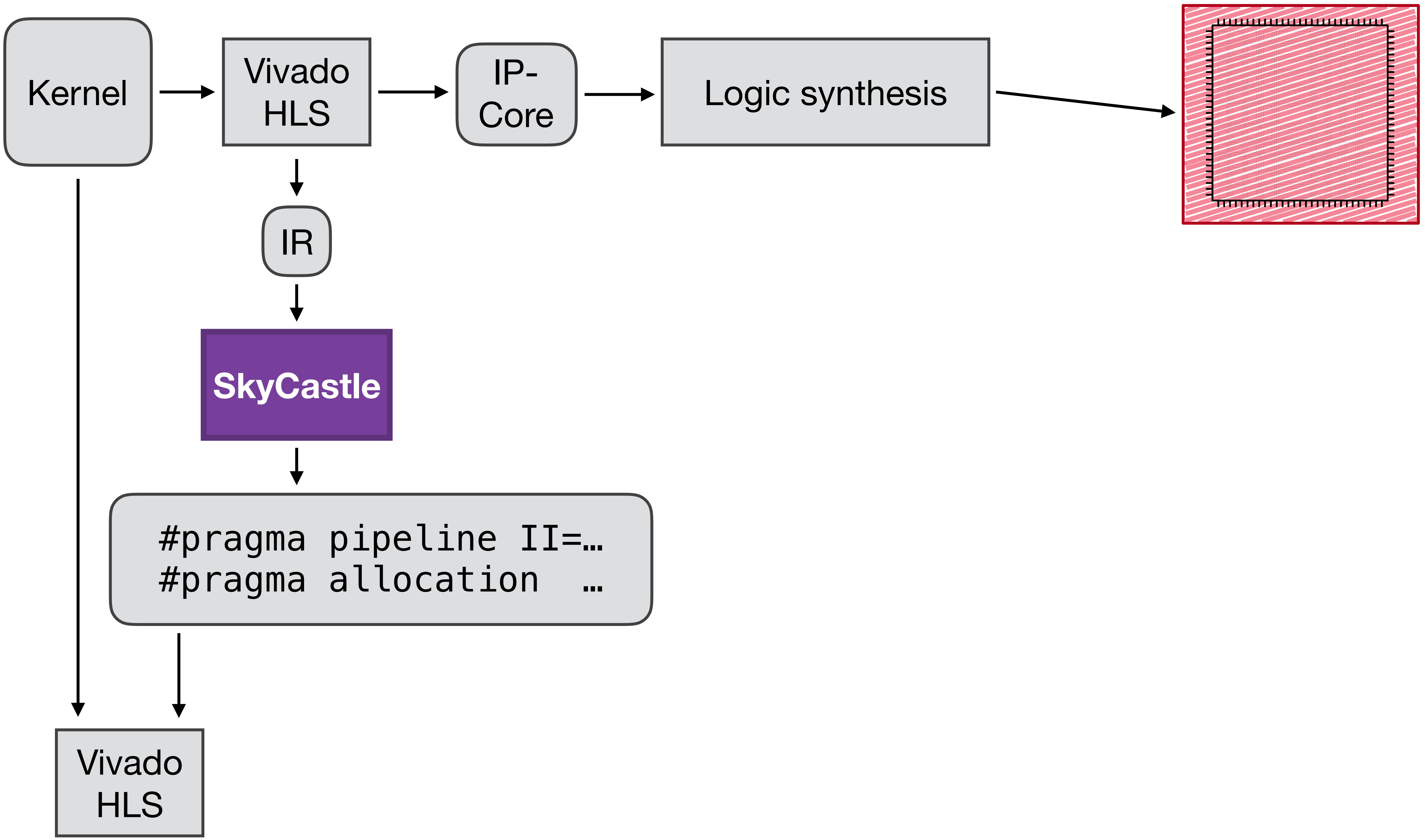


Proposed Flow



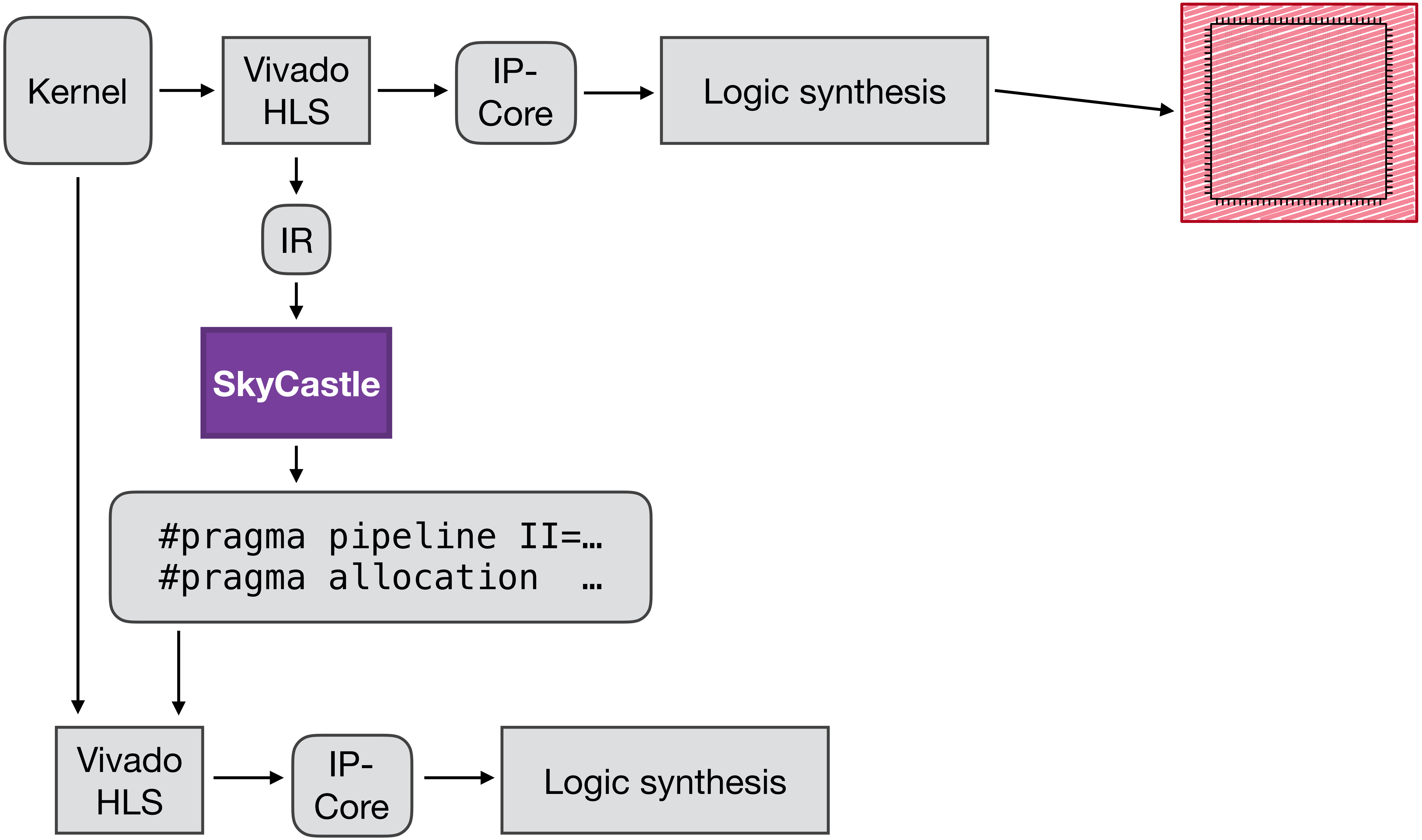
```
#pragma pipeline II=…
#pragma allocation   …
```

Proposed Flow

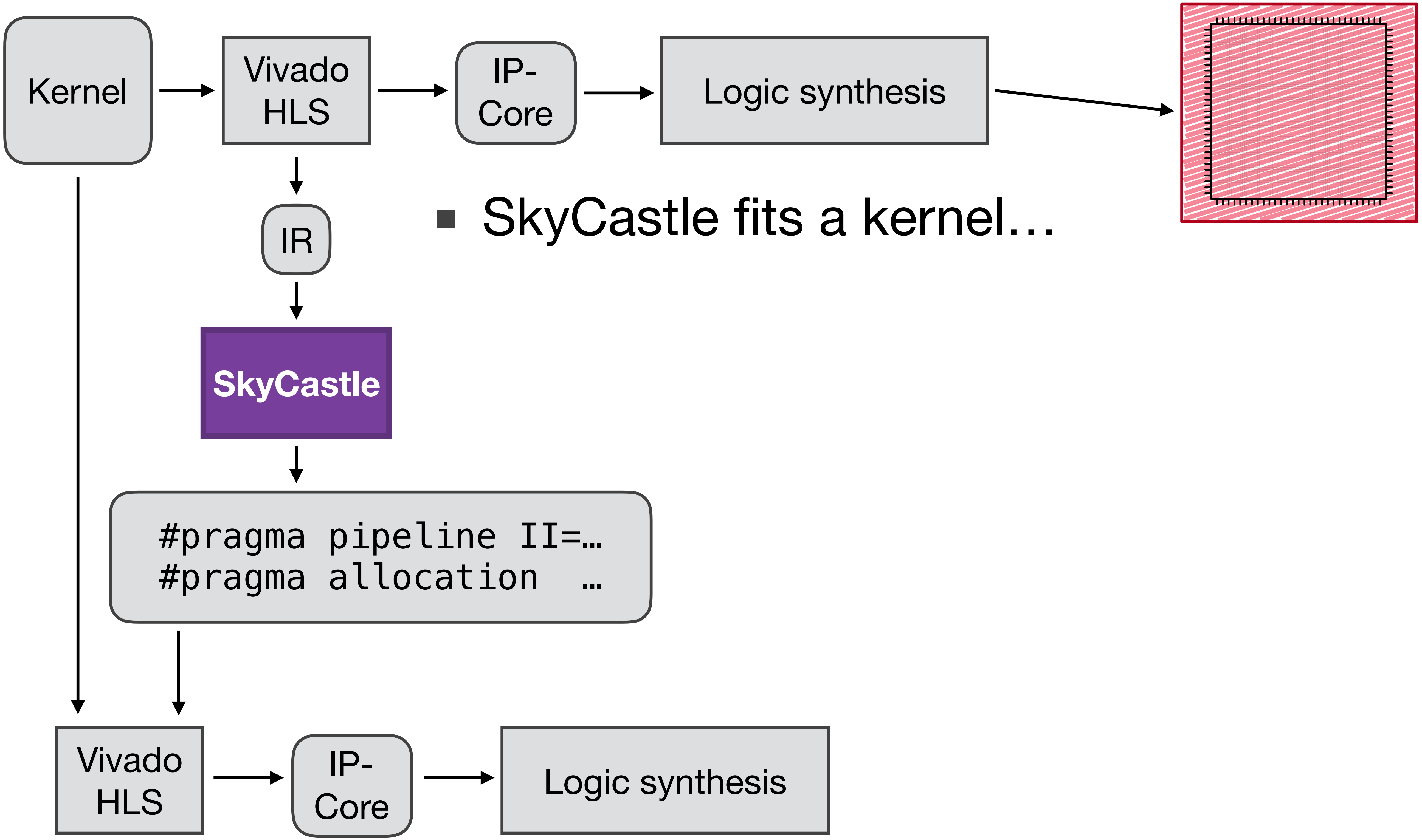


```
#pragma pipeline II=…
#pragma allocation   …
```

Proposed Flow



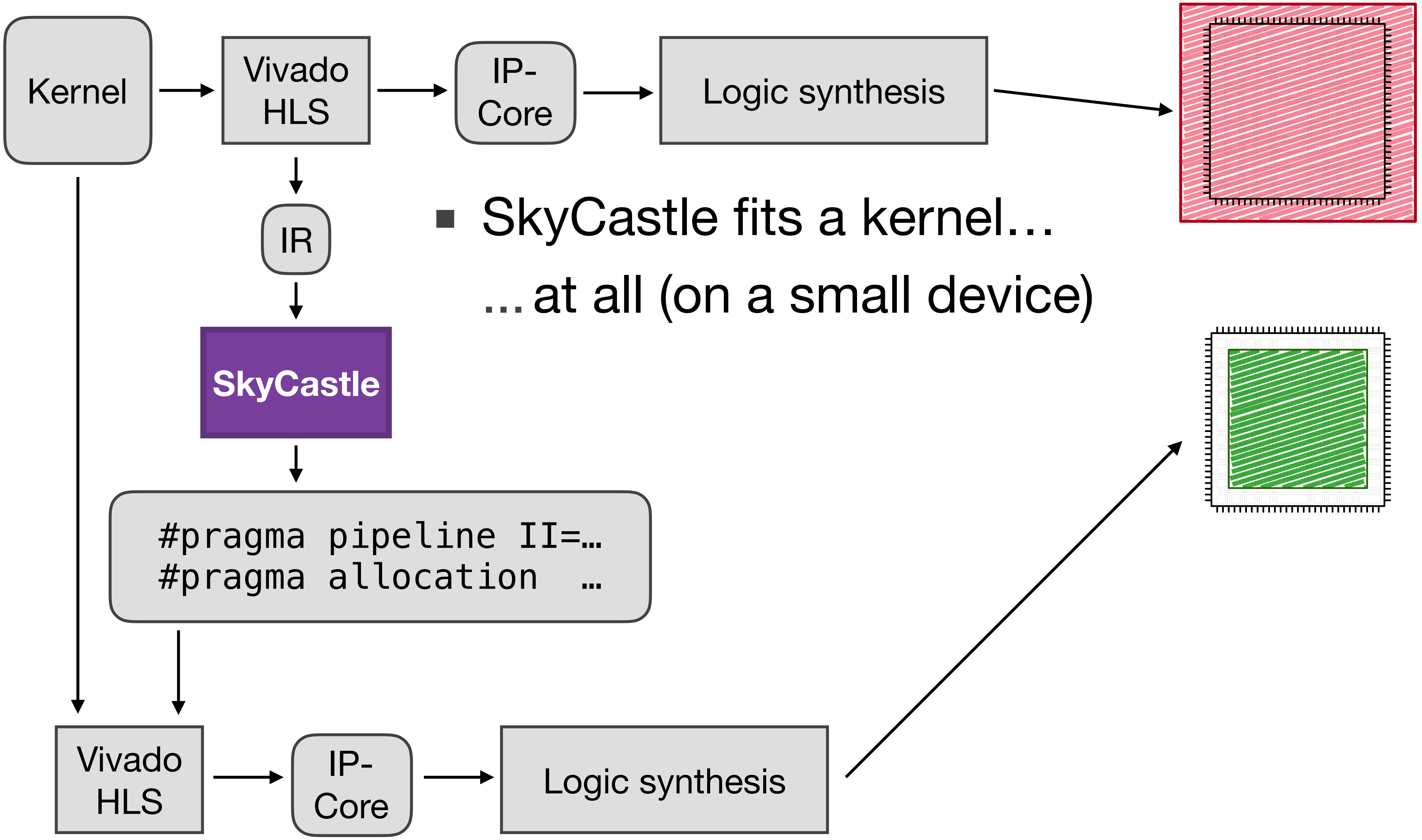
Proposed Flow



```
#pragma pipeline II=…
#pragma allocation   …
```

- SkyCastle fits a kernel…

Proposed Flow

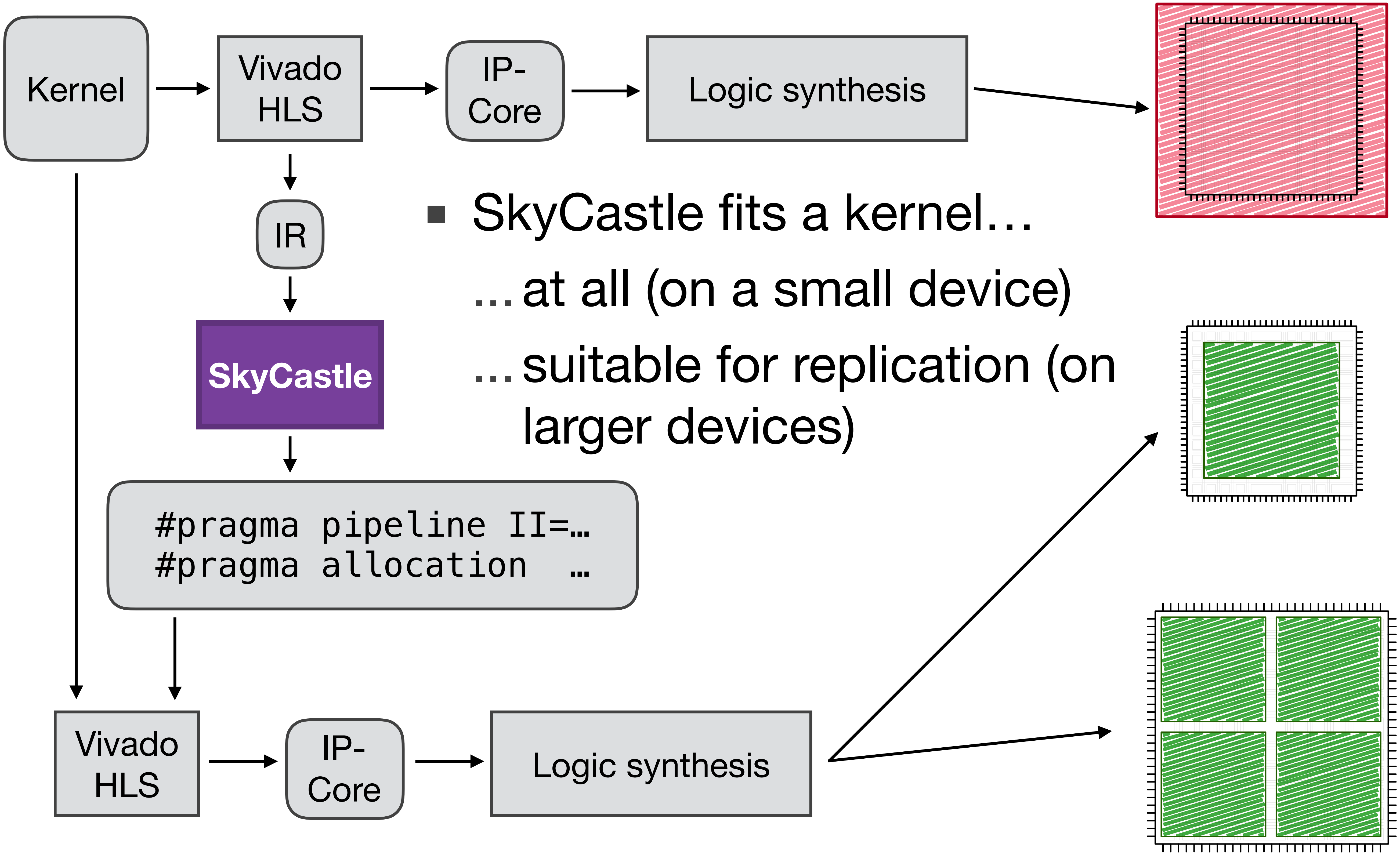


```
#pragma pipeline II=…
#pragma allocation  …
```

- SkyCastle fits a kernel…

  … at all (on a small device)

Proposed Flow

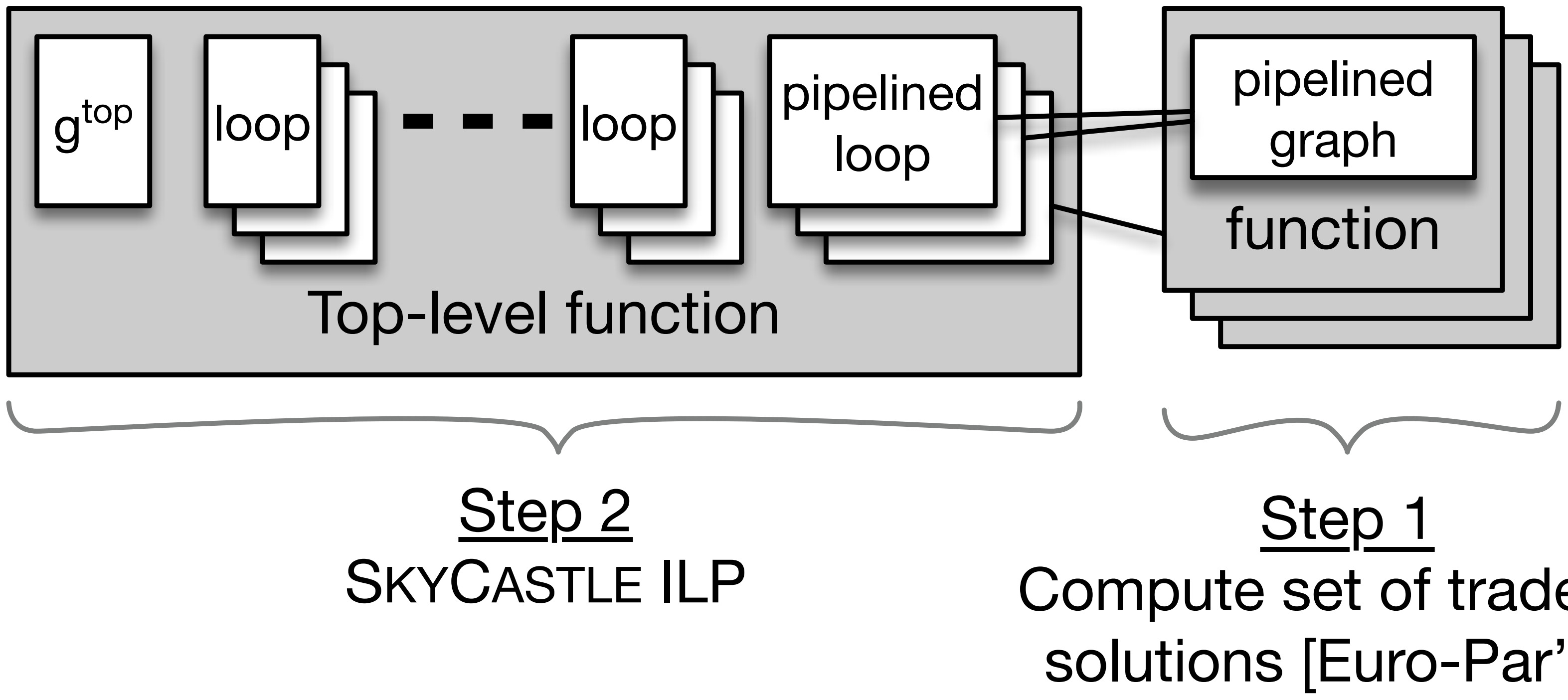


```
#pragma pipeline II=…
#pragma allocation  …
```

- SkyCastle fits a kernel…
  …at all (on a small device)
  …suitable for replication (on larger devices)

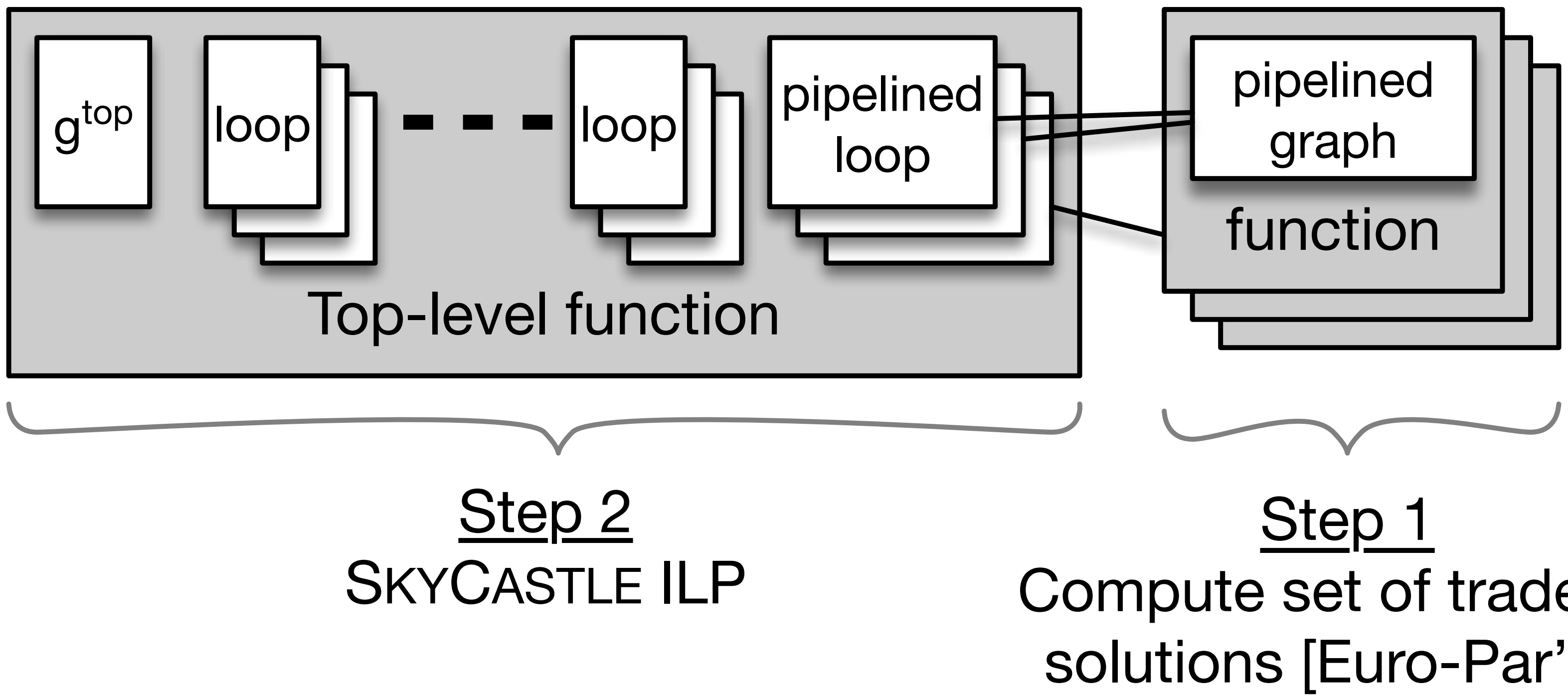
SkyCastle

- Novel **SkyCastle** approach for a subclass of kernels



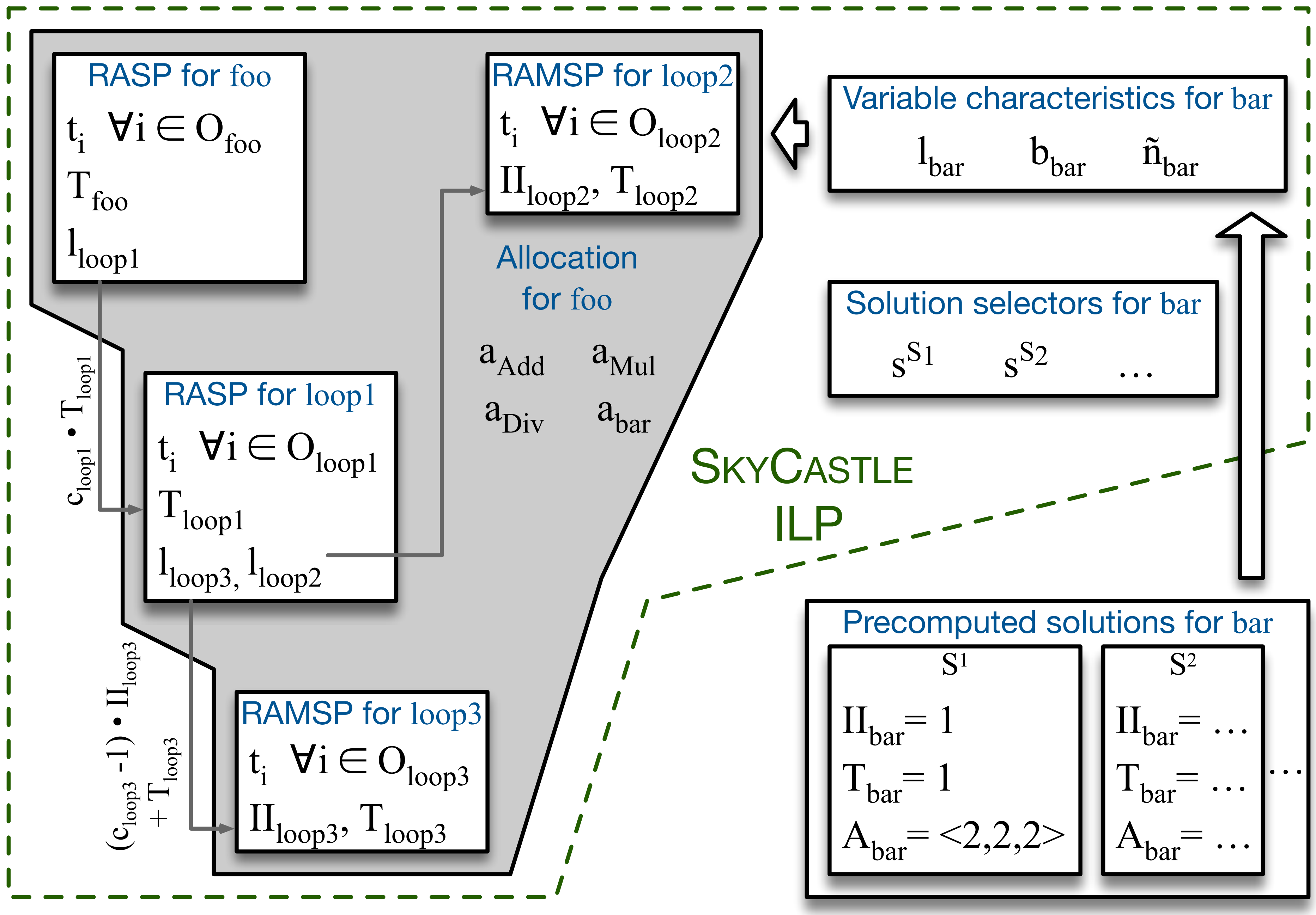
SkyCastle

- Novel **SkyCastle** approach for a subclass of kernels



- = first „level“ of the multi-loop scheduling problem
  - Arbitrary number and nesting structure of loops in top-level function
  - Only innermost loops may be pipelined

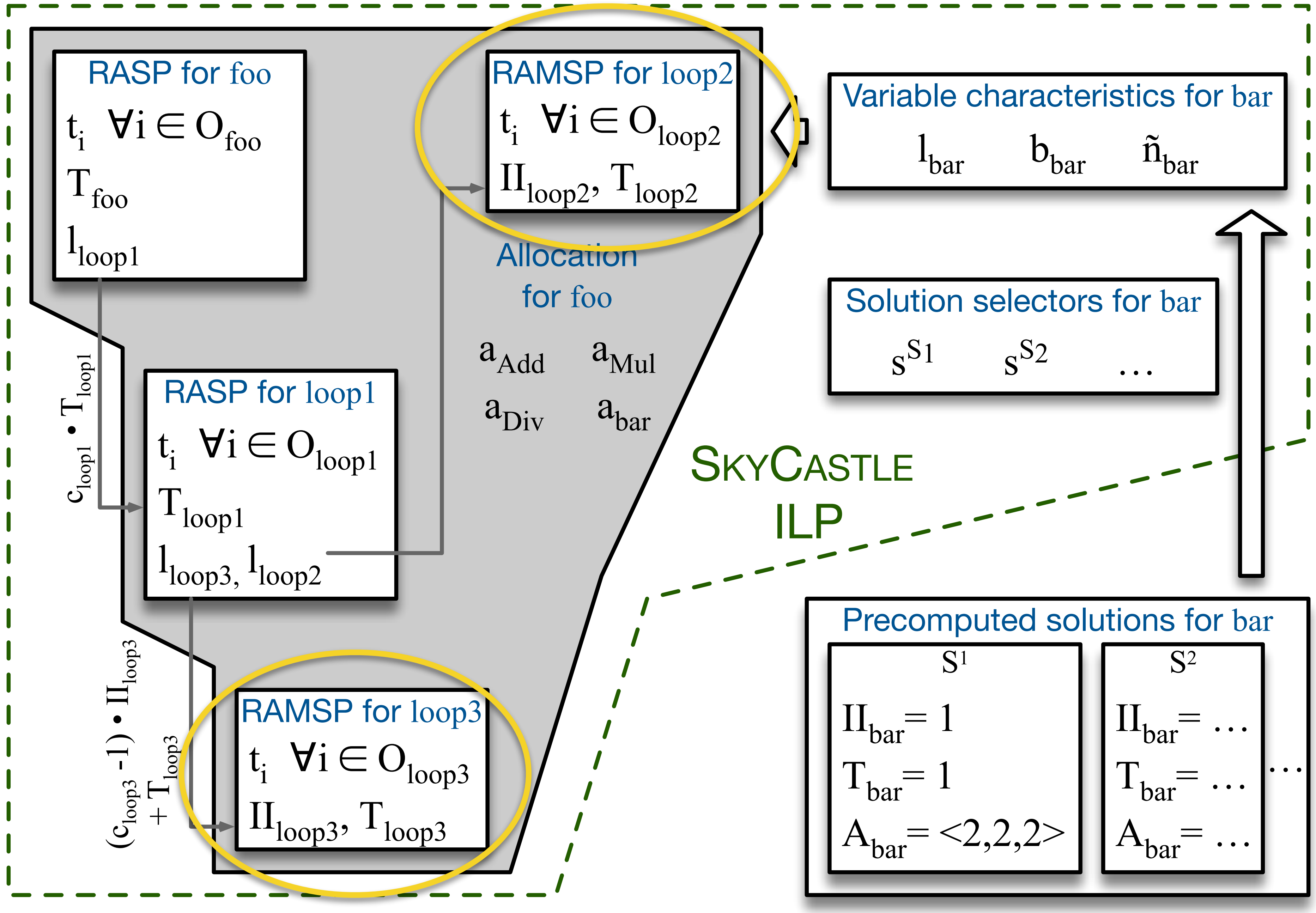
SkyCastle ILP



RASP for foo
$t_i \ \forall i \in O_{foo}$
T_{foo}
l_{loop1}
$c_{loop1} \cdot T_{loop1}$
RASP for loop1
$t_i \ \forall i \in O_{loop1}$
T_{loop1}
l_{loop3}, l_{loop2}
$(c_{loop3} - 1) \cdot II_{loop3} + T_{loop3}$
RAMSP for loop3
$t_i \ \forall i \in O_{loop3}$
II_{loop3}, T_{loop3}
RAMSP for loop2
$t_i \ \forall i \in O_{loop2}$
II_{loop2}, T_{loop2}
Allocation for foo
a_{Add} a_{Mul} a_{Div} a_{bar}
Variable characteristics for bar
l_{bar} b_{bar} $\tilde{n}_{bar}$
Solution selectors for bar
s^{S1} s^{S2} …
SkyCastle ILP
Precomputed solutions for bar
S^1
$II_{bar} = 1$
$T_{bar} = 1$
$A_{bar} = <2,2,2>$
S^2
$II_{bar} = \ldots$
$T_{bar} = \ldots$
$A_{bar} = \ldots$
…

RASP = Resource-Aware Scheduling Problem
RAMSP = Resource-Aware **Modulo** Scheduling Problem

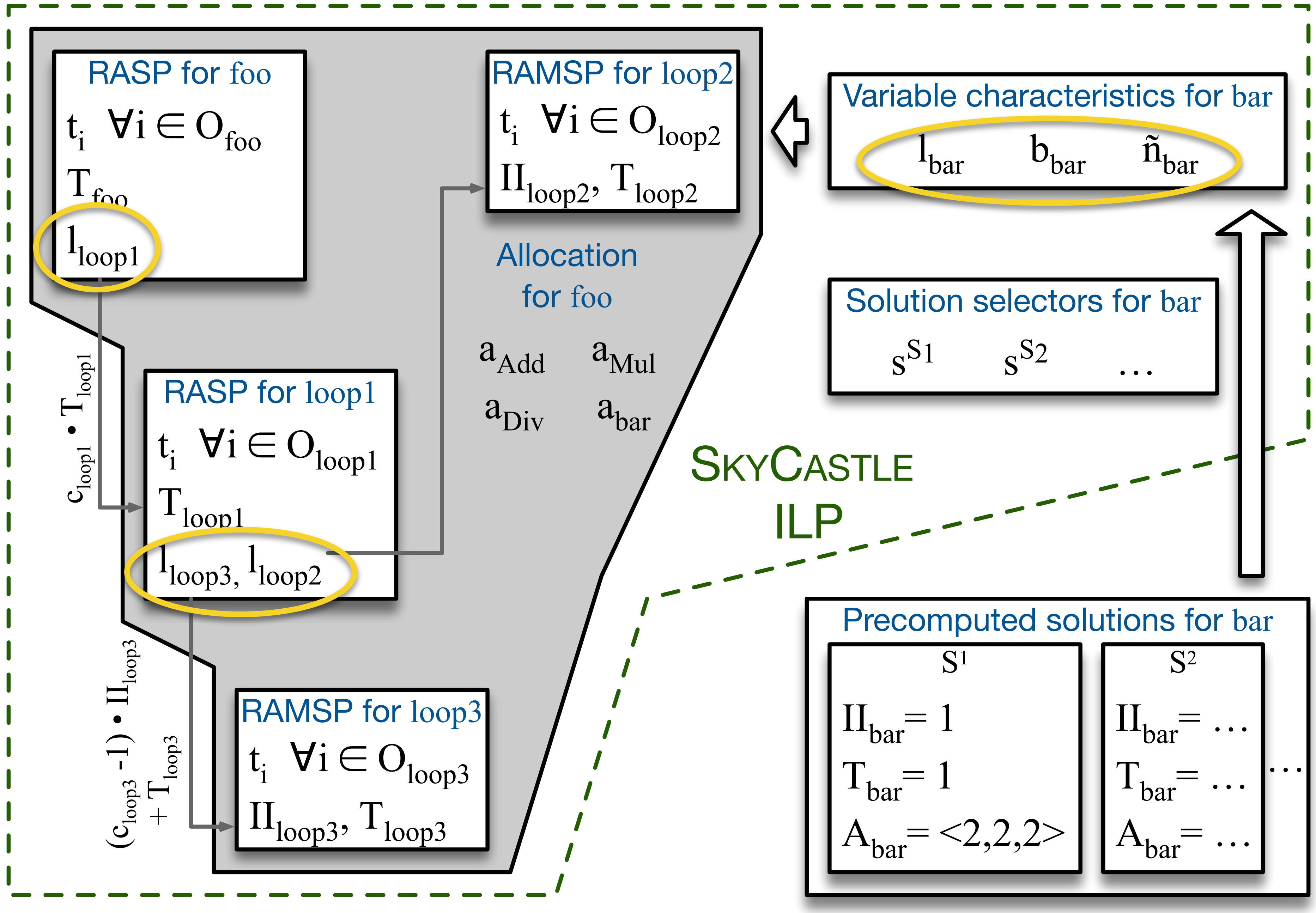
SkyCastle ILP



RASP = Resource-Aware Scheduling Problem
RAMSP = Resource-Aware **Modulo** Scheduling Problem

- Uses Moovac formulation [TRETS'19]
  - II is decision variable

SkyCastle ILP

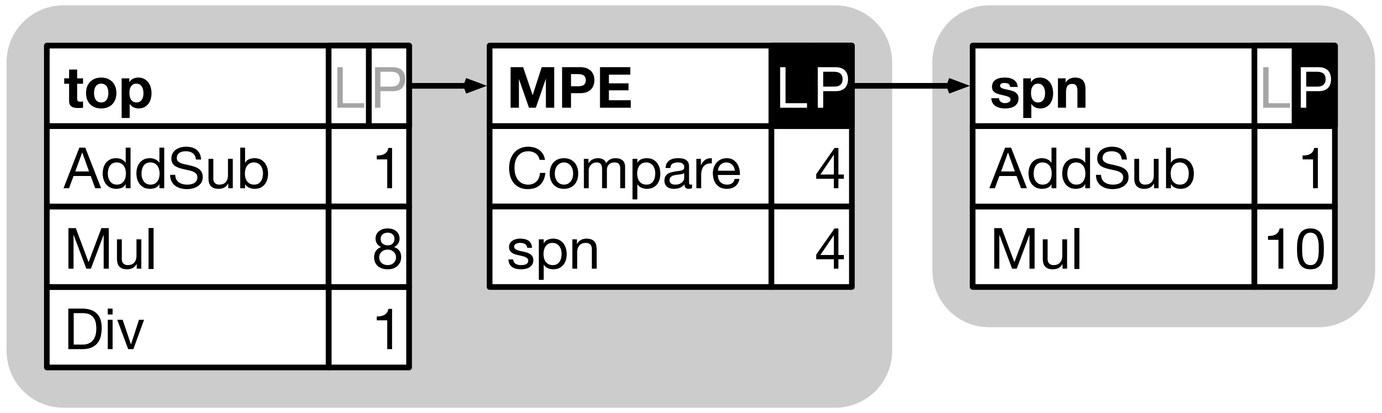


- Uses Moovac formulation [TRETS'19]
  - II is decision variable
- characteristics are variable for a subset of operations/ operators

RASP = Resource-Aware Scheduling Problem
RAMSP = Resource-Aware **Modulo** Scheduling Problem

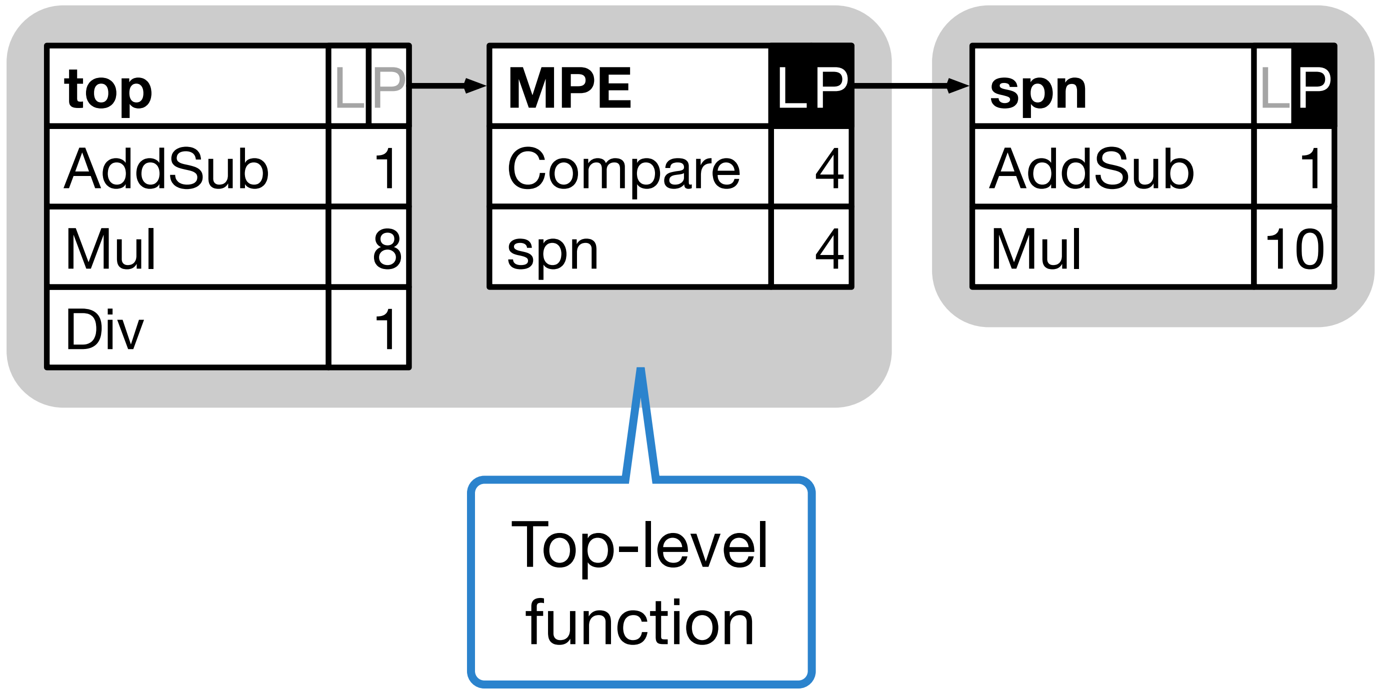
Evaluation: Case study „SPN“

- Different queries of a Sum-Product Network
  - Motivational example from the first slide



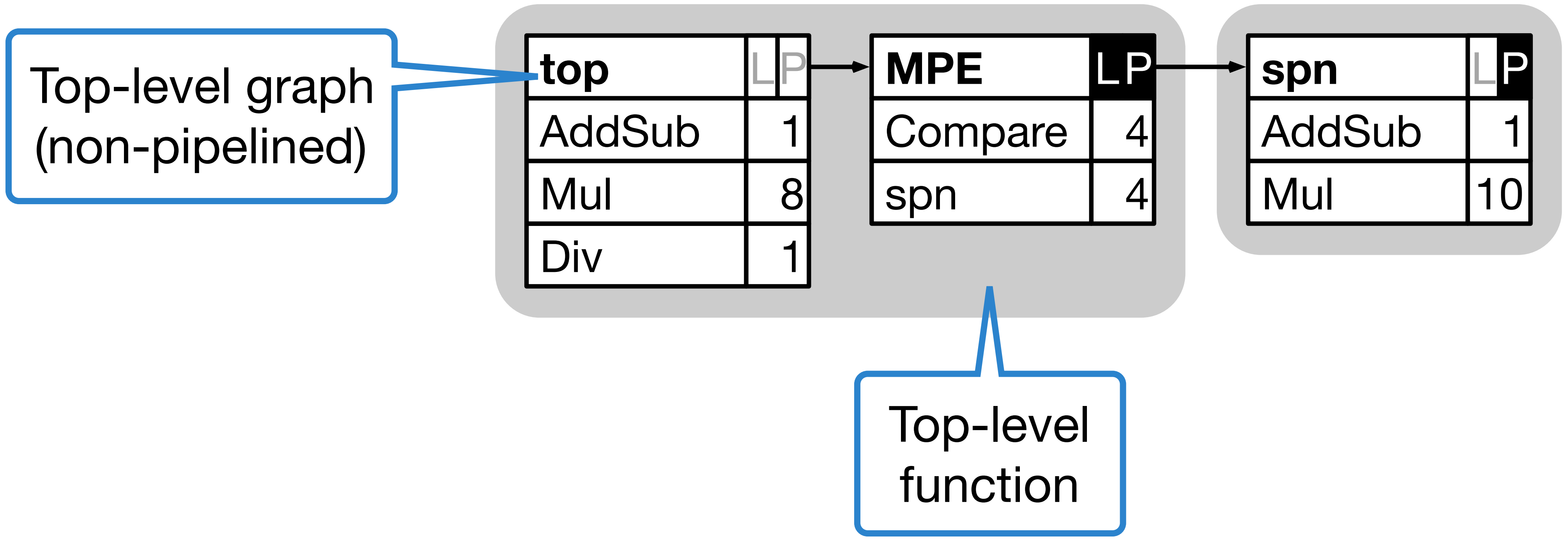
Evaluation: Case study „SPN“

- Different queries of a Sum-Product Network
  - Motivational example from the first slide



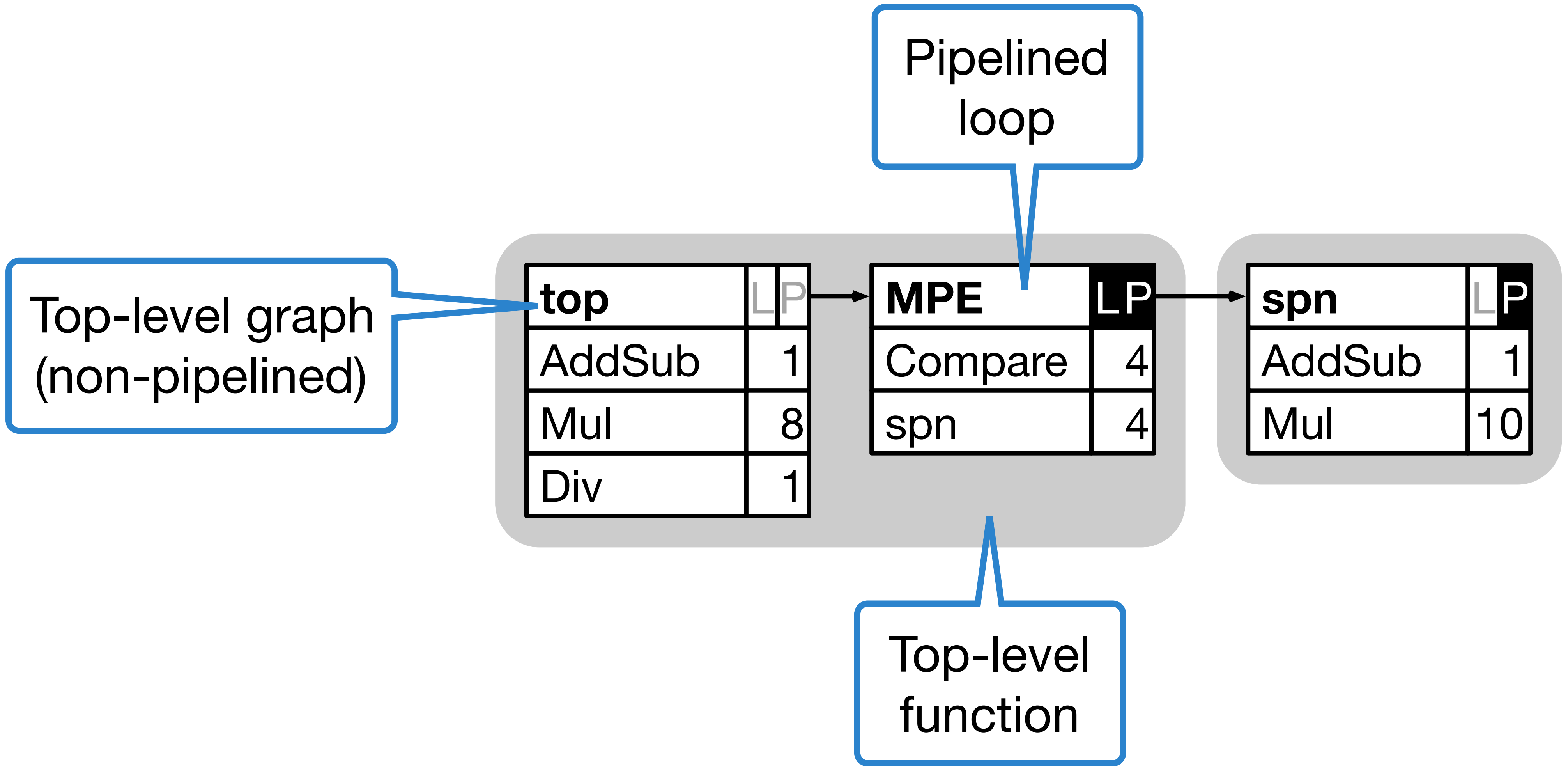
Evaluation: Case study „SPN“

- Different queries of a Sum-Product Network
  - Motivational example from the first slide



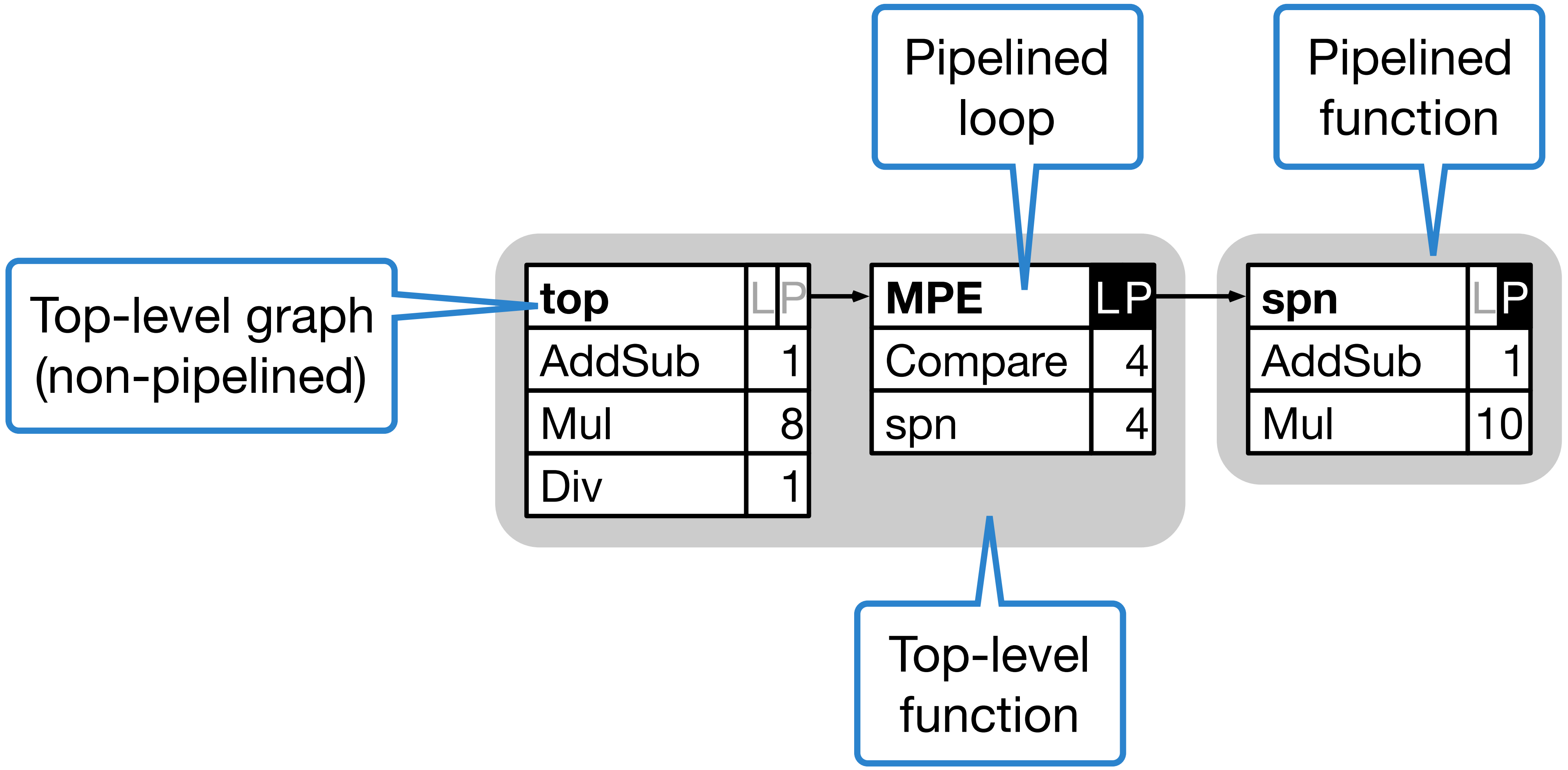
Evaluation: Case study „SPN"

- Different queries of a Sum-Product Network
  - Motivational example from the first slide

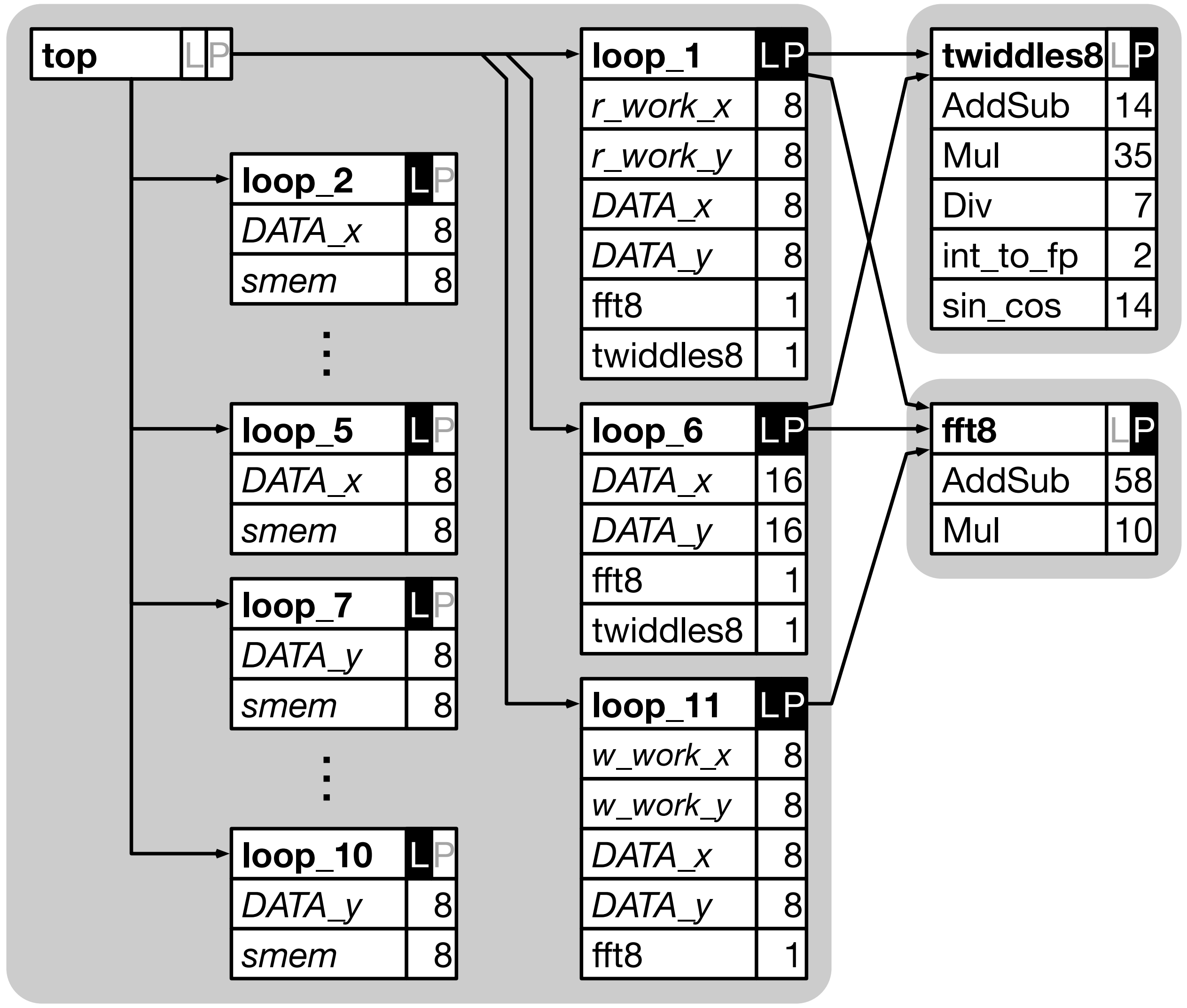


Evaluation: Case study „SPN“

- Different queries of a Sum-Product Network
  - Motivational example from the first slide

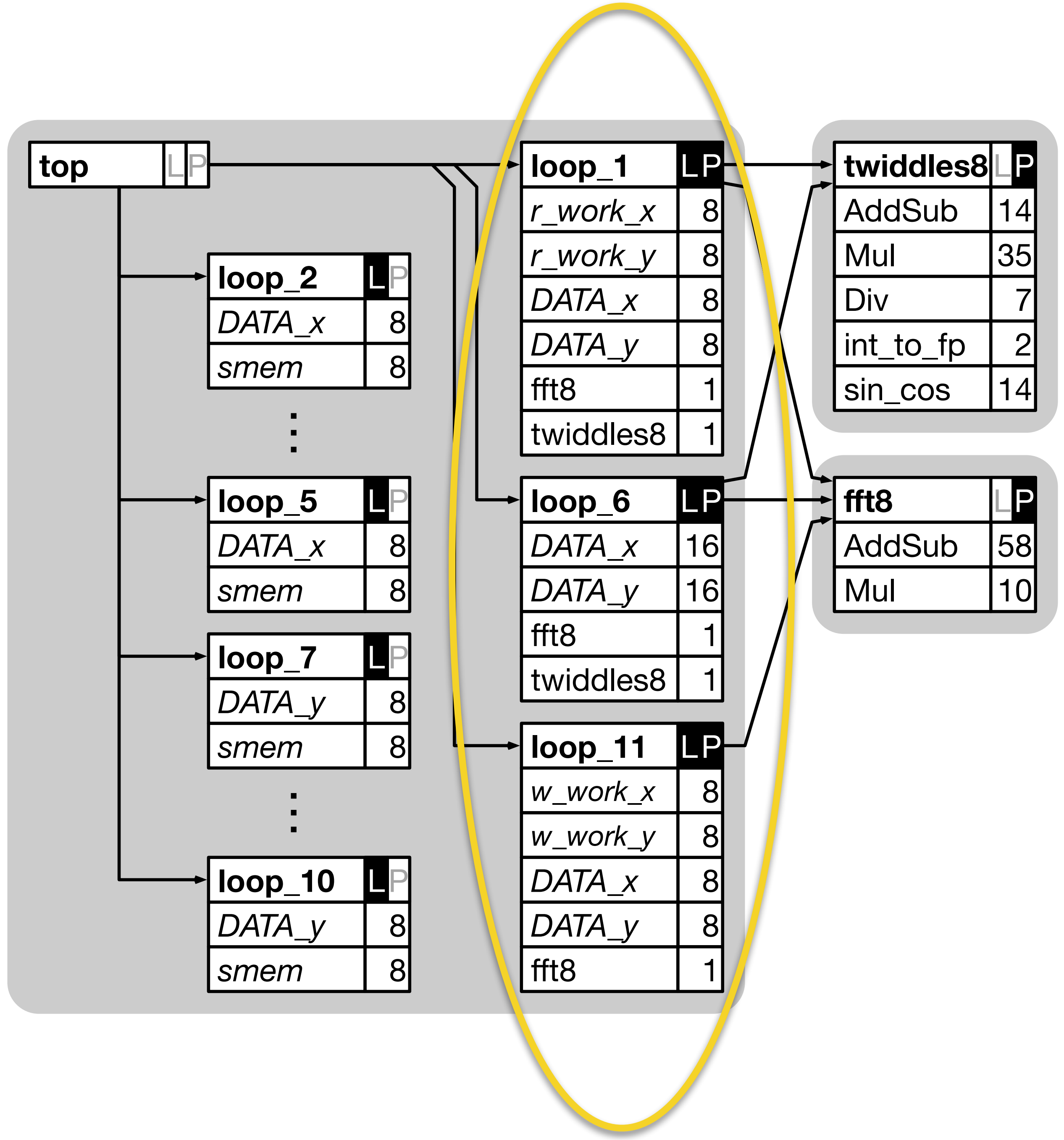


Evaluation: Case study „FFT“



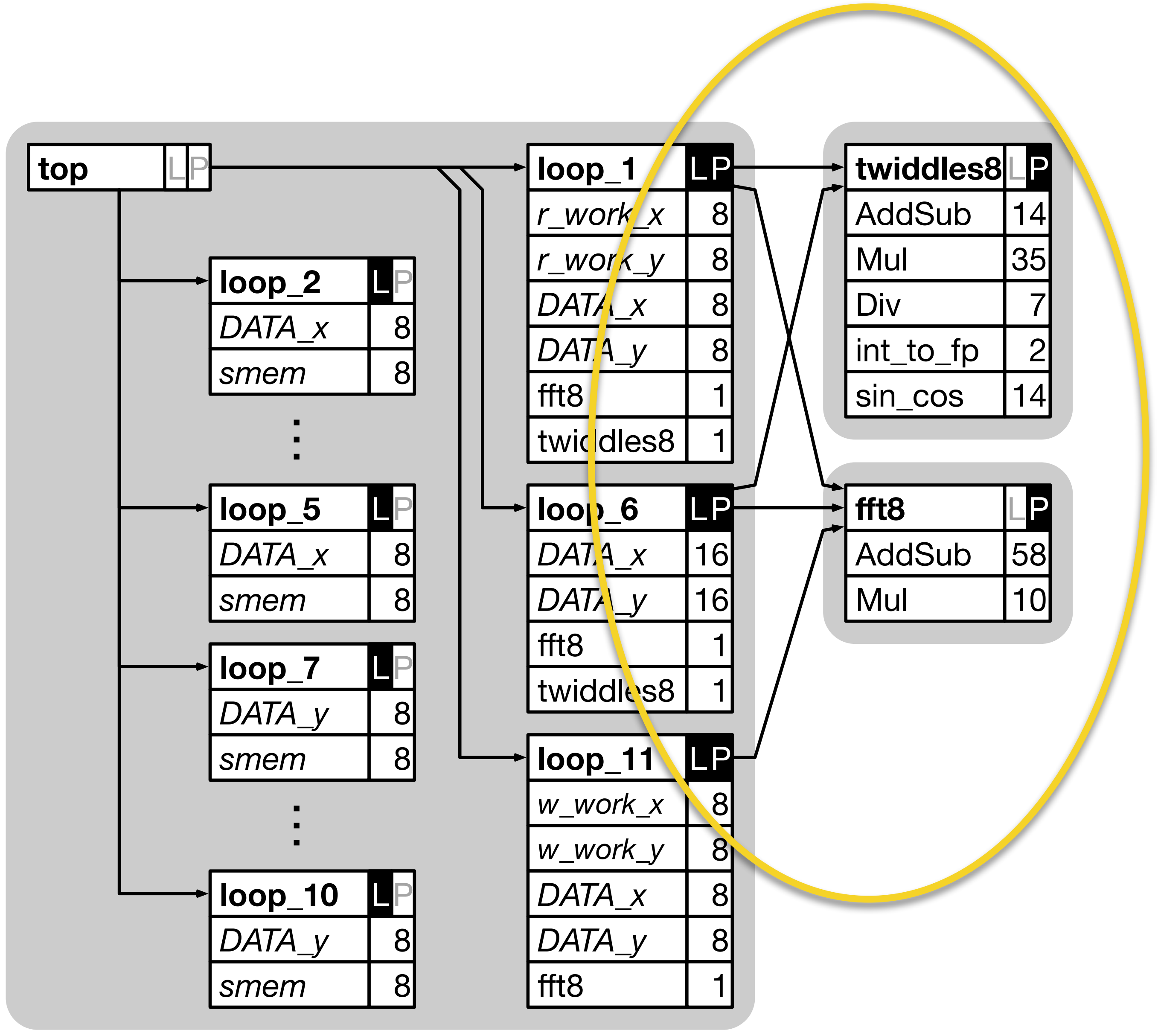
- Fast Fourier Transformation
  - code from MachSuite/ fft_transpose

Evaluation: Case study „FFT“



- Fast Fourier Transformation
  - code from MachSuite/ fft_transpose
  - 3 of 11 loops are pipelined

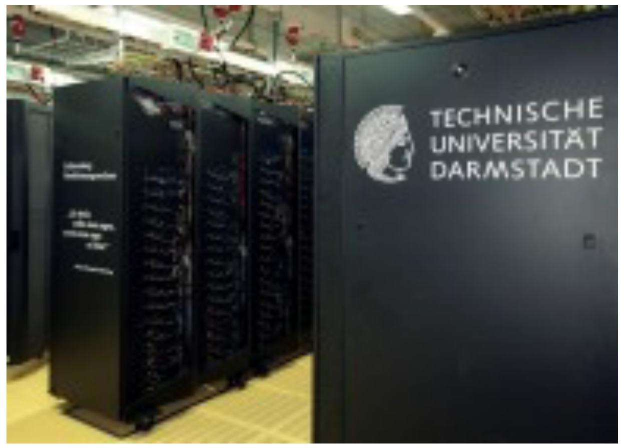
Evaluation: Case study „FFT“



- Fast Fourier Transformation
  - code from MachSuite/ fft_transpose
  - 3 of 11 loops are pipelined
  - Pipelined functions are called from different loops
    → consider II divisibility

Evaluation: Setup

- **Gurobi 8.1, 8 threads, 16 GiB RAM**
  on Xeon E5-2680 v3 servers @ 2.8 GHz

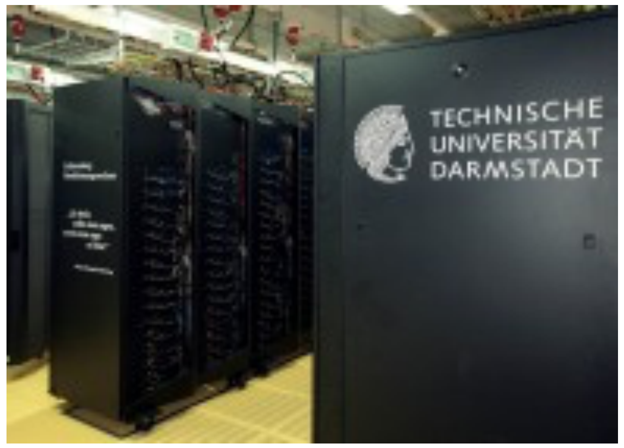


src: HHLR TU-DA

Evaluation: Setup

- **Gurobi 8.1, 8 threads, 16 GiB RAM**
  on Xeon E5-2680 v3 servers @ 2.8 GHz

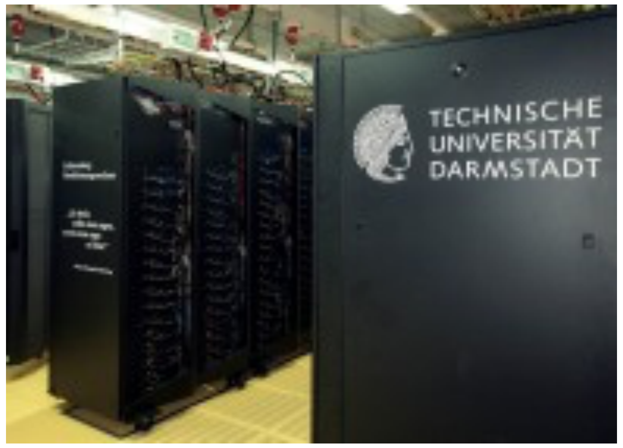
- **Timelimits** (for solving ILP)
  15 min: minimise latency
  5 min: minimise resource utilisation



src: HHLR TU-DA

Evaluation: Setup

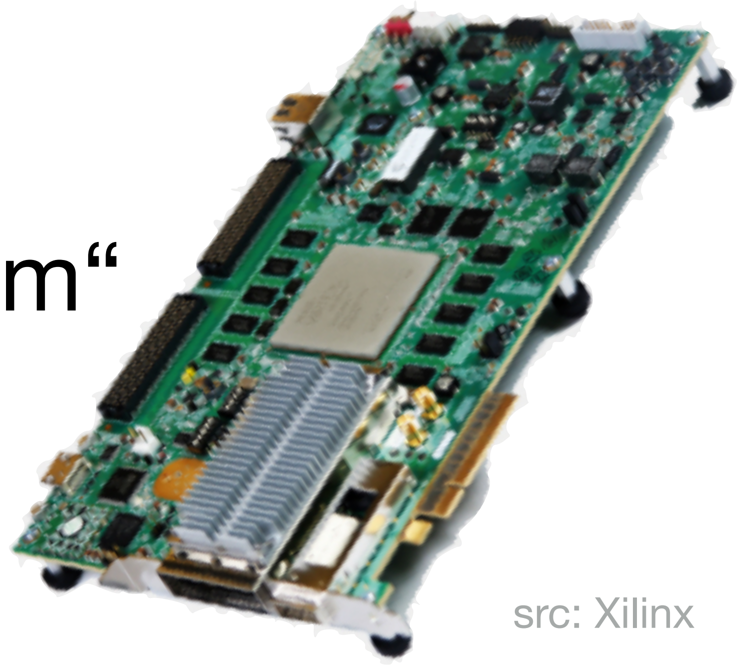
- **Gurobi 8.1, 8 threads, 16 GiB RAM**
  on Xeon E5-2680 v3 servers @ 2.8 GHz



src: HHLR TU-DA

- **Timelimits** (for solving ILP)
  15 min: minimise latency
   5 min: minimise resource utilisation

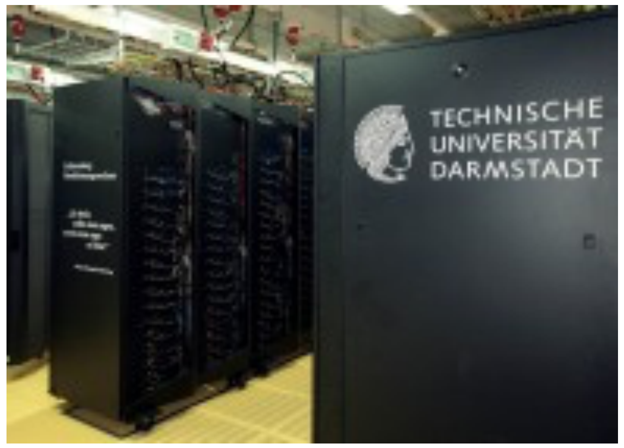
- **2 FPGA boards**
  - ZedBoard — XC7Z020 — „small“
  - VCU108 — XCVU095 — „medium“



src: Xilinx

Evaluation: Setup

- **Gurobi 8.1, 8 threads, 16 GiB RAM**
  on Xeon E5-2680 v3 servers @ 2.8 GHz

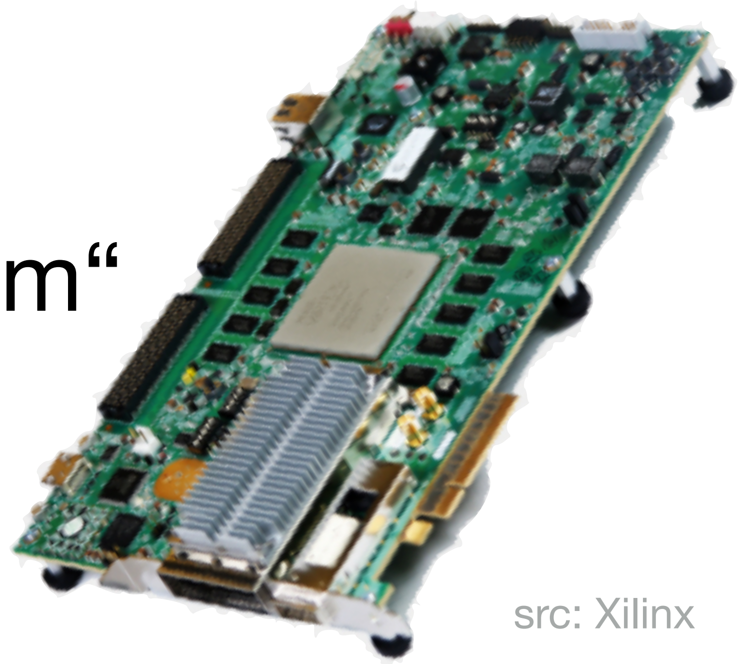


src: HHLR TU-DA

- **Timelimits** (for solving ILP)
  15 min: minimise latency
  5 min: minimise resource utilisation

- **2 FPGA boards**
  - ZedBoard — `XC7Z020` — „small“
  - VCU108 — `XCVU095` — „medium“

- Xilinx Vivado HLS 2018.3


src: Xilinx

Evaluation: Key Insights

- The next slides illustrate that …

Evaluation: Key Insights

- The next slides illustrate that …

  ... solving the ILP is **tractable**

Evaluation: Key Insights

- The next slides illustrate that …

  ... solving the ILP is **tractable**

  ... we **capture** the most important **aspects** of the Vivado HLS **scheduling & allocation problem**

Evaluation: Key Insights

- The next slides illustrate that …

  ... solving the ILP is **tractable**

  ... we **capture** the most important **aspects** of the Vivado HLS **scheduling & allocation problem**

  ... using SkyCastle leads to **synthesisable designs**

Evaluation: Key Insights

- The next slides illustrate that …

  ... solving the ILP is **tractable**

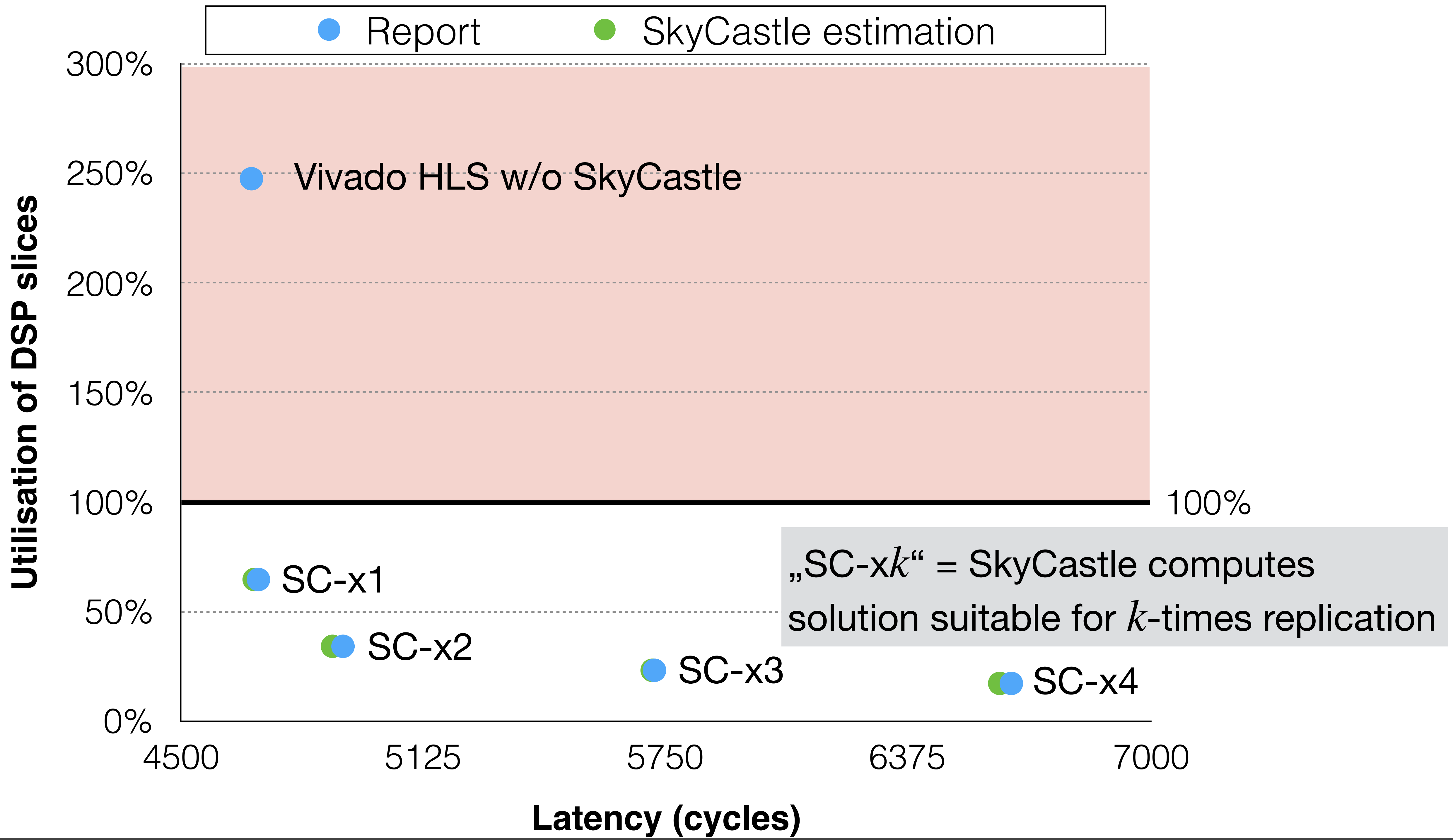
  ... we **capture** the most important **aspects** of the Vivado HLS **scheduling & allocation problem**

  ... using SkyCastle leads to **synthesisable designs**

  ... we expect **improved throughput** from **replicating** slower-but-smaller kernel implementations

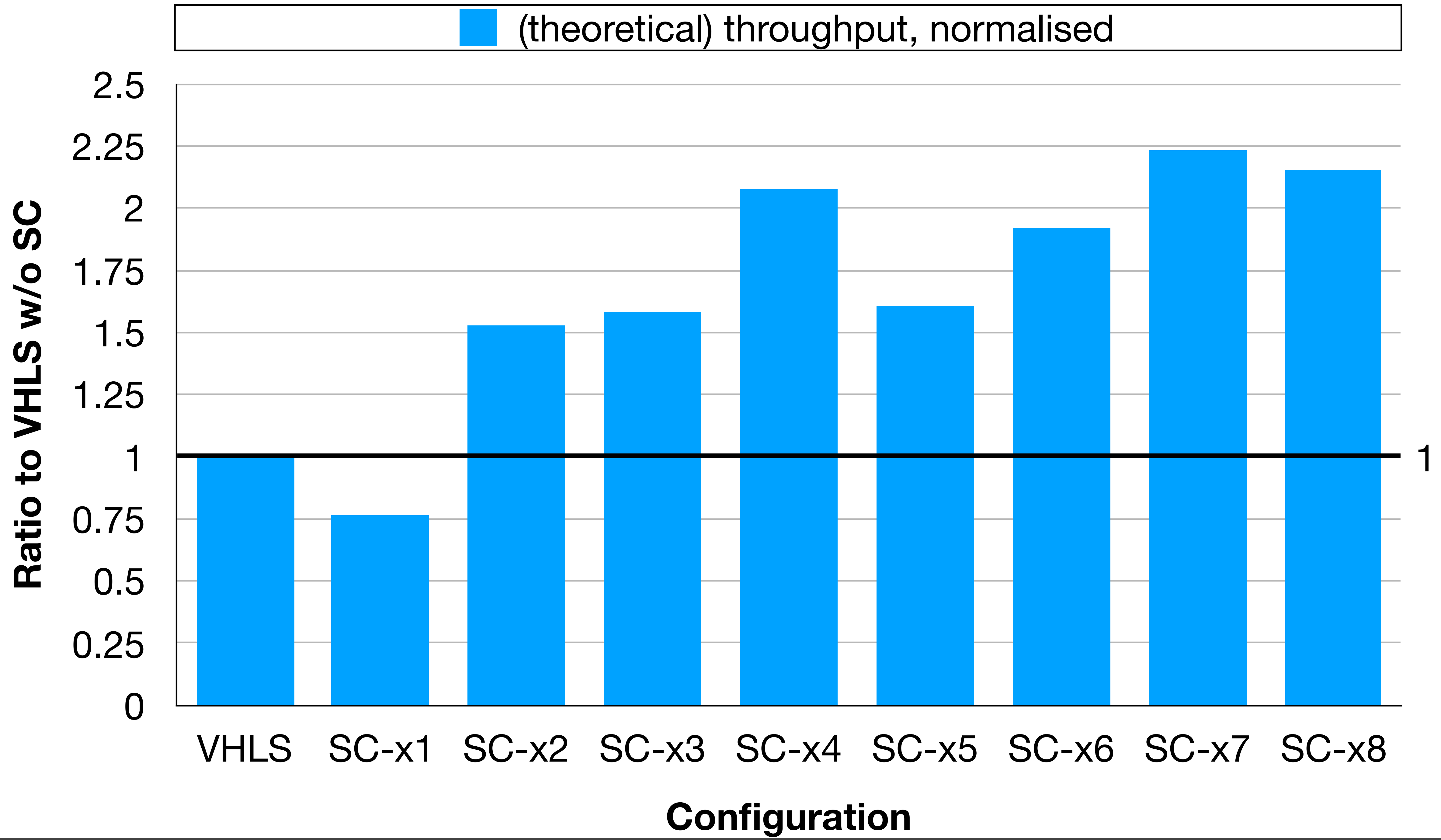
Results — Trade-off Solutions

- **FFT**, VCU108



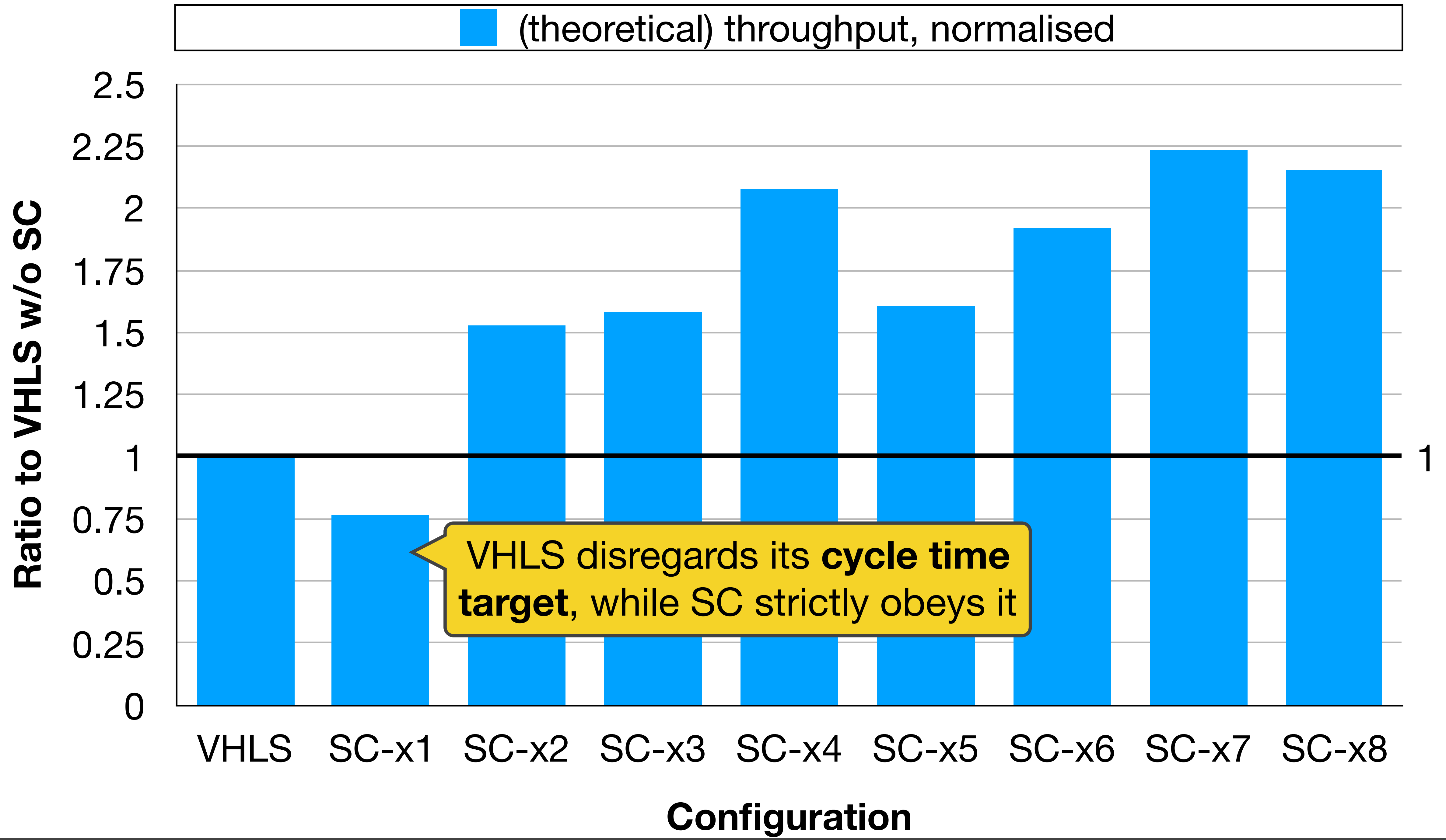
Results — Replication

- **SPN**, VCU108



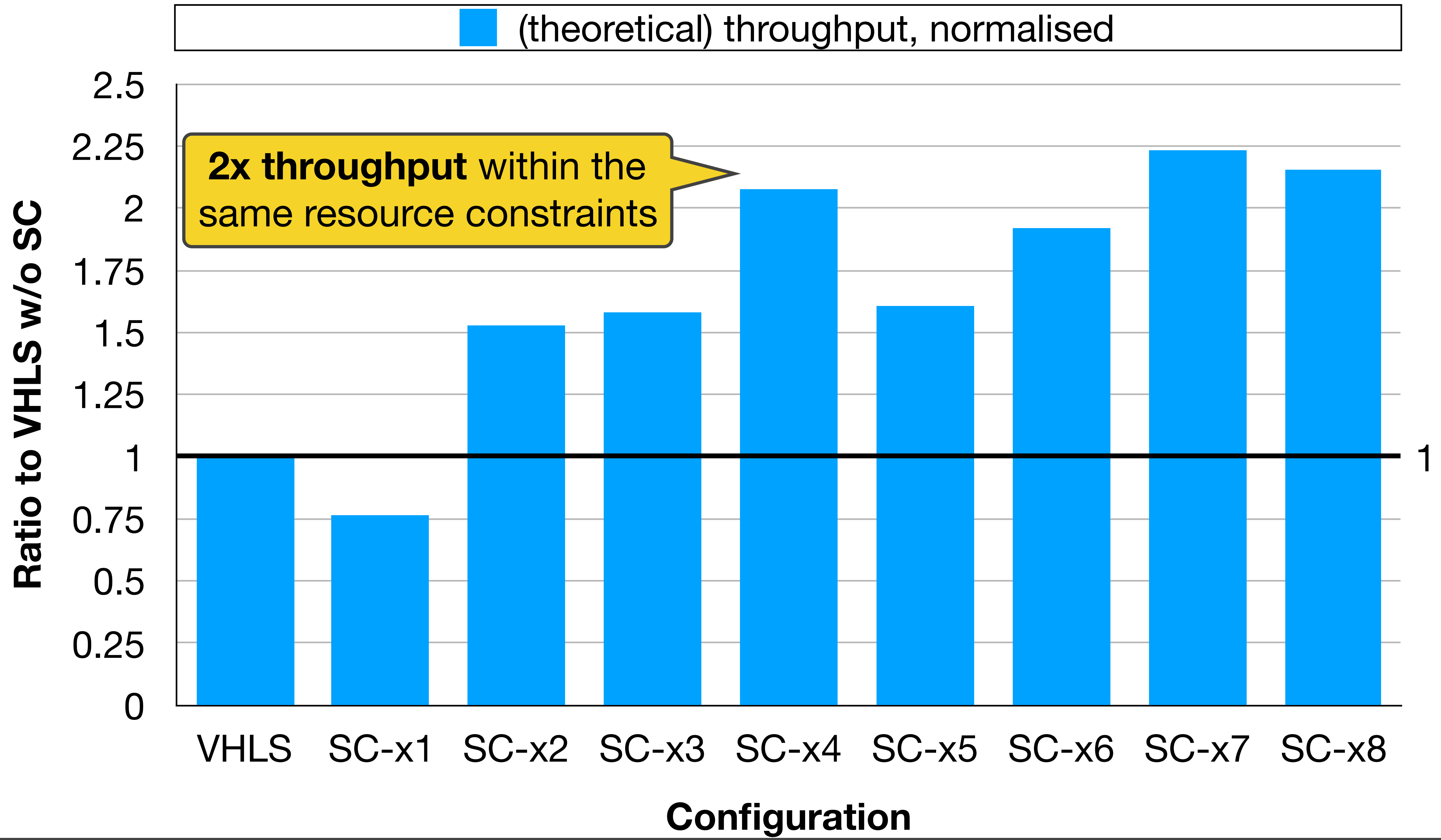
Results — Replication

- **SPN**, VCU108



Results — Replication

- **SPN**, VCU108



Conclusion / Outlook

- New approach to automatic hardware design using mathematical optimisation

Conclusion / Outlook

- New approach to automatic hardware design using mathematical optimisation
- Would benefit tremendously from public interface into the HLS steps (similar to RapidWright)

Conclusion / Outlook

- New approach to automatic hardware design using mathematical optimisation
- Would benefit tremendously from public interface into the HLS steps (similar to RapidWright)
- Could be a key ingredient to the automatic design-space exploration of multi-kernel OpenCL applications