ECE382M.20: System-on-Chip (SoC) Design

Lecture 9 – Advanced HLS Techniques

Source: Z. Zhang, Cornell Univ.

Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu

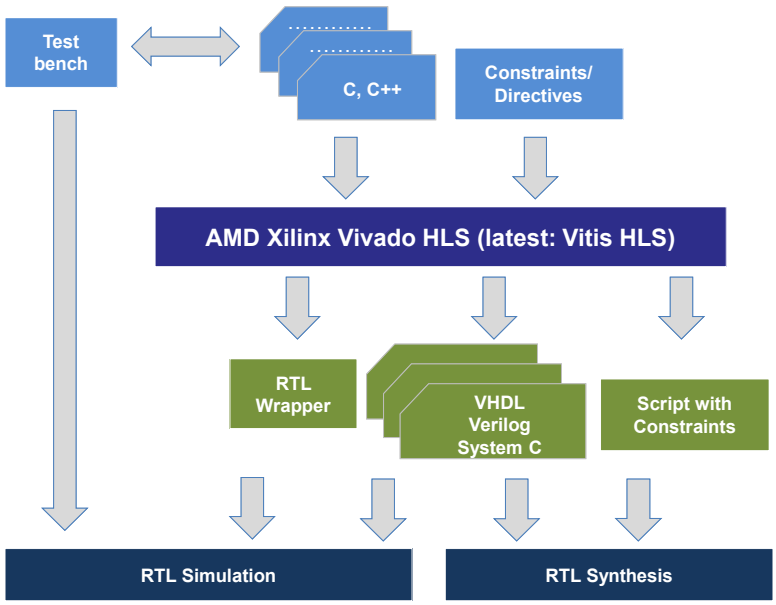


Lecture 9: Outline

- **HLS tools**
  - Vitis HLS
- **Advanced scheduling approaches**
  - SDC-based scheduling
  - Modulo scheduling
- **HLS design space exploration (DSE)**
  - Multi-objective optimization
- **HLS outlook**
  - Machine learning for HLS

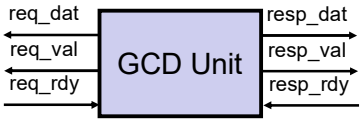
© 2026 A. Gerstlauer

Vivado/Vitis HLS



A Simple Example: RTL vs. HLS

- **A GCD unit with handshake**



HLS Code

```
void GCD (
  req_t req,
  resp_t& resp
) {
  short a = req.dat_a;
  short b = req.dat_b;
  while ( a != b ) {
    if (a > b)
      a = a - b;
    else
      b = b - a;
  }
  resp.dat = a;
}
```

Manual RTL (partial)

```
module GcdUnitRTL(
  input  wire        clk,
  input  wire [31:0] req_dat,
  output wire        req_rdy,
  input  wire        req_val,
  input  wire        reset,
  output wire [15:0] resp_dat,
  input  wire        resp_rdy,
  output wire        resp_val
);
```

Module declaration

```
always @(*) begin
  if (curr_state == STATE_IDLE)
    if (req_val)
      next_state = STATE_CALC;
  if (curr_state == STATE_CALC)
    if (!is_a_lt_b&&is_b_zero)
      next_state = STATE_DONE;
  if (curr_state == STATE_DONE)
    if (resp_val&&resp_rdy)
      next_state = STATE_IDLE;
end
```

State transition

```
always @ (*) begin
  if (current_state == STATE_IDLE) begin
    req_rdy = 1; resp_val = 0;
    a_mux_sel = A_MUX_SEL_IN;
    b_mux_sel = B_MUX_SEL_IN;
    a_reg_en = 1; b_reg_en = 1;
  end
  if (current_state == STATE_CALC) begin
    req_rdy = 0; resp_val = 0;
    do_swap = is_a_lt_b;
    do_sub = ~is_b_zero;
    a_mux_sel = do_swap ? A_MUX_SEL_B : A_MUX_SEL_SUB;
    a_reg_en = 1; b_reg_en = do_swap;
    b_mux_sel = B_MUX_SEL_A;
  end
  if (current_state == STATE_DONE) begin
    req_rdy = 0; resp_val = 1;
    a_mux_sel = A_MUX_SEL_X;
    b_mux_sel = B_MUX_SEL_X;
    a_reg_en = 0; b_reg_en = 0;
  end
end
```

Output logic

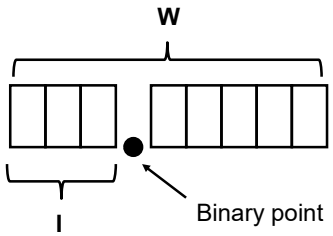
© 2026 A. Gerstlauer

C/C++ Synthesizable Subset

- **Data types**
  - **Primitive types:** `int`, `unsigned`, `float`, `double`, ...
  - **Arbitrary precision:** `ap_int`, `ap_fixed`, ...
  - **Composite types:** `array`, `struct`, `union`, ...
  - **Templated types:** `template<>`
  - **Statically determinable pointers**
- **No dynamic memory allocations:** ~~`malloc`~~, ~~`new`~~, ~~`std::vector<>`~~
- **No recursive function calls**
- **Prints in csim only**

Fixed-Point Type

- **`ap_fixed` is a templated C++ data type used for representing fixed-point numbers**
  - Signed: `ap_fixed`; Unsigned: `ap_ufixed`
  - Template parameters `ap_fixed<W, I, Q, O>`
    - W: total bitwidth
    - I: integer bitwidth
    - Q: quantization mode (optional, default is `AP_TRN`)
    - O: overflow mode (optional, default is `AP_WRAP`)



Arbitrary Precision Integer

- **C/C++ only provides a limited set of native integer types**
  - Usually: `char` (8b), `short` (16b), `int` (32b), `long` (64b), `long long` (64b)
  - Byte aligned: efficient in processors
- **Arbitrary precision integer in Vivado HLS**
  - Signed: `ap_int`; Unsigned `ap_uint`
    - Two's complement representation for signed integer
  - Templatized class `ap_int<W>` or `ap_uint<W>`
    - W is the user-specified bitwidth

```
#include <ap_int.h>
ap_int<9>    x; // 9-bit
ap_uint<24>  y; // 24-bit unsigned
ap_uint<512> z; // 512-bit unsigned
```

ECE382M.20: SoC Design, Lecture 9 © Z. Zhang 7

Typical C/C++ Constructs to RTL Mapping

C/C++ Constructs		RTL Components
Functions	→	**Modules**
Arguments	→	**Input/output ports**
Operators	→	**Functional units**
Scalars	→	**Wires or registers**
Arrays	→	**Memories**
Control flows	→	**Control logics**

ECE382M.20: SoC Design, Lecture 9 © Z. Zhang 8

Functions and Design Hierarchy

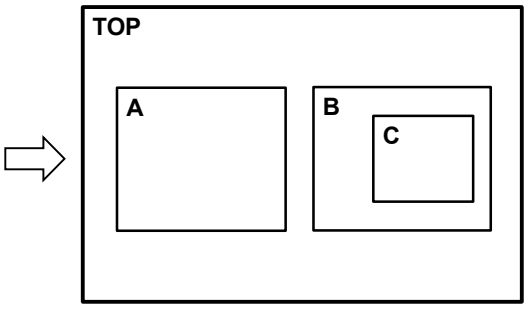
- **Each function is usually translated into an RTL module**
  - Function arguments become ports on the RTL blocks
  - Functions may be inlined to dissolve their hierarchy

Source code

```
void A(...) { ... }
void C(...) { ... }
void B(...) {
  C(...);
  ...
}

void TOP(...) {
  A(...); B(...);
}
```

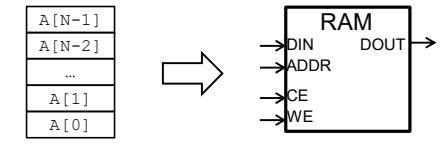
RTL module hierarchy



Arrays

- **An array is usually implemented by a memory module in RTL**
  - Reading and writing to the array correspond to accessing RAM, while constant arrays are stored in ROM
  - Typically, each memory module supports a limited number of read/write ports, typically up to 2

```
void TOP() {
  int A[N];
  for (i = 0; i < N; i++)
    A[i] = A[i] + i;
}
```



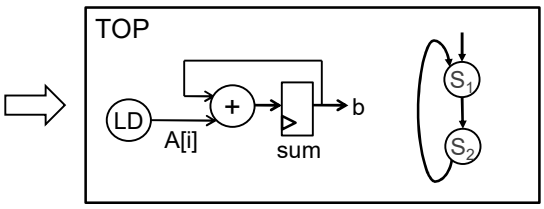
Array **A[N]** Memory Module(s)

- An array can be partitioned and implemented with multiple RAMs
  - Increase number of ports and bandwidth
  - Extreme case: completely partitioned into individual elements that map to discrete registers
- Multiples arrays can be merged and mapped to one RAM

Loops

- **By default, loops remain "rolled"**
  - Each loop iteration corresponds to a "sequence" of states (more generally, an FSM)
  - This state sequence will be repeated multiple times based on the loop trip count (or loop bound)

```
void TOP() {
  ...
  for (i = 0; i < N; i++)
    sum += A[i];
}
```



Loop Unrolling

- **Unrolling can expose more parallelism to achieve shorter latency or higher throughput**
  - (+) Decreased loop control overhead
  - (+) Increased parallelism for scheduling
  - (–) Increased operation count, which may negatively impact area, timing, and power

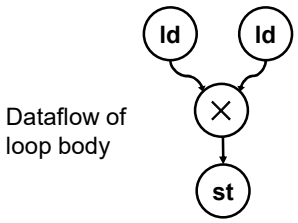
```
void TOP() {
  ...
  for (i = 0; i < N; i++){
    #pragma HLS unroll factor=4
    sum += A[i];
  }
}
```

```
void TOP() {
  ...
  for (i = 0; i < N/4; i++){
    sum += A[4*i];
    sum += A[4*i+1];
    sum += A[4*i+2];
    sum += A[4*i+3];
  }
}
```

Loop Pipelining

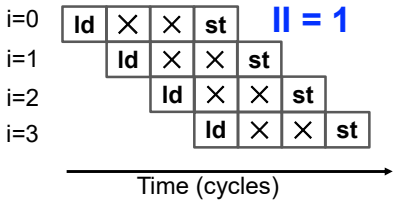
- **Pipelining is one of the most important HLS optimizations**
  - Key factor: **Initiation Interval (II)**
  - Allows a new iteration to begin processing, II cycles after the start of the previous iteration **(II=1 means the loop is fully pipelined)**

```
for (i = 0; i < N; ++i){
  #pragma HLS pipeline II=1
  p[i] = x[i] * y[i];
}
```



Dataflow of loop body

Pipelined schedule



ld – Load (memory read)
st – Store (memory write)

Here we assume multiplication (×) takes two cycles

Matrix-Vector Multiplication Example

$$\mathbf{y} = A\,\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N \\ \vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N \end{bmatrix}$$

$$y[i] = \sum_{j=0}^{N} A[i][j] \times x[j]$$

A: input matrix

x : input vector

y : output vector

```
// mv.cpp
// original, non-optimized version of matrix-
vector multiplication

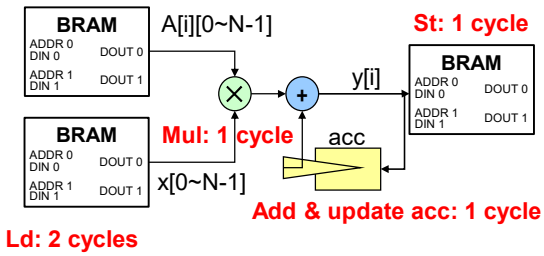
#include "mv.h" // N is defined 16 here

void MV(int A[N][N], int x[N], int y[N]) {
  int i, j;
  int acc;
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

Baseline Micro-Architecture

- Latency of `OUTER` should read 1056

```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
```



- Latency of INNER = (2+1+1) x 16 = 64
- Latency of OUTER = (64+2) x 16 = 1056

Unroll Inner Loop

- Add the unroll pragma to loop `INNER`
  -

```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
```

Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll	2.653	1030
unroll + pipeline		
unroll + partition + pipeline		

Pipeline Outer Loop

- Add the pipeline pragma to loop `OUTER`

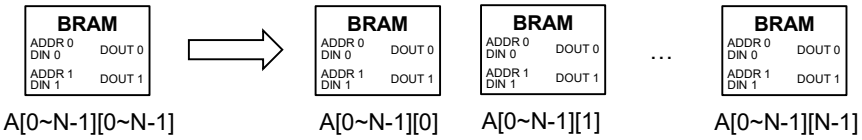
```
// N = 16
OUTER:
  for (i = 0; i < N; i++) {
    #pragma HLS pipeline II=1
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
```

The inner loop will be automatically unrolled due to pipeline, but we still keep the pragma for clarity.

- Failure
  - With an II = 1, we expect to load 16 elements of A and x every cycle, but one BRAM only has 2 ports
  - The II is relaxed to 16 / 2 = 8

The Solution: Partition Arrays

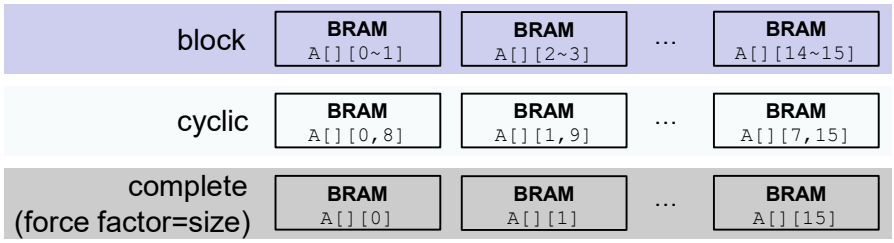
- Implements array with multiple BRAM modules




```
void MV(int A[N][N], int x[N], int y[N]) {
  #pragma HLS array_partition block variable=A dim=2 factor=8
  #pragma HLS array_partition block variable=x factor=8
  int i, j;
  int acc;
OUTER:
  for (i = 0; i < N; i++) {
    #pragma HLS pipeline II=1
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

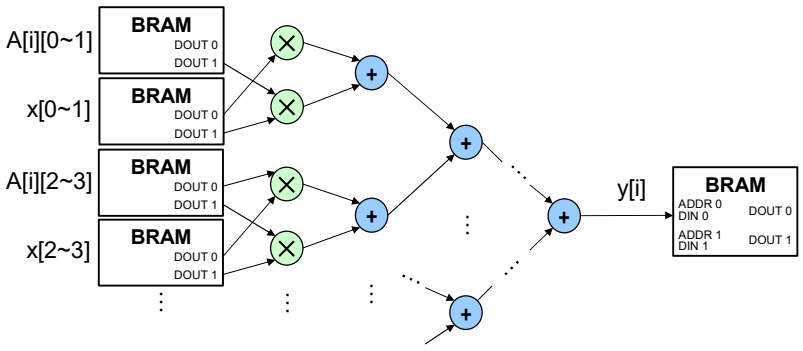
Partition Arrays

- `#pragma HLS array_partition block variable=A dim=2 factor=8`
  - **dim**: dimension to be partitioned, select according to the unrolled loop (dim=0 partitions all dimensions)
  - **factor**: number of small arrays after partition
  - **block**: mode of partition (other modes are cyclic and complete)



- In our case, both block and cyclic will achieve II = 1
- When all dimensions are completely partitioned, array will be implemented with registers

Microarchitecture of the Optimized Design

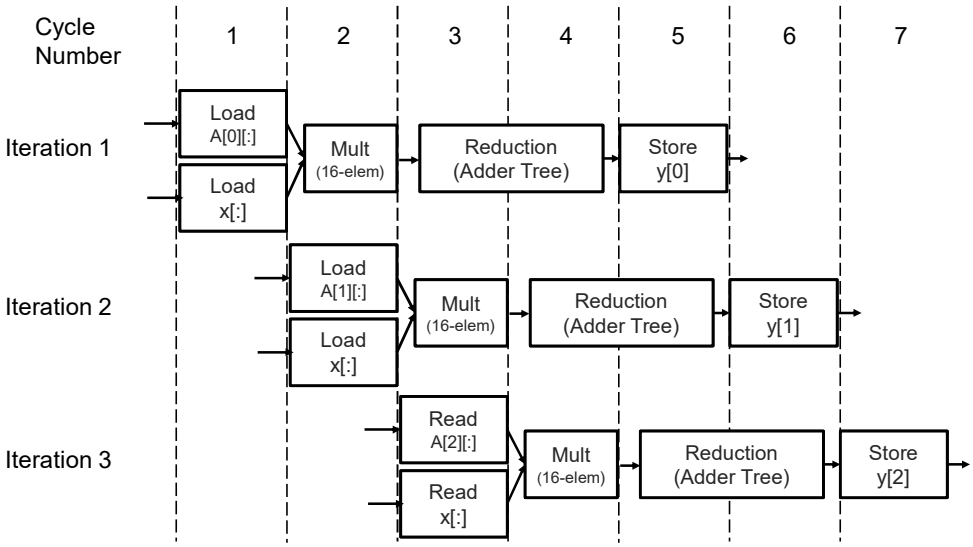


Design	Throughput (Ops/Cycle)	Area (DSP+FF+LUT)
baseline	0.484	357
unroll	2.653	1030
unroll + pipeline	3.850	1047
unroll + partition + pipeline	23.27	2862

48x speedup

8x area

Pipeline Schedule of the Optimized Design

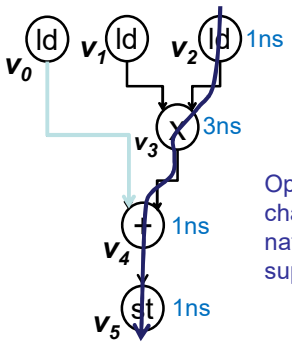


Lecture 9: Outline

- ✓ Vitis HLS
  - ✓ Vitis overview & tutorial
- **Advanced scheduling approaches**
  - SDC-based scheduling
  - Modulo scheduling
- **HLS design space exploration (DSE)**
  - Multi-objective optimization
- **HLS outlook**
  - Machine learning for HLS

SDC-Based Scheduling

- **SDC = System of difference constraints**



Operation chaining is naturally supported

s_i : schedule variable for operation i

- Dependence constraints
  - $<v_0, v_4> : s_0 - s_4 \le 0$
  - $<v_1, v_3> : s_1 - s_3 \le 0$
  - $<v_2, v_3> : s_2 - s_3 \le 0$
  - $<v_3, v_4> : s_3 - s_4 \le 0$
  - $<v_4, v_5> : s_4 - s_5 \le 0$
- Cycle time constraints
  - $v_1 \rightarrow v_5 : s_1 - s_5 \le -1$
  - $v_2 \rightarrow v_5 : s_2 - s_5 \le -1$

Timing constraints

- Target cycle time: 5ns
- Delay estimates
  - Mul (x): 3ns
  - Add (+): 1ns
  - Load/Store (ld/st): 1ns

To meet the cycle time, v_2 and v_5 should have a minimum separation of one cycle

[J. Cong & Z. Zhang, DAC, 2006] [Z. Zhang & B. Liu, ICCAD, 2013]

Difference Constraints

- **A difference constraint is a formula in the form of $x - y \le b$ or $x - y < b$ for numeric variables x and y, and constant b**
- **With scheduling variables, we use integer difference constraints to model a variety of scheduling constraints**
  - x and y must have integral values
    - Thus b only needs to be an integer => form $x - y < b$ is redundant

SDC Constraint Matrix

- **Constraint matrix of SDC(*X*, *C*) is a totally unimodular matrix (TUM)**
  - Every nonsingular square submatrix has a determinant of -1/+1

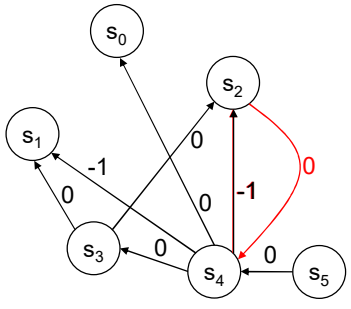
$$\begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & \\ 0 & 0 & 1 & -1 & 0 & 0 & \\ 0 & 0 & 0 & 1 & -1 & -1 & \\ 0 & 0 & 0 & 0 & 1 & -1 & \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & & & \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ -1 \end{pmatrix}$$

A x b

  - Theorem (Hoffman & Kruskal, 1956): If *A* is totally unimodular and *b* is a vector of integers, every extreme point of polyhedron $\{x : Ax \leq b\}$ is integral.
  - ➢ **Solving linear programming (LP) relaxation leads to integral solutions**

SDC Constraint Graph

- **Difference constraints can be conveniently represented using constraint graph**
  - Each vertex represents a variable, and each weighted edge corresponds to a different constraint
  - Detect infeasibility by the presence of negative cycle (by solving single-source shortest path)

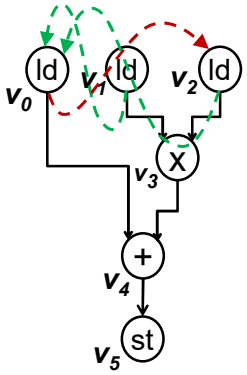


$s_2 - s_4 \leq -1$
$s_4 - s_2 \leq 0$
$0 \leq -1$

$s_0 - s_4 \leq 0$
$s_1 - s_3 \leq 0$
$s_2 - s_3 \leq 0$
$s_3 - s_4 \leq 0$
$s_4 - s_5 \leq 0$
$s_2 - s_4 \leq -1$
$s_1 - s_4 \leq -1$
$s_4 - s_2 \leq 0$

Handling Resource Constraints

- **Resource constraints cannot be represented exactly in integer difference form**



- Resource constraint
  - Two read ports

- Resource constraints
  - ➔ Heuristic partial orderings

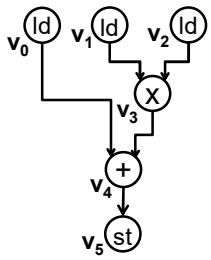
$v_0 \rightarrow v_2 : s_0 - s_2 \leq -1$ 3 cycle latency

OR

$v_1 \rightarrow v_0 : s_1 - s_0 \leq -1$
$v_2 \rightarrow v_0 : s_2 - s_0 \leq -1$ 2 cycle latency

Linear Objectives

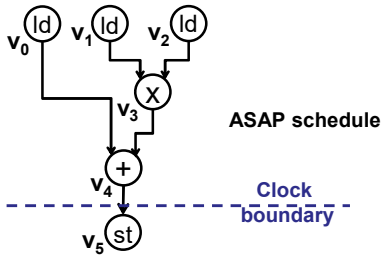
- **ASAP: min $\sum_{i \in V} s_i$**
- **ALAP: max $\sum_{i \in V} s_i$**
- **Minimum latency: min $\max_{i \in V} \{s_i\}$**
- **Minimum average case latency (control-intensive design)**
- **Many other …**

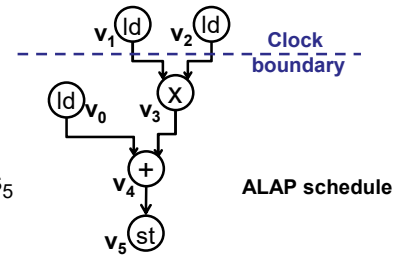


- Target cycle time: 5ns
- Delay estimates
  - Mul (x): 3ns
  - Add (+): 1ns
  - Load/Store (ld/st): 1ns

min s_0 + … + s_5

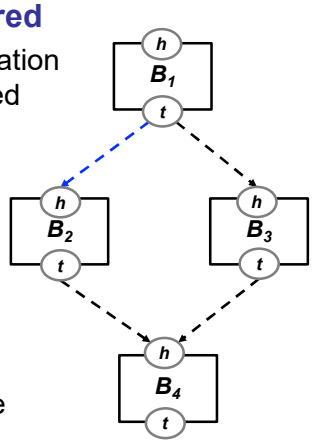
max s_0 + … + s_5





Control Flow Graphs

- **Control dependencies can also be honored**
  - If bb_2 is control dependent on bb_1, the operation nodes of bb_2 are not allowed to be scheduled before those of bb_1
  - Polarize each basic block bb_i with two scheduling variables (head and tail)
    - $\forall v \in bb_i$, $s_h(bb_i) - s_h(v) \leq 0$
    - $\forall v \in bb_i$, $s_t(v) - s_t(bb_i) \leq 0$
  - If $e_c(bb_i, bb_j) \in E_c$ and e_c is not a back edge
    - $s_t(bb_i) - s_h(bb_j) \leq 0$



$s_t(B_1) - s_h(B_2) \leq 0$

ECE382M.20: SoC Design, Lecture 9 © Z. Zhang 29

Lecture 9: Outline

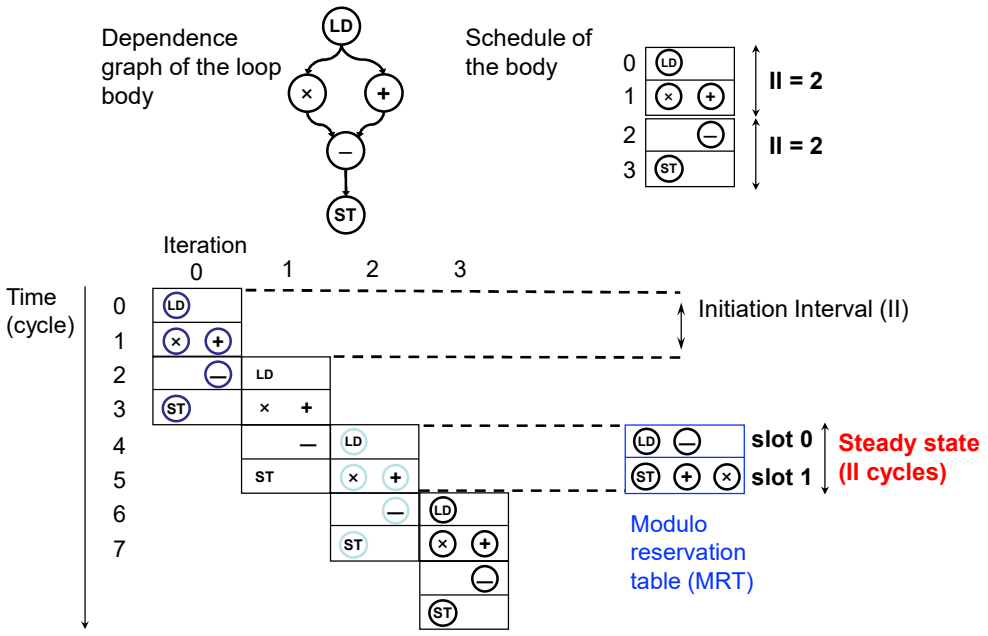
- ✓ Vitis HLS
  - ✓ Vitis overview & tutorial
- **Advanced scheduling approaches**
  - ✓ SDC-based scheduling
  - Modulo scheduling
- HLS design space exploration (DSE)
  - Multi-objective optimization
- HLS outlook
  - Machine learning for HLS

ECE382M.20: SoC Design, Lecture 9 © A. Gerstlauer 30

Modulo Scheduling

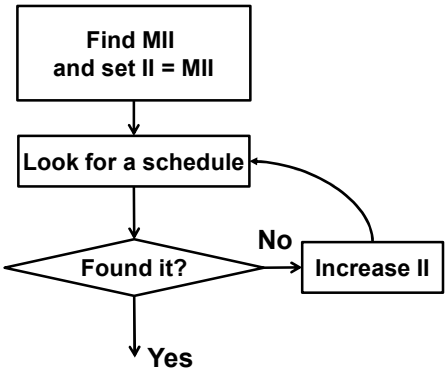
- **A regular form of loop (or function) pipelining technique**
  - Also applies to software pipelining in compiler optimization
  - Loop iterations use the same schedule, which are initiated at a constant rate
  - Typical objective: minimize II under resource constraints
    - NP-hard in general
    - Optimal polynomial time solution exists without recurrences or resource constraints
- **Advantages of modulo scheduling**
  - Cost efficient: No code or hardware replication
  - Easy to analyze
    - Steady state determines performance & resource

Modulo Scheduling Example



Algorithm for Modulo Scheduling

- **Heuristic algorithm**
  - Find a lower bound on II
  - Look for a schedule with the given II
  - If a feasible schedule not found, increase II and try again

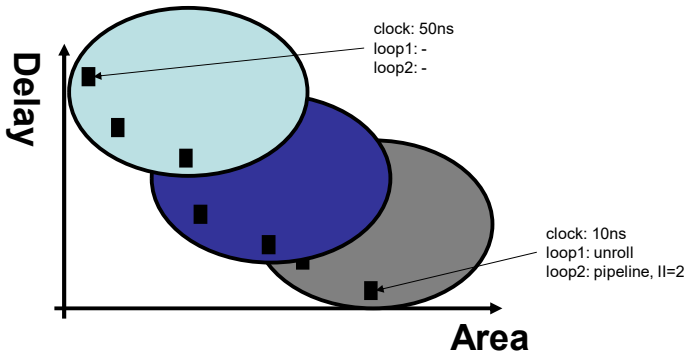


Lecture 9: Outline

- ✓ **Vitis HLS**
  - ✓ Vitis overview & tutorial
- ✓ **Advanced scheduling approaches**
  - ✓ SDC-based scheduling
  - ✓ Modulo scheduling
- **HLS design space exploration (DSE)**
  - Multi-objective optimization
- **HLS outlook**
  - Machine learning for HLS

Multi-Objective Design Space

- **Design space enabled by manual synthesis directives**
  - Local pragma settings, global constraints & options
  - Automatic exploration not handled by traditional algorithms

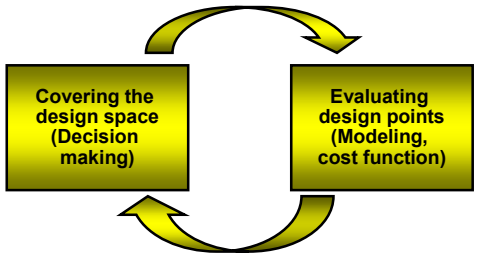


- **Multi-objective optimization (MOO)**
  - Pareto optimality, find Pareto front

Source: J. Abraham

Design Space Exploration (DSE)

- **Design space exploration**



- **Automatic DSE (Auto-DSE)**
  - Automatic pragma, option & configuration setting
  - Synthesis area & delay result estimation

Source: C. Haubelt

Automatic DSE

- **Brute-force methods**
  - Exhaustive search
  - Random search
- **Meta-heuristics**
  - Simulated annealing
  - Genetic algorithms
  - …
- **Dedicated heuristics**
  - Divide & conquer
  - Clustering
  - …

Source: B. Carrion-Schaefer

Synthesis Result Estimation

- **Synthesis-based**
  - Run design through HLS tool
  - Use estimated HLS result metrics (synthesis report)
- **Model-based**
  - Sample the design space
  - Learn a model that predicts synthesis results
  - Potentially even predict (parts of) the design space

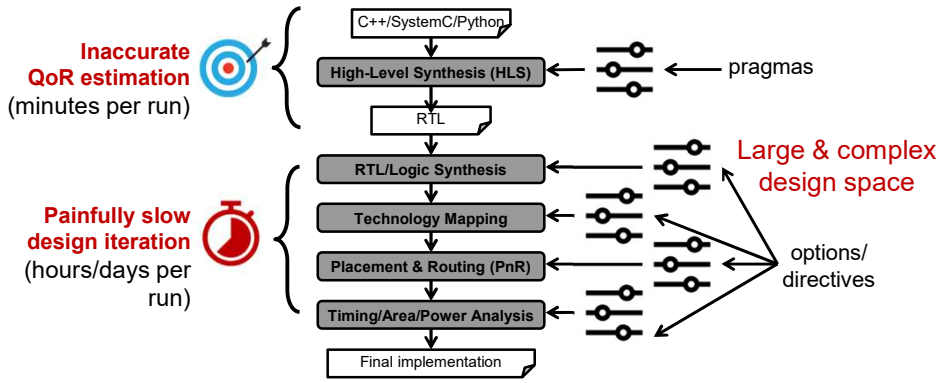
Source: B. Carrion-Schaefer

Lecture 9: Outline

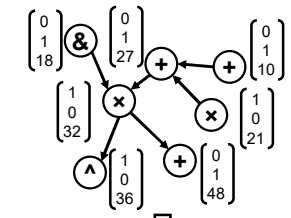
- ✓ Advanced scheduling approaches
  - ✓ SDC-based scheduling
  - ✓ Modulo scheduling
- ✓ HLS design space exploration (DSE)
  - ✓ Multi-objective optimization
- **HLS outlook**
  - Machine learning for HLS

Hurdles in Agile Hardware Design

- **Tension between speed and fidelity w/ HLS-based design**



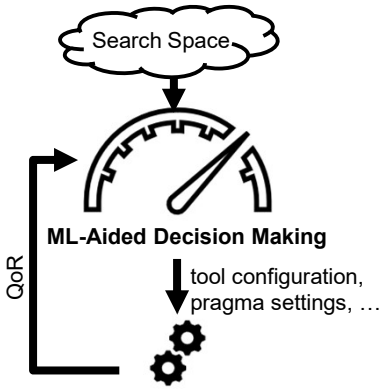
Machine Learning (ML) in HLS



ML-Based Function Approximation

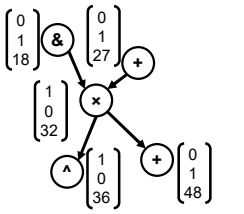
Predicted area, delay, power

① **Faster & More Accurate QoR Inference**



② **More Intelligent Design Space Exploration (DSE)**

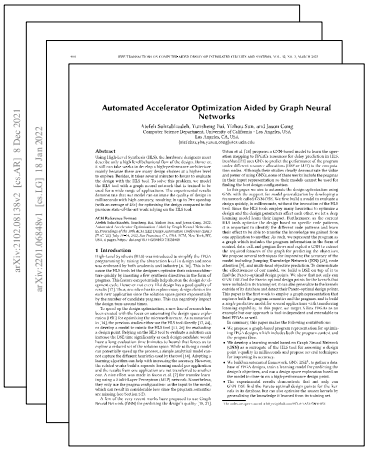
ML for HLS QoR Inference



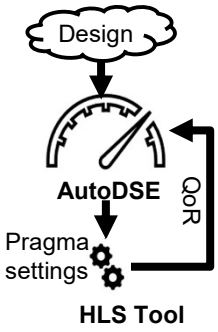
ML-Based Function Approximation

Predicted QoR

ML Model	Estimation Targets	Reference
GNN	Post-HLS Resource & Latency	GNN4HLS [Ferretti et al. TODAES'22]
GNN	Post-HLS Resource & Latency	GNN-DSE [Sohrabizadeh et al. DAC'22]
GNN	Post-PnR Resource & Timing	IronMan [Wu et al. GLSVLSI'21]
GNN	Post-PnR Resource & Timing	– [Wu et al. DAC'22]
GNN	Post-PnR Resource & Timing	IronMan-Pro [Wu et al. TCAD'23]
GNN	Post-PnR Power	PowerGear [Lin et al. DATE'22]
CNN	Post-PnR Power	HL-Pow [Lin et al. TCAD'23]



ML for HLS DSE



ML Model	Model Inputs	Reference
Gaussian Processes	HLS Pragmas	Prospector [Mehrabi et al. DATE'22]
Gaussian Processes	HLS Pragmas	– [Sun et al. TODAES'22]
CNN	HLS Pragmas	HL-Pow [Lin et al. TCAD'23]
GNN	HLS Pragmas + CDFGs	GNN4HLS [Ferretti et al. TODAES'22]
GNN	HLS Pragmas + CDFGs	GNN-DSE [Sohrabizadeh et al. DAC'22]
GNN + RL	HLS Pragmas + DFGs	IronMan [Wu et al. GLSVLSI'21]
GNN + RL	HLS Pragmas + DFGs	IronMan-Pro [Wu et al. TCAD'23]
LLM + GNN	HLS Source + Pragmas + CDFG	ProgSG [Bai et al. ArXiv'23]



HLS Estimation Accuracy

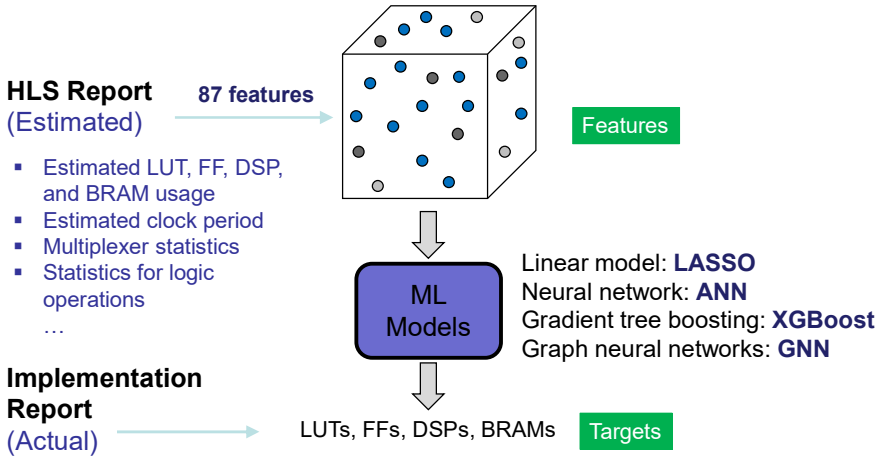
- **65 individual designs**
  - CHStone [Hara et.al, JIP'08]
  - Machsuite [Reagen et.al, IISWC'14]
  - S2CBench [Schafer & Mahapatra, ESL'14]
  - Rosetta [Zhou et.al, FPGA'18]
- **1300 data samples**
  - Clock periods: 1, 2, 3, 5, 10ns
  - Zynq, Artix, Kintex, and Virtex devices

Commercial HLS
(Post-HLS estimates)
- Resource usage
- Timing
- Performance
- HDL details, etc.

Implementation
(Post-PnR results)
- Resource usage
- Timing, etc.

Estimation Error
LUT: mean 2~4x
FF: mean 2~3x

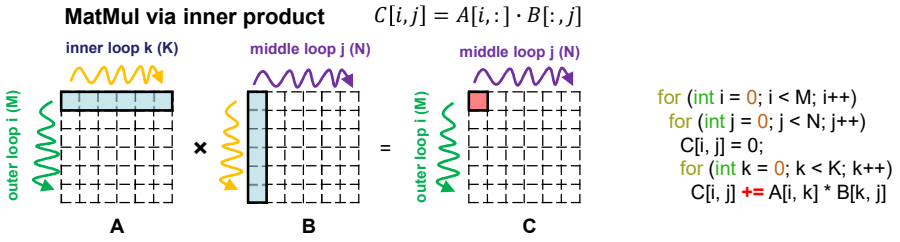
ML-Based Resource Estimation in HLS



[1] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F.Y. Young, and Z. Zhang, Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning, FCCM, May 2018. (Best Paper Award, Short Paper Category)

Challenges to Effective DSE

- **Creating a high-performance HLS design require nontrivial *manual* source-level transformations that alters the loop structures and data layout**
- **Example: Inefficient matrix multiplication using inner product**
  - Slow pipeline due to carried dependency from floating-point accumulation

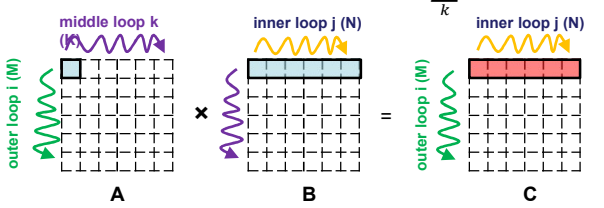



```
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    C[i, j] = 0;
    for (int k = 0; k < K; k++)
      C[i, j] += A[i, k] * B[k, j]
```

Challenges to Effective DSE

- **A faster approach using row-wise product**
  - Different loop orders & additional on-chip storage
  - ➢ Existing HLS AutoDSE methods cannot achieve this solution through permutations of the pragma settings

MatMul via row-wise product $C[i,:] = \sum_k A[i,k] \cdot B[k,:]$




```
for (int i = 0; i < M; i++) {
  float C_vec[N];
  for (int j = 0; j < N; j++)
    C_vec[j] = 0.0;

  for (int k = 0; k < K; k++)
    for (int j = 0; j < N; j++)
      #pragma pipeline II=1
      C_vec[j] += A[i, k] * B[k, j];

  for (int j = 0; j < N; j++)
    C[i, j] = C_vec[j];
}
```

New opportunities in co-designing HLS programming models and ML-based DSE to enable exploration of high-level (initially algebraic) transformations

Lecture 9: Summary

- **HLS is (again) an active research area**
  - Golden age for hardware specialization
    - End of traditional semiconductor scaling
  - Application demands
    - E.g. machine learning
  - "Programmable" hardware substrates
    - FPGAs
- **Advanced HLS techniques**
  - Scheduling generalizations and optimality
  - Automatic design space exploration (Auto-DSE)
  - Machine learning for HLS