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

Julian Oppermann¹, Lukas Sommer¹, Lukas Weber¹, Melanie Reuter-Oppermann², Andreas Koch¹, Oliver Sinnen³
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 microarchitecture that still fits on the device?
A Common Problem

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

```c
{ /* 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 microarchitecture that still fits on the device?

```c
{ /* 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

HLS

src: Xilinx
A Common Problem

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

```c
{ /* 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

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

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

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

```verilog
wire r_mul;
assign r_mul = r_load_1 * r_load_2;
```
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
  \( \text{think: C code} \)

- 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

- 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

- 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]

![Diagram of a modulo scheduler with a resource-aware allocation for II = 1, showing allocation for ADD and MUL operations.]

Highest throughput

Least resource demand
Trade-offs

- For one loop, trade-offs can be computed with a resource-aware modulo scheduler [Euro-Par’19]
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 that!
Multi-Loop Scheduling Problem

Function: foo

Graph: foo

Graph: loop 1

Graph: loop 2

Graph: loop 3

Function: bar

Graph: bar

Operations:
- Add
- Mul
- Div
- Cmp

Operator types:
- Add
- Mul
- Div
- Cmp
Objective: **Minimise** latency of top-level function
Multi-Loop Scheduling Problem

Objective: \textbf{Minimise} latency of top-level function

Nested scheduling problems: \textbf{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

Operations:
- Add
- Mul
- Div
- Cmp

Operator types:
- Add
- Mul
- Div
- Cmp

Normal:
- Add
- Mul
- Cmp

Loop:
- Add
- Mul

Dependence edges:

Graph reference:
- Graph: loop 3

Variable latency is derived from the scheduling result of the referenced graph

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

$$\text{latency}_x = (\text{trip\_count}_{\text{loop}2} - 1) \times I_{\text{loop}2} + \text{schedule\_length}_{\text{loop}2}$$
Multi-Loop Scheduling Problem

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

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

Nested scheduling problems: **Variable latency** is derived from the scheduling result of the referenced graph
Multi-Loop Scheduling Problem

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

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

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

**Variable latency**, variable II, and variable resource demands, derived from scheduling „bar“

**Function:** foo

**Graph:** foo

**Graph:** loop 1

**Graph:** loop 2

**Graph:** loop 3

**Function:** bar

**Graph:** bar

**Operator types:**
- Add
- Mul
- Div
- Cmp

**Operations:**
- Add
- Mul
- Cmp

**Dependence edges:**

**Graph reference:**
- Graph: loop 1
- Graph: loop 2
- Graph: loop 3
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 function 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

Kernel
Proposed Flow

Kernel → Vivado HLS
Proposed Flow

Kernel → Vivado HLS → IP-Core → Logic synthesis
Proposed Flow

- Kernel
- Vivado HLS
- IP-Core
- Logic synthesis
Proposed Flow

Kernel → Vivado HLS → IP-Core → Logic synthesis

IR → SkyCastle
Proposed Flow

Kernel → Vivado HLS → IP-Core → Logic synthesis

IR

SkyCastle

#pragma pipeline II=...
#pragma allocation ...
Proposed Flow

Kernel → Vivado HLS → IP-Core → Logic synthesis

IR

SkyCastle

#pragma pipeline II=...
#pragma allocation ...

Vivado HLS
Proposed Flow

Kernel → Vivado HLS → IP-Core → Logic synthesis

IR → SkyCastle

#pragma pipeline II=...
#pragma allocation ...

Vivado HLS → IP-Core → Logic synthesis
Proposed Flow

- **SkyCastle fits a kernel**...
Proposed Flow

- SkyCastle fits a kernel... 
  ...at all (on a small device)
SkyCastle fits a kernel...

...at all (on a small device)

...suitable for replication (on larger devices)
- **Novel SkyCastle** approach for a subclass of kernels

**Figure:**

- **Step 1:** Compute set of trade-off solutions [Euro-Par’19]
- **Step 2:** SKYCASTLE ILP

**Diagram:**

- Top-level function
- \( g^{top} \)
- Loop
- Pipelined loop
- Pipelined graph function
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
\[ t_i \quad \forall i \in O_{\text{foo}} \]
\[ T_{\text{foo}} \]
\[ l_{\text{loop1}} \]

RASP for loop1
\[ t_i \quad \forall i \in O_{\text{loop1}} \]
\[ T_{\text{loop1}} \]
\[ l_{\text{loop1}}, l_{\text{loop2}} \]

RAMSP for loop2
\[ t_i \quad \forall i \in O_{\text{loop2}} \]
\[ II_{\text{loop2}}, T_{\text{loop2}} \]

Allocation for foo
\[ a_{\text{Add}} \quad a_{\text{Mul}} \quad a_{\text{Div}} \quad a_{\text{bar}} \]

Variable characteristics for bar
\[ l_{\text{bar}} \quad b_{\text{bar}} \quad \tilde{n}_{\text{bar}} \]

Solution selectors for bar
\[ S^{S1} \quad S^{S2} \quad \ldots \]

Precomputed solutions for bar
\[ S^1 \]
\[ II_{\text{bar}} = 1 \]
\[ T_{\text{bar}} = 1 \]
\[ A_{\text{bar}} = \langle 2, 2, 2 \rangle \]

SkyCastle ILP

RASP = Resource-Aware Scheduling Problem

RAMSP = Resource-Aware Modulo Scheduling Problem
SkyCastle ILP

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

RASP for foo
\[
t_i \quad \forall i \in O_{foo} \\
T_{foo} \\
l_{loop1}
\]

RAMSP for loop2
\[
t_i \quad \forall i \in O_{loop2} \\
II_{loop2}, T_{loop2}
\]

Allocation for foo
\[
a_{Add} \quad a_{Mul} \\
a_{Div} \quad a_{bar}
\]

Variable characteristics for bar
\[
l_{bar} \quad b_{bar} \quad \tilde{n}_{bar}
\]

Solution selectors for bar
\[
s^{S1} \quad s^{S2} \quad \ldots
\]

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

O. Sinnen, University of Auckland — SkyCastle: A Resource-Aware Multi-Loop Scheduler for High-Level Synthesis
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
Different queries of a Sum-Product Network
  • Motivational example from the first slide

<table>
<thead>
<tr>
<th>top</th>
<th>LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddSub</td>
<td>1</td>
</tr>
<tr>
<td>Mul</td>
<td>8</td>
</tr>
<tr>
<td>Div</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPE</th>
<th>LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>4</td>
</tr>
<tr>
<td>spn</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>spn</th>
<th>LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddSub</td>
<td>1</td>
</tr>
<tr>
<td>Mul</td>
<td>10</td>
</tr>
</tbody>
</table>

Evaluation: Case study „SPN“
Different queries of a Sum-Product Network
- Motivational example from the first slide

<table>
<thead>
<tr>
<th>top</th>
<th>LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddSub</td>
<td>1</td>
</tr>
<tr>
<td>Mul</td>
<td>8</td>
</tr>
<tr>
<td>Div</td>
<td>1</td>
</tr>
</tbody>
</table>

```
Mul
AddSub
```

```
AddSub
spn
```

```
Div
```

```
spn
```

```
Top-level function
```

MPE

```
Compare
spn
```

```
AddSub
Mul
```

Top-level function
Different queries of a Sum-Product Network

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

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

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

- Xilinx Vivado HLS 2018.3
Evaluation: Key Insights

- The next slides illustrate that ...
Evaluation: Key Insights

- The next slides illustrate that ...

...solving the ILP is **tractable**
The next slides illustrate that …

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

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

![Graph showing FFT, VCU108 results with utilisation of DSP slices and latency in cycles.](image)

- Vivado HLS w/o SkyCastle
- SkyCastle estimation

Report

Vivado HLS w/o SkyCastle

SC-x1

SC-x2

SC-x3

SC-x4

"SC-xk" = SkyCastle computes solution suitable for k-times replication
Results — Replication

- **SPN, VCU108**

![Graph showing replication results](image-url)

- Configuration: VHLS, SC-x1, SC-x2, SC-x3, SC-x4, SC-x5, SC-x6, SC-x7, SC-x8
- Ratio to VHLS w/o SC (theoretical) throughput, normalised
Results — Replication

- **SPN**, VCU108

<table>
<thead>
<tr>
<th>Configuration</th>
<th>VHLS disregards its cycle time target, while SC strictly obeys it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio to VHLS w/o SC</td>
<td>(theoretical) throughput, normalised</td>
</tr>
<tr>
<td>VHLS</td>
<td>SC-x1</td>
</tr>
<tr>
<td>1</td>
<td>0.75</td>
</tr>
</tbody>
</table>

VHLS disregards its cycle time target, while SC strictly obeys it.
Results — Replication

- **SPN, VCU108**

<table>
<thead>
<tr>
<th>Configuration</th>
<th>2x throughput within the same resource constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHLS</td>
<td>1</td>
</tr>
<tr>
<td>SC-x1</td>
<td>0.75</td>
</tr>
<tr>
<td>SC-x2</td>
<td>1.25</td>
</tr>
<tr>
<td>SC-x3</td>
<td>1.5</td>
</tr>
<tr>
<td>SC-x4</td>
<td>1.75</td>
</tr>
<tr>
<td>SC-x5</td>
<td>2</td>
</tr>
<tr>
<td>SC-x6</td>
<td>2.25</td>
</tr>
<tr>
<td>SC-x7</td>
<td>2.5</td>
</tr>
<tr>
<td>SC-x8</td>
<td>2.75</td>
</tr>
</tbody>
</table>

(Bar chart showing the ratio to VHLS w/o SC with normalised (theoretical) throughput.)
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