More Pipelining
Announcements

- Lab 3 due Friday at 11:59pm

- Ritchie Zhao (PhD student) will give a tutorial on neural networks this Thursday

- Instructor OH on Thursday is rescheduled to Friday 10/5, 11am-noon

- First student-led seminar is on Thursday 10/16
  - 4 students per group (find you partners now!)
  - 18 mins per presentation
  - Paper bidding instructions will be posted on Piazza soon
Review: Compatibility/Conflict Graph

<table>
<thead>
<tr>
<th>Meeting</th>
<th>Schedule (am)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>9:00~11:00</td>
</tr>
<tr>
<td>B</td>
<td>9:30~10:00</td>
</tr>
<tr>
<td>C</td>
<td>10:00~11:00</td>
</tr>
<tr>
<td>D</td>
<td>11:00~11:30</td>
</tr>
</tbody>
</table>

Interval graph

Conflict graph
chromatic number = 2
max clique size = 2

Compatibility graph
max clique size = 3
chromatic number = 3
Outline

▸ Modulo scheduling concepts

▸ Extending SDC formulation for pipelining

▸ Case studies on HLS pipelining
Data dependences of a loop often represented by a dependence graph

- Forward edges: **Intra-iteration** (loop-independent) dependences
- Back edges: **Inter-iteration** (loop-carried) dependences
- Edges are annotated with distance values: number of iterations separating the two dependent operations involved

Recurrence manifests itself as a **circuit** in the dependence graph
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

- Advantages of modulo scheduling
  - Easy to analyze: Steady state determines performance & resource
  - Cost efficient: No code or hardware replication

- Optimization objective:
  (1) minimize II under resource constraints; or
  (2) minimize resource usage under II constraint
  - NP-hard in general
  - Optimal polynomial time solution exists without recurrences or resource constraints
Modulo Scheduling Example

Dependence graph:

Schedule:

Initiation Interval (II)

Steady state determines both performance resource usage
Common scheme of heuristic algorithms
- Find a lower bound on II, called $M_{II} = \max \{ \text{Res}_{II}, \text{Rec}_{II} \}$
- Look for a schedule with the given II
- If a feasible schedule not found, increase II and try again
Calculating Lower Bound of Initiation Interval

- Minimum possible II (MII)
  - \( \text{MII} = \max(\text{ResMII}, \text{RecMII}) \)
  - A lower bound, not necessary achievable

- Resource constrained MII (ResMII)
  - \( \text{ResMII} = \max_i \left[ \frac{\text{OPs}(r_i)}{\text{Limit}(r_i)} \right] \)
    - \( \text{OPs}(r) \): number of operations that use resource of type \( r \)
    - \( \text{Limit}(r) \): number of available resources of type \( r \)

- Recurrence constrained MII (RecMII)
  - \( \text{RecMII} = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right] \)
    - \( \text{Latency}(c_i) \): total latency in dependence circuit \( c_i \)
    - \( \text{Distance}(c_i) \): total distance in dependence circuit \( c_i \)
Compute ResMII: Max among all types of resources

- \( \text{ResMII} = \max_i \left\lfloor \frac{\text{OPs}(r_i)}{\text{Limit}(r_i)} \right\rfloor \)
Compute Recurrence Minimum II ($\text{RecMII}$):
- Max among all circuits of:
  \[ \text{RecMII} = \max_i \left[ \frac{\text{Latency}(c_i)}{\text{Distance}(c_i)} \right] \]
- $\text{Latency}(c)$ : sum of operation latencies along circuit $c$
- $\text{Distance}(c)$ : sum of dependence distances along circuit $c$
SDC-Based Modulo Scheduling

- The SDC formulation can be extended to support modulo scheduling
  - Unifies intra-iteration and inter-iteration scheduling constraints in a single SDC
  - Iterative algorithm with efficient incremental SDC update
Modeling Loop-Carried Dependence with SDC

- The dependence between two operations from different iterations is termed inter-iteration (loop-carried) dependence
  - Loop-carried dependence $u \rightarrow v$ with $\text{Dist}(u, v) = K$
  - $s_u + \text{Lat}_u \leq s_v + K^* \text{II}$

```
for (i = 0; i < N-2; i++)
{
    B[i] = A[i] * C[i];
    A[i+2] = B[i] + C[i];
}
```

Dist($v_5$, $v_1$) = 2
Modeling Loop-Carried Dependence with SDC

- The dependence between two operations from different iterations is termed inter-iteration (loop-carried) dependence
  - Loop-carried dependence $u \rightarrow v$ with $\text{Dist}(u, v) = K$
    
    $$s_u + \text{Lat}_u \leq s_v + K \cdot 2$$

for ($i = 0; i < N-2; i++$)
{

  $B[i] = A[i] \times C[i]$;

}
Loop pipelining may result in operation chaining across iterations

- Given a target clock period $T$, the maximum combinational delay within a clock cycle must not exceed $T$
  
  - $T < \text{Delay}(v_i, v_j) = \text{the max combinational delay}$
  
  $s_u - s_v \leq \text{Dist}(u, v) \ast \text{II} - \text{Lat}_u - 1$

Cycle time: 6ns
Add: 2ns
Mult: 5ns
Inter-Iteration Cycle Time Constraint with SDC

- Loop pipelining may result in operation chaining across iterations
  - Given a target clock period T, the maximum combinational delay within a clock cycle must not exceed T
    - \( T < \text{Delay}(v_i, v_j) = \text{the max combinational delay} \)
    - \( s_u - s_v \leq \text{Dist}(u, v) * \text{Lat}_u - 1 \)

\[
S = 0; \\
\text{for } (i = 0; i < 10; i++) \\
\{ \\
\quad S = S \times A[i]; \\
\quad S = S + M; \\
\}
\]

- Cycle time: 6ns
- Add: 2ns
- Mult: 5ns
Case Study: Prefix Sum

Prefix sum computes a cumulative sum of a sequence of numbers
  - commonly used in many applications such as radix sort, histogram, etc.

```c
void prefixsum ( int in[N], int out[N] )
out[0] = in[0];
for ( int i = 1; i < N; i++ ) {
    #pragma HLS pipeline II=?
    out[i] = out[i-1] + in[i];
}
```
out[0] = in[0];
out[1] = in[0] + in[1];
...
```
Prefix Sum: RecMII

- Loop-carried dependence exists between to reads on ‘out’
- Assume chaining is not possible on memory reads (ld) and writes (st) due to target cycle time
  - RecMII = 3

in[i]

ld_2

ld_1

+ — Store

out[i]

out[i-1]

Assume chaining is not possible on memory reads (i.e., ld) and writes (i.e., st) due to cycle time constraint

out[0] = in[0];

for ( int i = 1; i < N; i++ )
out[i] = out[i-1]+ in[i];

<table>
<thead>
<tr>
<th>i = 0</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld_1</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld_2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i = 1</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld_1</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld_2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

id – Load
st – Store
Prefix Sum: Code Optimization

- Introduce an intermediate variable ‘tmp’ to hold the running sum from the previous ‘in’ values
- Shorter dependence circuit leads to RecMII = 1

```
int tmp = in[0];
for ( int i = 1; i < N; i++ ) {
    tmp += in[i];
    out[i] = tmp;
}
```

<table>
<thead>
<tr>
<th></th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>ld</td>
<td>+</td>
<td>st</td>
<td></td>
</tr>
<tr>
<td>$i = 1$</td>
<td>![comments]</td>
<td>ld</td>
<td>+</td>
<td>st</td>
</tr>
</tbody>
</table>

**Id** – Load
**st** – Store
Case Study: Convolution for Image Processing

- A common computation of image/video processing is performed over overlapping stencils, termed as convolution.

\[
(Img \otimes f)[n + \frac{k-1}{2}, m + \frac{k-1}{2}] = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} Img[n+i][m+j] \cdot f[i,j]
\]

![Input image frame](image)

![3x3 convolution](image)

![Output image frame](image)
Achieving High Throughput with Pipelining

```c
for (r = 1; r < R; r++)
    for (c = 1; c < C; c++) {
        #pragma HLS pipeline II=
        for (i = 0; i < 3; i++)
            for (j = 0; j < 3; j++)
                out[r][c] += img[r+i-1][c+j-1] * f[i][j];
    }
```

- Inner loops (i & j) are automatically unrolled
- With a 3x3 convolution kernel, 9 pixels are required for calculating the value of one output pixel
- If the entire input image is stored in an on-chip buffer with two read ports
  - ResMII = ?
  - What about RecMII?
Achieving II=1 for 3x3 Convolution

Push three pixels into shift register window: one new pixel plus two pixels from line buffer

Pixels in **line buffer**
(2 lines stored)

New pixel fetched from input stream or frame buffer in off-chip memory

Output pixel produced by one convolution operation
Resulting Specialized Memory Hierarchy

- Memory architecture customized for convolution

Input pixel stream → Flip-Flops → Convolve → Output pixel stream

- On-chip SRAMs
- Off-chip DDR
- Line buffers
- Frame buffers

Processing window
HLS Code Snippet

```c
LineBuffer<2,C,pixel_t> linebuf;
Window<3,3,pixel_t> window;
for (int r = 1; r < R+1; r++) {
    for (int c = 1; c < C+1; c++) {
        #pragma HLS pipeline II=1
        pixel_t new_pixel = img[r][c];
        // Update shift window
        window.shift_left();
        if (r < R && c < C) {
            for (int i = 0; i < 2; i++)
                window.insert(buf[i][c]);
        } else { // zero padding
            for (int i = 0; i < 2; i++)
                window.insert(0);
        }
        window.insert(new_pixel);
        // Update line buffer
        linebuf.shift_up(c);
        if (r < R && c < C)
            linebuf[1].insert(c, new_pixel);
        else // Zero padding
            linebuf[1].insert(c, 0);
        // Perform 3x3 convolution
        out[r-1][c-1] = convolve(window, weights);
    }
}
```
Summary

- Pipelining is one of the most commonly used techniques in HLS to boost performance of the synthesized hardware

- Recurrences and resource restrictions limit the pipeline throughput

- Modulo scheduling
  - A regular form of software pipeline technique
    - Also applies to loop pipelining for hardware synthesis
  - NP-hard problem in general
Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Ryan Kastner (UCSD)
  - Prof. Scott Mahlke (UMich)
  - Dr. Stephen Neuendorffer (Xilinx)