L10: Memory and Data Movement Optimization

Cong (Callie) Hao
callie.hao@ece.gatech.edu

Assistant Professor
ECE, Georgia Institute of Technology

Sharc-lab @ Georgia Tech https://sharclab.ece.gatech.edu/
Outline

• Burst mode
• Wide bus transaction
• Double Buffer
• Streaming
void test( FIX_TYPE A[100][100], ...) {
    #pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100][100];

    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            A_local[i][j] = A[i][j];
        }
    }

    ...
Bad Practice

for(int i = 0; i < 100; i++) {
    for(int j = 0; j < 100; j+=2) {
        A_local[i][j] = A[i][j];
    }
}

for(int j = 0; j < 100; j++) {
    for(int i = 0; i < 100; i++) {
        A_local[i][j] = A[i][j];
    }
}

Do not burst because memory is not continuous

> 12000 cycles (expected 5000)

> 20000 cycles (expected 10000)
Wider Bus Transaction

```c
void test( FIX_TYPE A[100][100], ...) {
    #pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100][100];

    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            A_local[i][j] = A[i][j];
        }
    }

    ...}
```

Multiple burst reads of length 10000 and bit width 32 in loop xxx has been inferred on port 'mem'

One 32-bit data per cycle

AXI bus is 512 bit wide – 15/16 bandwidth is wasted!
typedef ap_uint<320> MEM_TYPE;

void test( MEM_TYPE A[100*10], ... ) {
    #pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100*100];
    #pragma HLS array_partition variable=A_local cyclic factor=10

    for(int i = 0; i < 100*10; i+=10) {
        #pragma HLS pipeline
        MEM_TYPE data = A[i];
        for(int ii = 0; ii < 10; ii++) {
            A_local[i*10 + ii] = data.range(0 + (ii*32), 31 + (ii*32));
        }
    }
}

[INFO] Multiple burst reads of length 1000
and bit width 512
<table>
<thead>
<tr>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
<th>32b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>320-bit customized datatype</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitioned local arrays
Single Buffer Limitation

• Read-Execute-Write create dependency
Single Buffer Limitation

- Read-Execute-Write create dependency

*PE: processing element
Single Buffer Limitation

• Read-Execute-Write create dependency
Single Buffer Limitation

• Read-Execute-Write create dependency
Single Buffer Limitation

• Read-Execute-Write create dependency

Write
Overlap Read and Write (Pre-fetching)

```java
for(int i = 0; i < N; i++) {
    read(buf_A, i);
    execute(buf_A, buf_B);
    write(buf_B, i);
}
```

```java
read(buf_A, i);
for(int i = 0; i < N; i++) {
    execute(buf_A, buf_B);
    write(buf_B, i);
    if(i < N-1) read(buf_A, i+1);
}
```
Overlap Read and Write (Pre-fetching)

for(int i = 0; i < N; i++) {
    read(buf_A, i);
    execute(buf_A, buf_B);
    write(buf_B, i);
}

read(buf_A, i);
for(int i = 0; i < N; i++) {
    execute(buf_A, buf_B);
    write(buf_B, i);
    if(i < N-1) read(buf_A, i+1);
}

How to resolve this dependency?

Hidden…
Double Buffer (Ping-Pong Buffer)

Read

Execute

Write
Double Buffer (Ping-Pong Buffer)
Double Buffer (Ping-Pong Buffer)

- Read
  - Execute
  - Write
- Read
  - Execute
  - Write
Double Buffer (Ping-Pong Buffer)

```c
read(buf_A_Ping, i);

for(int i = 0; i < N; i++) {
    if( i % 2 == 0 ){
        execute(buf_A_Ping, buf_B);
        write(buf_B, i);
        if( i < N-1 ) read(buf_A_Pong, i+1);
    }
    else if (i % 2 == 1) {
        execute(buf_A_Pong, buf_B);
        write(buf_B, i);
        if( i < N-1 ) read(buf_A_Ping, i+1);
    }
}
```
Double Buffer (Ping-Pong Buffer)

```c
for(int i = 0; i < N; i++) {
    if( i % 2 == 0 ) {
        execute(buf_A_Ping, buf_B_Ping);
        if( i > 0 ) write(buf_B_Pong, i-1);
        if( i < N-1 ) read(buf_A_Pong, i+1);
    }
    else if (i % 2 == 1) {
        execute(buf_A_Pong, buf_B_Pong);
        if( i > 0 ) write(buf_B_Ping, i-1);
        if( i < N-1 ) read(buf_A_Ping, i+1);
    }
}
if( xxx )
    write(buf_B_Ping, i);
else
    write(buf_B_Pong, i);
```
Ping-Pong Buffer Pros and Cons

• Pros
  o Overlapped execution – shorter latency

• Cons
  o Memory overhead (4x)
  o Programmer’s effort
Producer-Consumer Paradigm

- Another way to explain “overlapped execution”
Data Streaming for Producer-Consumer

• **Stream**: an unbounded, continuously updating data set
  - Unbounded means “of **unknown or of unlimited** size”
  - A sequence of data flowing unidirectionally between a source (producer) process and a destination (consumer) process

• **Example**: real-time video, audio, etc.

*Started processing – don’t wait for the entire*
Data Streaming for Producer-Consumer

• **Enabled by FIFO (first-in first-out) buffers**
  o The consumer process can start accessing the data inside the FIFO buffer as soon as the producer inserts the data into the buffer
  o If the buffer is full/empty, automatically stalls
Data Streaming in HLS

- **Step 1: create FIFOs using `hls::stream<type>`**
  - Specify a depth (how large the FIFO is)

- **Step 2: organize into two functions or loops**
  - One writing to FIFO, one reading from FIFO

- **Step 3: apply dataflow pragma**
void test( FIX_TYPE* A, FIX_TYPE* sum ) {
    #pragma HLS dataflow
    hls::stream<FIX_TYPE> buffer;
    #pragma HLS STREAM variable=buffer depth=10

    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;
    }
}
Data Streaming in HLS – Example

```c
void test( FIX_TYPE* A, FIX_TYPE* sum ) {
    #pragma HLS dataflow
    hls::stream<FIX_TYPE> buffer;
    #pragma HLS STREAM variable=buffer depth=10

    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;
    }
}
```
void test( FIX_TYPE* A, FIX_TYPE* sum ) {
    #pragma HLS dataflow

    hls::stream<FIX_TYPE> buffer;
    #pragma HLS STREAM variable=buffer depth=10

    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;
    }
}
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions
void dataflow_top(Input0, Input1, Output0, Output1) {
    for (int i = 0; i < N; i++) {
        #pragma HLS dataflow

        Streaming_Buffer C0, C1, C2;

        func1(read_Input0, read_Input1, write_C0, write_C1);
        func2(read_C0, read_C1, write_C2);
        func3(read_C2, write_Output0, write_Output1);
    }
}
Synthesis v.s. Implementation

- This is what you get from **Synthesis**
- Performance and Resource **Estimation**

---

<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</th>
<th>Latency</th>
<th>Interval</th>
<th>Trip</th>
<th>Count</th>
<th>Pipelined</th>
<th>BRAM</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>-</td>
<td>31236</td>
<td>3.120E5</td>
<td>-</td>
<td>31237</td>
<td>-</td>
<td>no</td>
<td>38</td>
<td>1</td>
<td>3402, 4427</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fooA</td>
<td>-</td>
<td>10017</td>
<td>1.000E5</td>
<td>-</td>
<td>10017</td>
<td>-</td>
<td>no</td>
<td>0</td>
<td>0</td>
<td>329, 511</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fooB</td>
<td>-</td>
<td>5017</td>
<td>5.017E4</td>
<td>-</td>
<td>5017</td>
<td>-</td>
<td>no</td>
<td>0</td>
<td>0</td>
<td>328, 510</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1_VITIS_LOOP_420_1</td>
<td>-</td>
<td>10009</td>
<td>1.000E5</td>
<td>11</td>
<td>1</td>
<td>10000</td>
<td>yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2</td>
<td>-</td>
<td>11200</td>
<td>1.120E5</td>
<td>112</td>
<td>1</td>
<td>100</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• **To run Implementation**
• **If run into errors:**
  o Try “lastyear_vitis_hls”
### Synthesis v.s. Implementation

#### Timing

<table>
<thead>
<tr>
<th>Timing</th>
<th>Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>10.00 ns</td>
</tr>
<tr>
<td>Target</td>
<td>7.300 ns</td>
</tr>
<tr>
<td>Estimated</td>
<td>2.70 ns</td>
</tr>
</tbody>
</table>

#### Utilization Estimates

**Synthesis**

<table>
<thead>
<tr>
<th>Name</th>
<th>BRAM_18K</th>
<th>DSP</th>
<th>FF</th>
<th>LUT</th>
<th>URAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSP</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Expression</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>414</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instance</td>
<td>6</td>
<td>0</td>
<td>2433</td>
<td>3248</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td>32</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>605</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>969</td>
<td>160</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>38</td>
<td>1</td>
<td>3402</td>
<td>4427</td>
<td>0</td>
</tr>
<tr>
<td>Available</td>
<td>40</td>
<td>40</td>
<td>16000</td>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>Utilization (%)</td>
<td>95</td>
<td>2</td>
<td>21</td>
<td>55</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Implementation

**Resource Usage**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLICE</td>
<td>1510</td>
</tr>
<tr>
<td>LUT</td>
<td>3220</td>
</tr>
<tr>
<td>FF</td>
<td>6109</td>
</tr>
<tr>
<td>DSP</td>
<td>4</td>
</tr>
<tr>
<td>BRAM</td>
<td>38</td>
</tr>
<tr>
<td>SRL</td>
<td>303</td>
</tr>
</tbody>
</table>

**Final Timing**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP required</td>
<td>10.000</td>
</tr>
<tr>
<td>CP achieved post-synthesis</td>
<td>7.820</td>
</tr>
<tr>
<td>CP achieved post-implementation</td>
<td>8.759</td>
</tr>
</tbody>
</table>

Timing met
Summary

- Widening the memory port
- Double buffer to hide the data loading latency
- More advanced (challenging) technique: streaming and dataflow
  - BUT! Dataflow architecture is really efficient and will be very promising in the future!