18-643 Lecture 8: C-function-to-IP HLS (Vitis HLS IP-Flow)

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today: learn how to tell Vitis what you want and understand what Vitis tells you back

• Notices
  – Handout #5: lab 2, due noon, 10/9
  – Project status report due each Friday

• Readings (see lecture schedule online)
  – Ch 15, The Zynq Book (skim Ch 14)
  – for lab2, C. Zhang, et al., ISFPGA, 2015
Vitis C-to-RTL HLS

- Function-to-IP, not Program-to-HW
  - never mind all of C (what’s main()? what malloc?)
  - never mind all usages of allowed subset (all loops okay, but static ones actually work well)
  - what else beyond C might a HW designer need (types, interface, structural hints)

  You can use it as a better RTL

- Designer still in charge (garbage in, garbage out)
  - specify functionality as algorithm (in C)
  - specify structure as pragmas (beyond C)
  - set optimization constraints (beyond C)

  Offload bit- and cycle-level design/opt. to tools
What does Vitis HLS see?

```c
int fibi(int n) {
    int last=1; int lastlast=0; int temp;

    if (n==0) return 0;
    if (n==1) return 1;

    for(;n>1;n--) {
        temp=last+lastlast;
        lastlast=last;
        last=temp;
    }

    return temp;
}
```
Function to IP Block

Later—what if you want 2 outputs?

```c
int fibi(int n) {
    . . . .
    return ...;
}
```
Function Invocation: Latency vs Throughput

- start
- ready
- done

minimum initiation interval

latency
Other Block Control Options

- **ap_ctrl_chain**
  - separate input producer and output consumer
  - **ap_continue**: driven by the consumer to backpressure the block and producer
  - IF a block reaches “done” AND **ap_continue** is deasserted, the block will hold **ap_done** and keep output valid until **ap_continue** is asserted

- **AXI compatible memory-mapped control**
  - software on ARM interacts with the block using fxn-call-like interfaces (input, output, start, etc.)
  - IP-specific .h and routines generated automatically
Scalar I/O Port Timing

• By default (ap_none)
  – input ports should be stable between ap_start and ap_ready
  – output port is valid when ap_done

• 3 asynchronous handshake options on input
  – ap_vld only: consumes only if input valid
  – ap_ack only: signals back when input consumed
  – ap_hs: ap_vld + ap_ack

• HLS’s job to follow protocol
Pass-by-Reference Arguments

void fibi(int *n, int *fib) {
    int last=1; int lastlast=0; int temp;
    int nn=*n;

    if (nn==0) { *fib=0; *n=0; return; }
    if (nn==1) { *fib=1; *n=0; return; }
    for(;nn>1;nn--) {
        temp=last+lastlast;
        lastlast=last;
        last=temp;
    }

    *fib=last; *n=lastlast;
}
They are not really “pointers”
• do not evaluate *(fib+1) or fib
• except to pretend to be a fifo

```c
void fibi(int *n, int *fib) {
    // ... 
    *n and *fib assigned to;
    *n in RHS before assigned;
    *fib in RHS after assigned;
    // ... 
}
```
## All I/O Options

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Interface Mode</th>
<th>Scalar</th>
<th>Array</th>
<th>Pointer or Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Input</td>
<td>Return</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I</td>
</tr>
<tr>
<td>ap_ctrl_none</td>
<td>ap_ctrl_none</td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>ap_ctrl_hs</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>ap_ctrl_chain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>axis</td>
<td>axis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_axi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m_axi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_none</td>
<td>ap_none</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_stable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_vld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_ovld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_hs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_memory</td>
<td>ap_memory</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bram</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_fifo</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ap_bus</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Supported
- D = Default Interface
- Not Supported

---

Fig 1-49, Vivado Design Suite User Guide: High-Level Synthesis

pointer I/O can output before fxn ends and multiple times
Array Arguments

```c
#define N (1<<10)
void D2XPY (double Y[N], double X[N]) {
    for (i=0; i<N; i++) {
        Y[i] = 2*X[i] + Y[i];
    }
}
```

*could ask to use separate read and write ports*
Array Arg Options

- By default, array args become BRAM ports
  - array must be fixed size
  - can use 2 ports for bandwidth or split read/write
- If array arg is accessed always consecutively AND only either read or written
  - can become ap_fifo port
  - i.e., no address wires, just push or pop
- Array args can also become AXI or a generic bus master ports

Scheduler handles port sharing and dynamic delays
Time to Look Inside

n → fibi

ap_clk → ap_ready
ap_rst → ap_done
ap_start → ap_idle
void mmm(char A[N][N], char B[N][N], short C[N][N]) {

Row: for(int i=0; i<N; i++) {
    Col: for(int j=0; j<N; j++) {
        C[i][j]=0;
        Product: for(int k=0; k<N; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
}
If you want to be literal

\[
C[i][j] += A[i][k] \times B[k][j]
\]

for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++) {
            \text{write-enable}
        }
    }
}

RAW hazard?

```
for(int i=0; i<N; i++) {
    for(int j=0; j<N; j++) {
        for(int k=0; k<N; k++) {
            C[i][j] += A[i][k] \times B[k][j]
        }
    }
}
```

Diagram:

- FSM
- BRAM
- C[i][j] +
- A[i][k] \times B[k][j]
If you want to be literal for 5x5 matrices...
If you want to be literal (continued)

```c
for(int k=0; k<N; k++) {
    C[i][j] += A[i][k]*B[k][j];
}
```
Let's Try Pipelining

for 5x5 matrices
StructuralPragma: Pipelining

• Find minimum “iteration interval (II)” schedule
  – II >= num stages a resource instance is used
  – II >= RAW hazard distance
• E.g., to pipeline $C[i][j] += A[i][k]*B[k][j]$;

RAW hazard, II>=2
(w. write forward)

structural conflict, II>=2
(II>=1 if 2-port)
What Vitis HLS tells you . . .

- **matrix_mult**: slack violation, latency (cycles) 376, latency (ns) 3.76E3, iteration latency 377, interval 25, trip count 2, pipelined no.
- **Row_Col**: slack violation, latency (cycles) 375, latency (ns) 3.75E3, iteration latency 15, interval 25, trip count 2, pipelined no.
- **Product**: slack violation, latency (cycles) 12, latency (ns) 120,000, iteration latency 5, interval 2, trip count 2, pipelined yes.

The II Violation in module 'matrix_mult' (loop 'Product'): Unable to enforce a carried dependence constrair operation ('prod_addr_write_In16',../matrix_mult.cpp:16) of variable 'add_In16',../matrix_mult.cpp:16 on array 'prod'.
Pipelining result: Target II = 1, Final II = 2, Depth = 5, loop 'Product'.

**Operation\Control Step**
- **Product**
  - prod_load(read)
  - reuse_select(select)
  - add_In16(+)
  - prod_addr_write_In16(write)
Removing “volatile”

for 5x5 matrices
Inferring Accumulation Register

```c
int temp; // carries dependency
for(int cnt=0; cnt<125; cnt++) {
    int i=cnt/25, j=(cnt/5)%5, k=cnt%5;
    if (k==0) temp=C[i][j];
    temp += A[i][k]*B[k][j];
    C[i][j]=temp;
}
```

safe if no one else changing C[][]
Letting Vitis HLS just do its own thing

```cpp
#include "matrix_mult.h"

void matrix_mult(
  mat_a a[IN_A_ROWS][IN_A_COLS],
  mat_b b[IN_B_ROWS][IN_B_COLS],
  mat_prod prod[IN_A_ROWS][IN_B_COLS])
{
  // Iterate over the rows of the A matrix
  Row: for(int i = 0; i < IN_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < IN_B_COLS; j++) {
      prod[i][j] = 0;
      // Do the inner product of a row of A and col of B
      Product: for(int k = 0; k < IN_B_ROWS; k++) {
        prod[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

for 5x5 matrices

<table>
<thead>
<tr>
<th>Modules &amp; Loops</th>
<th>Issue Type</th>
<th>Slack</th>
<th>Latency(cycles)</th>
<th>Latency(ns)</th>
<th>Iteration Latency</th>
<th>Interval</th>
<th>Trip Count</th>
<th>Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_mult</td>
<td>II Violation</td>
<td>-</td>
<td>80</td>
<td>800.000</td>
<td>-</td>
<td>81</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>Row_Col</td>
<td>II Violation</td>
<td>-</td>
<td>78</td>
<td>780.000</td>
<td>7</td>
<td>-</td>
<td>25</td>
<td>yes</td>
</tr>
</tbody>
</table>
Optimizing Memory Layout

```
#include "matrix_mult.h"

void matrix_mult(
    mat_a a[IN_A ROWS][IN_A COLS],
    mat_b b[IN_B ROWS][IN_B COLS],
    mat_prod prod[IN_A ROWS][IN_B COLS])
{
    // Iterate over the rows of the A matrix
    Row: for(int i = 0; i < IN_A ROWS; i++) {
        // Iterate over the columns of the B matrix
        Col: for(int j = 0; j < IN_B COLS; j++) {
            prod[i][j] = 0;
            // Do the inner product of a row of A and col of B
            Product: for(int k = 0; k < IN_B ROWS; k++) {
                prod[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```
- 25x speedup off same code
- No new functional bugs!!
- Can you do better by RTL?
Pragma Crib Sheet: Arrays

- **Map**
  - multiple arrays in same BRAM
  - no perf loss if no scheduling conflicts

- **Reshape**
  - change BRAM aspect ratio to widen ports
  - change linear address to location mapping
  - higher bandwidth on consecutive locations

- **Partition**
  - map 1 array to multiple BRAMs
  - multiple independent ports if no bank conflicts

A lot more you can control; must read UG902
Pragma Crib Sheet: Loops

- **Loop Unroll (full and partial)**
  - amortize loop control overhead
  - increase loop-body size, hence “ILP” and scheduling flexibility

- **Loop Flatten**
  - streamline loop-nest control
  - reduce start/finish stutter

- **Loop Merge**
  - combine loop-bodies of independent loops of same control
  - improve parallelism and scheduling
Additional Control thru Code Structure

for (k=...)
   for (i=...)
      for (j=...)
         C[i][j] += f(i, j, k)

pipelined kernel

unrolled inner loops

C[i][j] += f(i, j, 0) + f(i, j, 1) + ... + f(i, j, k-1)
Don’t Forget HW Basics

- Literal (forced)
  686cyc, 6860ns (1x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00 ns</td>
<td>1.878 ns</td>
<td>2.70 ns</td>
<td>0</td>
<td>1</td>
<td>39</td>
<td>214</td>
</tr>
</tbody>
</table>

- Vitis Default
  80cyc, 800ns (8.6x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00 ns</td>
<td>3.772 ns</td>
<td>2.70 ns</td>
<td>0</td>
<td>3</td>
<td>128</td>
<td>467</td>
</tr>
</tbody>
</table>

- Pragma Directed
  28cyc, 280ns (25x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00 ns</td>
<td>4.195 ns</td>
<td>2.70 ns</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>190</td>
</tr>
</tbody>
</table>

- Clock go fast . . .
  29cyc, 116ns (60x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.00 ns</td>
<td>2.170 ns</td>
<td>1.08 ns</td>
<td>0</td>
<td>0</td>
<td>132</td>
<td>231</td>
</tr>
</tbody>
</table>

  29cyc, 87ns (79x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.00 ns</td>
<td>2.170 ns</td>
<td>0.81 ns</td>
<td>0</td>
<td>0</td>
<td>250</td>
<td>295</td>
</tr>
</tbody>
</table>

  57cyc, 114ns (60x)

<table>
<thead>
<tr>
<th>Target</th>
<th>Estimated</th>
<th>Uncertainty</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.00 ns</td>
<td>1.287 ns</td>
<td>0.54 ns</td>
<td>0</td>
<td>1</td>
<td>211</td>
<td>263</td>
</tr>
</tbody>
</table>
When this takes only minutes, a little trial-and-error is okay (just a little!!!)

When not good enough after backend, add pragmas

When good enough, validate using co-simulation

HLS & analysis algorithm & testbench reference

RTL backend

18-643F23-L8-S33, James C. Hoe, CMU/ECE/CALCM, ©2023
Putting it in context (from last time)

• For you to produce “good” structural RTL
  – identify suitable “temporal and spatial pattern”
  – flesh out concrete datapath (bit/cycle exact)
  – develop correct and efficient control sequencing

• C-to-HW (i.e., C-to-RTL) compiler bridges the gap between functionality and implementation
  – extract parallelism from a sequential specification
  – fill in the details below the functional abstraction
  – make good decisions when filling in the details

Vitis HLS does its part (under your direction)
  fast and without mistakes
Parting Thoughts

• Vitis HLS doesn’t turn program into HW
• Vitis HLS doesn’t turn programmer into HW designer
• Multifaceted benefits to HW designer
  – algo. development/debug/validate in SW
  – pragma steering (no RTL hacking, machine tuning)
  – fast analysis and visualization
  – data type support
    it is about more than adding “double” to Verilog
  – built-in, stylized IP interfaces
  – integration with the rest of Vitis and Zynq!!

Can we turn HPC programmers into HW designers?