CS-470: Homework Set #3
High-level Synthesis

GAO Xufeng

May 22, 2022

1 Introduction

This report gives an overview on the design optimization of 5 loop kernels. The naive implementation of each kernel is analyzed first. Then any possible optimizations such as unroll and pipeline are proposed, which are followed by the explanations and comparative results in terms of different performance metrics (e.g., area and timing). Finally, a conclusion on this homework is provided. To simplify the design, the clock cycle is set up in default, which is 10ns.

2 Kernel Designs

2.1 Kernel-1

The naive code for kernel-1 is shown below.

```
#include "kernel1.h"
void kernel1( int array[ARRAY_SIZE] )
{
    int i;
    loop:for(i=0; i<ARRAY_SIZE; i++)
        array[i] = array[i] * 5;
}
```

2.1.1 Analysis

Each iteration uses different index i, so there is no inter-loop dependency and we can apply a **PIPELINE** pragma to the loop. Thus, it is possible to start a new loop iteration every clock cycle, meaning that the time interval between starting successive loop iterations called the initiation interval (II) is 1. Because each iteration is independent, **UNROLL** pragma is possible to further shrink the time, but it may also use too much hardware units.

2.1.2 Synthesis Comparison

Performance Estimates: Because the code remains unchanged, both the naive and optimized implementations gives the same iteration latency and trip count. However, with the **PIPELINE** enabled, the initiation interval is optimized to 1 so that the loop latency are halved (e.g., 2048 to 1024 cycles), which led to a 2-times better computation efficiency than before in the total latency (e.g., 2049 to 1026 cycles). With a complete loop unrolling we obtain an smaller clock period but the latency is almost the same, meaning that our hardware sources (e.g., memory) are limited.

Implementation	Total Latency	Loop Latency	Iteration Latency	Iteration Interval	Trip Count	Clock (ns)
Naive	**2049**	**2048**	**2**	**-**	**1024**	**7.11**
PIPELINE (optimized)	**1026**	**1024**	**2**	**1**	**1024**	**7.11**
UNROLL	**1023**	**1023**	**-**	**1024**	**-**	**4.45**

Table 1: Comparison of performance estimates for kernel-1

Utilization Estimates: Compared to naive implementation, the **PIPELINE** enabled one uses 7 (23%) less FFs but 12 (11%) more LUTs. There is only a minor difference in area. However, the **UNROLL** case consumes a large number of hardware units, which is not a good choice in terms of area.

Implementation	BRAM_18K (unit)	DSP48E (unit)	FF (unit)	LUT (unit)
Naive	**0**	**0**	**35**	**108**
PIPELINE (optimized)	**0**	**0**	**27**	**120**
UNROLL	**0**	**0**	**33728**	**53401**

Table 2: Comparison of utilization estimates for kernel-1

Thus, the kernel-1 is optimized with **PIPELINE** pragma only:

```
#include "kernel1.h"
void kernel1( int array[ARRAY_SIZE] )
{
    int i;
    for(i=0; i<ARRAY_SIZE; i++){
    #pragma HLS PIPELINE
        array[i] = array[i] * 5;
    }
}
```

The corresponding scheduled data-flow graph is shown as:

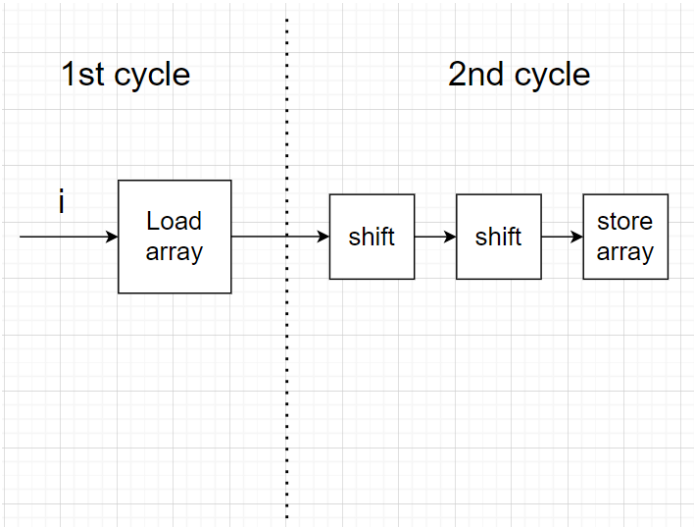

Figure 1: Scheduled dataflow graph of the loop body

Based on the synthesis report and performance view provided by Vivado HLS, the iteration latency for the loop is 2 cycles. The first cycle is used to **LOAD** array value from memory. In the second

cycle, the result is calculated by the combinatory **SHIFT** and **ADD** operations (i.e., uses much less cycles than normal **MUL** operation), which then be **STORE** back to memory.

2.2 Kernel-2

The naive code for kernel-2 is shown below.

```
#include "kernel2.h"
void kernel2( int array[ARRAY_SIZE] )
{
    int i;
    for(i=3; i<ARRAY_SIZE; i++)
        array[i] = array[i-1] + array[i-2] * array[i-3];
}
```

2.2.1 Analysis

Direct **PIPELINE** can be used to achieve latency reduction but it's not sufficient due to the inter-loop dependency and frequent memory accesses for each iteration. To obtain more performance, the intermediate results of loop are stored in three auxiliary registers instead of reading from memory, which reduces memory reads for each iteration to zero and requires less iteration latency. Thus, it is possible to achieve the II of 1. The optimized kernel-2 is displayed below.

```
#include "kernel2.h"
void kernel2( int array[ARRAY_SIZE] )
{
    int i, reg2, reg1, reg0, result;
    // assign first 3 array values
    reg2 = array[2];
    reg1 = array[1];
    reg0 = array[0];
    for(i=3; i<ARRAY_SIZE; i++){
    #pragma HLS PIPELINE
        result = reg2 + reg1 * reg0;
        // shift register values
        reg0 = reg1;
        reg1 = reg2;
        reg2 = result;
        // store result in memory
        array[i] = result;
    }
}
```

2.2.2 Synthesis Comparison

Performance Estimates: As the results show, the optimized code has the same trip count but a more advanced timing efficiency than naive version. The iteration latency is reduced to 2 cycles, which leads to a 4 times improvement in the total latency (e.g., 4085 to 1035 cycles). Due to the pipeline, the best II of 1 is achieved.

Implementation	Total Latency	Loop Latency	Iteration Latency	Iteration Interval	Trip Count	Clock (ns)
Naive	**4085**	**4084**	**4**	**-**	**1021**	**6.58**
Optimized	**1025**	**1021**	**2**	**1**	**1021**	**6.58**

Table 3: Comparison of performance estimates for kernel-2

Utilization Estimates: Compared to the naive version, the optimized version uses more hardware units, which are 29 (18%) more FFs and 32 (16%) more LUTs. This is to be expected, because the optimized version introduces additional registers.

Implementation	BRAM_18K (unit)	DSP48E (unit)	FF (unit)	LUT (unit)
Naive	**0**	**0**	**161**	**197**
Optimized	**0**	**0**	**190**	**229**

Table 4: Comparison of utilization estimates for kernel-2

The corresponding scheduled data-flow graph is shown as: The corresponding scheduled data-flow graph is shown as:

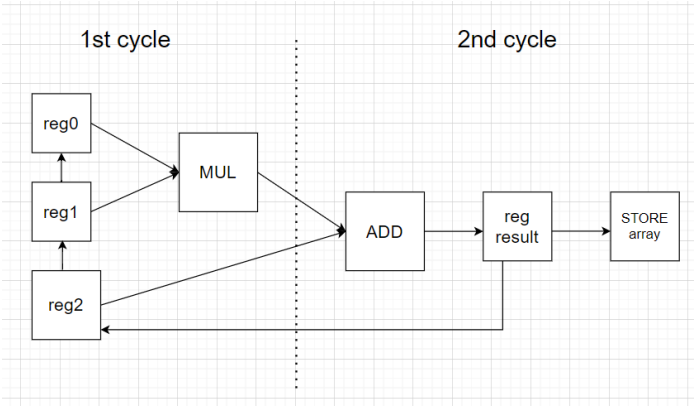

Figure 2: Scheduled dataflow graph of the loop body

Based on the synthesis report and performance view provided by Vivado HLS, the iteration latency for the loop is 2 cycles. To make things clear, I denote the registers I used, namely reg0, reg1, reg3 and reg_result with box to denote MUXs (i.e., the unit to store data in vivado synthesis). The first cycle is used to perform **MUL** operation between reg0 and reg1. In the second cycle, reg2 is added to the multiplication result and is written back to memory. Because this is dataflow chart, the data transfers between registers will work when data is available, e.g., result to reg2.

2.3 Kernel-3

The naive code for kernel-3 is shown below.

```
#include "kernel3.h"
void kernel3( float hist[ARRAY_SIZE], float weight[ARRAY_SIZE], int
↪  index[ARRAY_SIZE]){
  for (int i=0; i<ARRAY_SIZE; ++i) {
    hist[index[i]] = hist[index[i]]+ weight[i];
    }
}
```

2.3.1 Analysis

There may be a Read-After-Write (RAW) data dependency on the array *hist* because *hist* is not accessed sequentially but in an arbitrary order determined by array *index*. Based on Table. 5, the processor may need to stall for 7 cycles (i.e., If the result is available after the floating point addition is complete) if the current iteration requires the result of previous iteration. Direct **PIPELINE** can only provide limited latency reduction, i.e., a II of 7 is achieved. To obtain more performance, the strategy used in kernel-2 is applied again, that is introducing auxiliary registers to detect the dependency manually, as shown below. The best II I achieved is 4, which equals to the operation latency required by float addition.

```
#include "kernel3.h"
void kernel3( float hist[ARRAY_SIZE], float weight[ARRAY_SIZE], int
↪  index[ARRAY_SIZE]){
  int index0, index1, index2, index3;
        float hist0, hist1, hist2, hist3, weight3;

        // read first 3 index values
        index0 = index[0];
        index1 = index[1];
        index2 = index[2];

        // calculate hist values
        hist0 = hist[index0] + weight[0];

        // check dependency manually
        if(index1 == index0){
                index0 = -1;  // mark dependency index
                hist1 = hist0 + weight[1];
        }else{
                hist1 = hist[index1] + weight[1];
        }

        if(index2 == index1){
                index1 = -1;
                hist2 = hist1 + weight[2];
        }else if(index2 == index0){
                index0 = -1;
                hist2 = hist0 + weight[2];
        }else{
                hist2 = hist[index2] + weight[2];
        }

        for (int i=3; i<ARRAY_SIZE; ++i) {
  #pragma HLS PIPELINE
    index3 = index[i];
    weight3 = weight[i];

    // check dependency manually
```

```
    if(index3 == index2){
            index2 = -1;
            hist3 = hist2 + weight3;
    }else if(index3 == index1){
            index1 = -1;
            hist3 = hist1 + weight3;
    }else if(index3 == index0){
            index0 = -1;
            hist3 = hist0 + weight3;
    }else{
            hist3 = hist[index3] + weight3;
    }

    // only write back independent hist values to memory
    if(index0 != -1){
            hist[index0] = hist0;
    }

    // shift registers
    index0 = index1; hist0 = hist1;
    index1 = index2; hist1 = hist2;
    index2 = index3; hist2 = hist3;
  }

  // write back the last 3 values to memory
        if(index0 != -1){
                hist[index0] = hist0;
        }
        if(index1 != -1){
                hist[index1] = hist1;
        }
        if(index2 != -1){
                hist[index2] = hist2;
        }
}
```

2.3.2 Synthesis Comparison

Performance Estimates: From the results, if only **PIPELINE** pragma is used, the improved performance (12.5%) is quite limited. With the help of auxiliary registers, the timing is improved a lot compared to the naive version, which is 50% less in total latency (e.g., 8193 to 4106 cycles). Because the usage of registers for performing float operation for each iteration is more intensive, the clock period is expected to be larger than before (e.g., 7.72 to 8.56ns).

Utilization Estimates: Similar to the results of kernel-2, my final solution requires a large amount of hardware units because the usage of auxiliary registers. This consumption should be carefully considered if area is a constraint in the design.

Implementation	Total Latency	Loop Latency	Iteration Latency	Iteration Interval	Trip Count	Clock (ns)
Naive	**8193**	**8192**	**8**	**-**	**1024**	**7.72**
PIPELINE	**7170**	**7168**	**8**	**7**	**1024**	**7.72**
Optimized	**4106**	**4087**	**7**	**4**	**1021**	**8.56**

Table 5: Comparison of performance estimates for kernel-3

Implementation	BRAM_18K (unit)	DSP48E (unit)	FF (unit)	LUT (unit)
Naive	**0**	**2**	**364**	**255**
PIPELINE	**0**	**2**	**367**	**266**
Optimized	**0**	**2**	**983**	**1330**

Table 6: Comparison of utilization estimates for kernel-3

2.4 Kernel-4

The naive code for kernel-4 is shown below.

```
#include "kernel4.h"
void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
    for (int i=offset+1; i<ARRAY_SIZE-1; ++i)
    {
        array[offset] = array[offset]-index[i]*array[i]+index[i]*array[i+1];
    }
}
```

2.4.1 Analysis

There may be a RAW data dependency on the *array* in the case where writing into *array[offset]* and reading from *array[i]* or *array[i+1]* at the same time. Moreover, because only the final value of *array[offset]* is meaningful, the memory write-back operation for each iteration is useless, which causes redundant latency. Thus, the code is optimized as follow. As the code shows, the *sum* register is used to store intermediate results for each iteration, which avoids data dependency and then may achieve II of 1.

```
#include "kernel4.h"
void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
    int sum = array[offset];
    for (int i=offset+1; i<ARRAY_SIZE-1; ++i) {
    #pragma HLS PIPELINE
        sum = sum - index[i] * array[i] + index[i] * array[i+1];
    }
    array[offset] = sum;
}
```

I also try to optimize the code further, such as use one more register to store *array[i]* so that for each iteration only 2 memory reads are required (e.g., *index[i]* and *array[i+1]*), or partition the array so that we can access the two array elements in the same cycle, or combine the 2 multiply operations

into 1 to reduce iteration latency. However, all these implementations use more hardware units than the optimized version proposed above but give the same timing performance, which are then not considered anymore.

2.4.2 Synthesis Comparison

Performance Estimates: Here the total latency and loop latency should be estimated from the iteration latency and iteration interval, because the loop boundary varies for each different offset value. The ARRAY_SIZE is preset to 1024, thus the trip count N has a possible value from 1 to 1022. For Naive version, the iteration latency is 8, meaning that the loop latency is 8N and the total latency is 8N+1 (i.e., one more cycle for start state). For optimized version, II of 1 is achieved, thus the loop latency is N+7 and total latency is N+10 (i.e., 3 more cycles for start state, memory read and write for *sum*)[1]. Overall, the max timing performance is 8 times better because of the optimization, which led to an improvement in the total latency from 8177 to 1032 cycles.

Implementation	Total Latency		Loop Latency		Iteration Latency	Iteration Interval	Trip Count
	Min	Max	Min	Max			
Naive	**9**	**8177**	**8**	**8176**	**8**	**-**	**N**
Optimized	**11**	**1032**	**8**	**1029**	**7**	**1**	**N**

Table 7: Comparison of performance estimates for kernel-4

Utilization Estimates: There is only a minor difference in area. Compared to naive version, the optimized version uses 3 (3%) more FFs and 63 (39%) more LUTs compared to the naive version.

Implementation	BRAM_18K (unit)	DSP48E (unit)	FF (unit)	LUT (unit)
Naive	**0**	**8**	**115**	**163**
Optimized	**0**	**8**	**118**	**226**

Table 8: Comparison of utilization estimates for kernel-4

The corresponding scheduled data-flow graph is shown as:

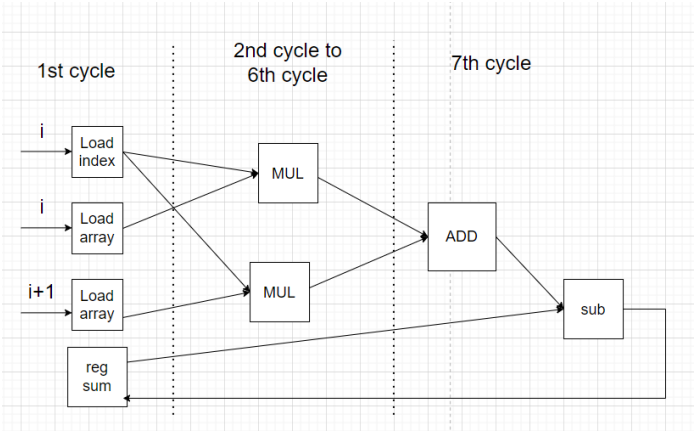

Figure 3: Scheduled dataflow graph of the loop body

Based on the synthesis report and performance view provided by Vivado HLS, the iteration latency for the loop is 7 cycles. To make things clear, I denote the register reg_sum with box to denote MUXs (i.e., the unit to store data in vivado synthesis). The first cycle is used to **LOAD** array and

[1]Three more cycles are observed from the Performance view of Vivado HLS

index values. Then two **MUL** operations are performed until the end of cycle 6. In the final cycle, **ADD** and **SUB** are performed to give the final result which is directly written back to reg_sum.

2.5 Kernel-5

The naive code for kernel-5 is shown below.

```
#include "kernel5.h"

float kernel5(float bound, float a[ARRAY_SIZE], float b[ARRAY_SIZE])
{
    int i=0;
    float sum;
    while (sum<bound && i<ARRAY_SIZE)
    {
        sum = a[i] + b[i];
        i++;
    }
    return sum;
}
```

2.5.1 Analysis

There is a control dependency on *sum* which determines the exit condition of the loop. The naive code is non-speculative, which means that a new iteration can only start after the previous sum has been compared to the bound so that a long iteration latency is caused. Direct **PIPELINE** is not sufficient. Thus, the code is optimized as follow.

```
#include "kernel5.h"

float kernel5(float bound, float a[ARRAY_SIZE], float b[ARRAY_SIZE])
{
    int i=0;
    float sum;
    while (sum<bound && i<ARRAY_SIZE)
    {
        #pragma HLS DEPENDENCE false
        #pragma HLS PIPELINE
        sum = a[i] + b[i];
        i++;
    }
    return sum;
}
```

Because there is no data dependency, **FALSE DEPENDENCY HINT** is used together with **PIPELINE** to implement speculative execution where the operation is performed before it is known whether it is actually needed. If it turns out the operation was not needed after all, the loop is break and the redundant results are ignored. Thus, it is possible to achieve II of 1.

2.5.2 Synthesis Comparison

Performance Estimates: Similar as kernel-4, this code also has variable loop boundaries. Thus, we assume a trip count of N (range: 1 to 1023). For naive version, due to the iteration latency of 6, the corresponding loop latency is 6N and the total latency is 6N+1 based on synthesis schedule. For optimized version, because the II of 1 is achieved, the loop latency is N+6 and total latency is N+7 (i.e., 1 more cycle for start state). Overall, the max timing performance is 6 times better because of the optimization, which led to an improvement in the total latency from 6139 to 1030 cycles.

Implementation	Total Latency		Loop Latency		Iteration Latency	Iteration Interval	Trip Count
	Min	Max	Min	Max			
Naive	**1**	**6139**	**0**	**6138**	**6**	**-**	**N**
Optimized	**7**	**1030**	**6**	**1029**	**6**	**1**	**N**

Table 9: Comparison of performance estimates for kernel-5

Utilization Estimates: We can see the optimized version uses much less hardware units compared to naive version.

Implementation	BRAM_18K (unit)	DSP48E (unit)	FF (unit)	LUT (unit)
Naive	**0**	**2**	**461**	**381**
Optimized	**0**	**0**	**109**	**158**

Table 10: Comparison of utilization estimates for kernel-5