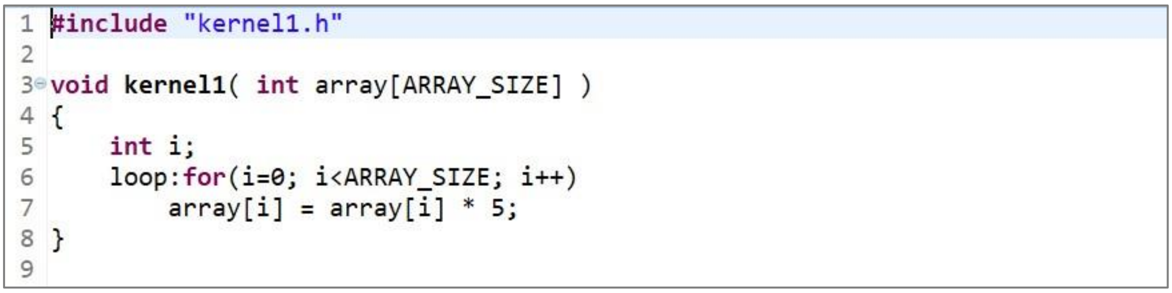
Homework 3 (Vivado HLS) Report

Introduction

This report presents an overview on the design optimization of 8 loop kernels as part of the Advanced Computer Architecture course (CS-470) at EPFL. It highlights the analysis of each kernel's naïve implementation followed by the optimized implementation. The explanations and comparative results (in terms of area and timing) for all the optimizations are also presented. Please note that the results are given in terms of clock cycles assuming a clock period of 10ns.

Kernel-1

The code for the loop kernel is shown below. The same code was used for the optimized version too.



```
#include "kernel1.h"

void kernel1( int array[ARRAY_SIZE] )
{
    int i;
    loop:for(i=0; i<ARRAY_SIZE; i++)
        array[i] = array[i] * 5;
}

```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive.
2. Each iteration is independent since it uses a different index i.
3. Hence, it was possible to achieve an initiation interval of 1.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The optimized version has the same trip count and iteration latency since the code was unchanged. However, the loop latency is 2 times better because of pipelining, which led to an improvement in the total latency from 2049 to 1026 cycles.

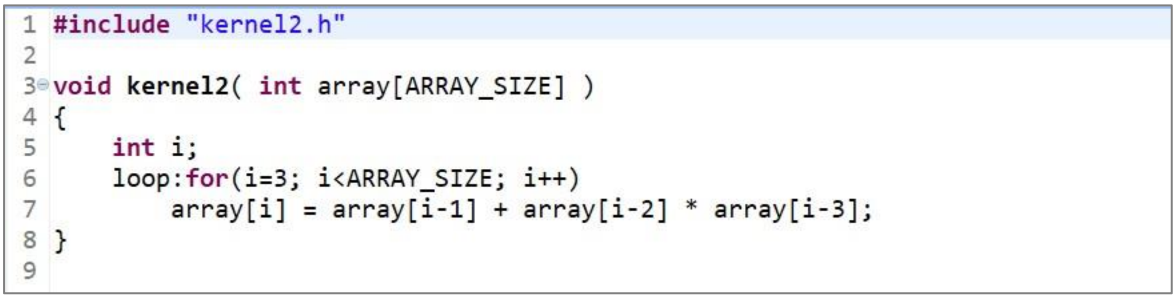
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	2049	2049	2048	2048	2	1024	-
Optimized	1026	1026	1024	1024	2	1024	1

Utilization Estimates: The optimized version uses 12 (10%) more LUTs but 8 (23%) less FFs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	0	0	35	~0	115	~0	0	0
Optimized	0	0	0	0	27	~0	127	~0	0	0

Kernel-2

The code for the loop kernel is shown below. The code was rewritten for the optimized version.

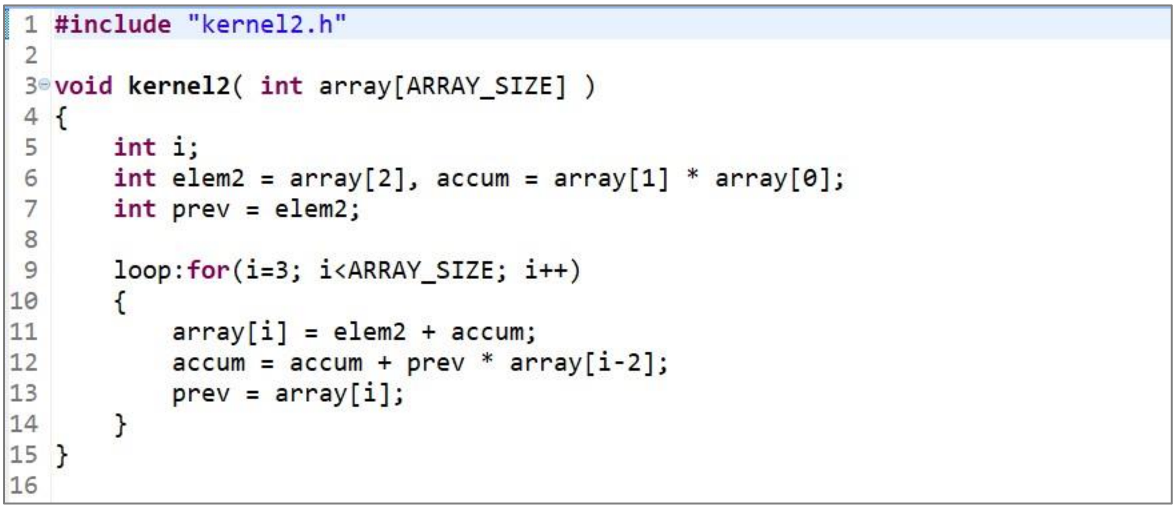


```
#include "kernel2.h"

void kernel2( int array[ARRAY_SIZE] )
{
    int i;
    loop:for(i=3; i<ARRAY_SIZE; i++)
        array[i] = array[i-1] + array[i-2] * array[i-3];
}

```

The optimized version of the code is shown below.



```
#include "kernel2.h"

void kernel2( int array[ARRAY_SIZE] )
{
    int i;
    int elem2 = array[2], accum = array[1] * array[0];
    int prev = elem2;

    loop:for(i=3; i<ARRAY_SIZE; i++)
    {
        array[i] = elem2 + accum;
        accum = accum + prev * array[i-2];
        prev = array[i];
    }
}

```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. The code was rewritten to reduce the number of memory-reads per iteration. The operation within the loop can be seen as: *array[i] = array[2] + accumulation of previous multiplications.*
3. The code was further improved by using a new variable *prev* to store the resulting *array[i]* so that it can be reused in the next iteration. This reduces the need to read *array[i-1]*.
4. Finally, since the resulting code requires to read *array[i-2]* and write *array[i]* every iteration, the inter-iteration dependency on *array* can be disabled using the dependence primitive.
5. Hence, it was possible to achieve an initiation interval of 1.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The optimized version has the same trip count but a better iteration latency after rewriting the code. The loop latency is 5 times better because of pipelining, which led to an improvement in the total latency from 5106 to 1027 cycles.

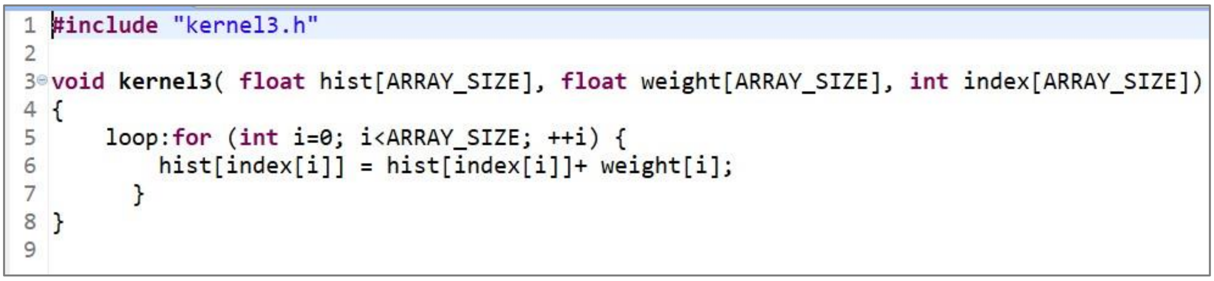
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	5106	5106	5105	5105	5	1021	-
Optimized	1027	1027	1022	1022	3	1021	1

Utilization Estimates: The optimized version uses 3 (2x) more DSP48Es, 59 (40%) more FFs and 120 (54%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	3	~0	145	~0	222	~0	0	0
Optimized	0	0	6	1	204	~0	342	~0	0	0

Kernel-3

The code for the loop kernel is shown below. The same code was used for the optimized version too.



```
#include "kernel3.h"

void kernel3( float hist[ARRAY_SIZE], float weight[ARRAY_SIZE], int index[ARRAY_SIZE])
{
    loop:for (int i=0; i<ARRAY_SIZE; ++i) {
        hist[index[i]] = hist[index[i]]+ weight[i];
      }
}
```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. The iterations are not independent. There may be a RAW data dependency on the *hist* array in the case where at least 2 *index[i]* in the subsequent 7 iterations have the same value.
3. A perfect initiation interval could not be achieved because of the above unpredictability. It cannot be ignored, but we can use a complex logic to check the subsequent 7 *index[i]* values and accumulate the *weight* if necessary. But it will be very expensive in terms of area.
4. Hence, it was only possible to achieve an initiation interval of 7.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The optimized version has the same trip count and iteration latency since the code was unchanged. However, the loop latency is a little better because of pipelining, which led to an improvement in the total latency from 8193 to 7170 cycles.

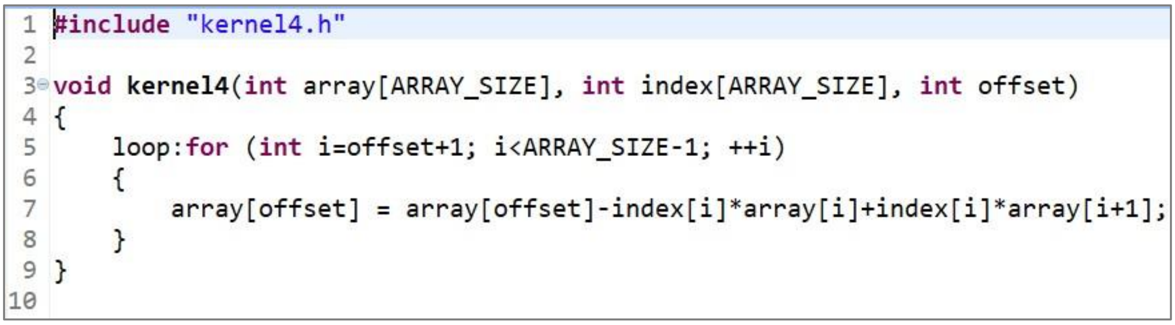
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	8193	8193	8192	8192	8	1024	-
Optimized	7170	7170	7168	7168	8	1024	7

Utilization Estimates: The optimized version uses the same number of DSP48Es but 3 (1%) more FFs and 24 (7.6%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	2	~0	364	~0	316	~0	0	0
Optimized	0	0	2	~0	367	~0	340	~0	0	0

Kernel-4

The code for the loop kernel is shown below. The code was rewritten for the optimized version.

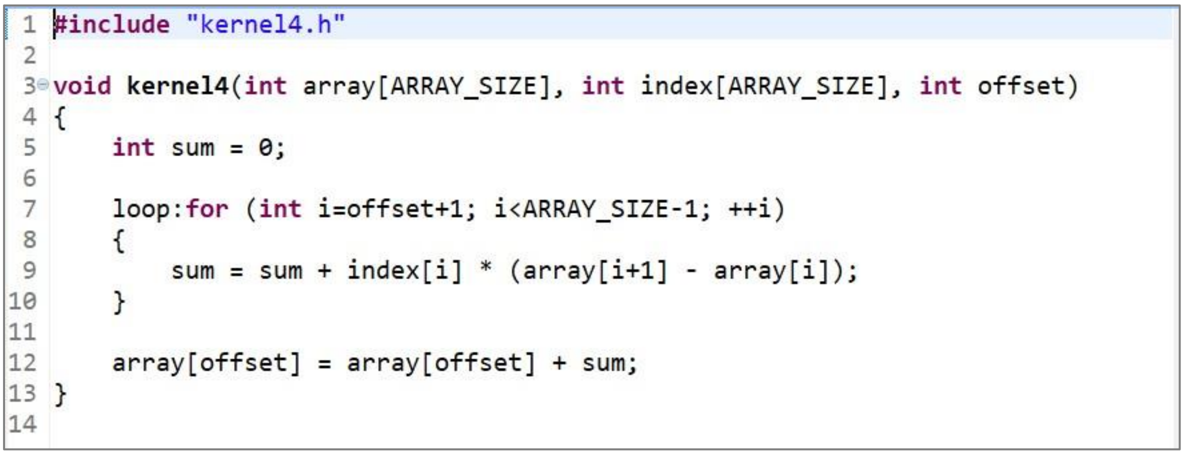


```
#include "kernel4.h"

void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
    loop:for (int i=offset+1; i<ARRAY_SIZE-1; ++i)
    {
        array[offset] = array[offset]-index[i]*array[i]+index[i]*array[i+1];
    }
}

```

The optimized version of the code is shown below.



```
#include "kernel4.h"

void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
    int sum = 0;

    loop:for (int i=offset+1; i<ARRAY_SIZE-1; ++i)
    {
        sum = sum + index[i] * (array[i+1] - array[i]);
    }

    array[offset] = array[offset] + sum;
}

```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. The code was rewritten to avoid the RAW dependency which may arise from writing into *array[offset]* and reading *array[i]* or *array[i+1]* at the same time.
3. Further, the final result just needs to be written in one location, hence, it can be pulled out of the loop to reduce memory operations in the loop. The value is accumulated in *sum* and finally added to *array[offset]* in the end.
4. Hence, it was possible to achieve an initiation interval of 1.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The code has variable loop boundaries; hence, we assume a trip count of N (range: 1 to 1022) to calculate the loop latency (5N: naïve, N+2: optimized) and total latency (5N+1: naïve, N+5: optimized) based on the synthesis schedule. The optimized version has the same trip count but a better iteration latency after rewriting the code. The loop latency is 5 times better because of pipelining, which led to an improvement in the total max latency from 5111 to 1027 cycles.

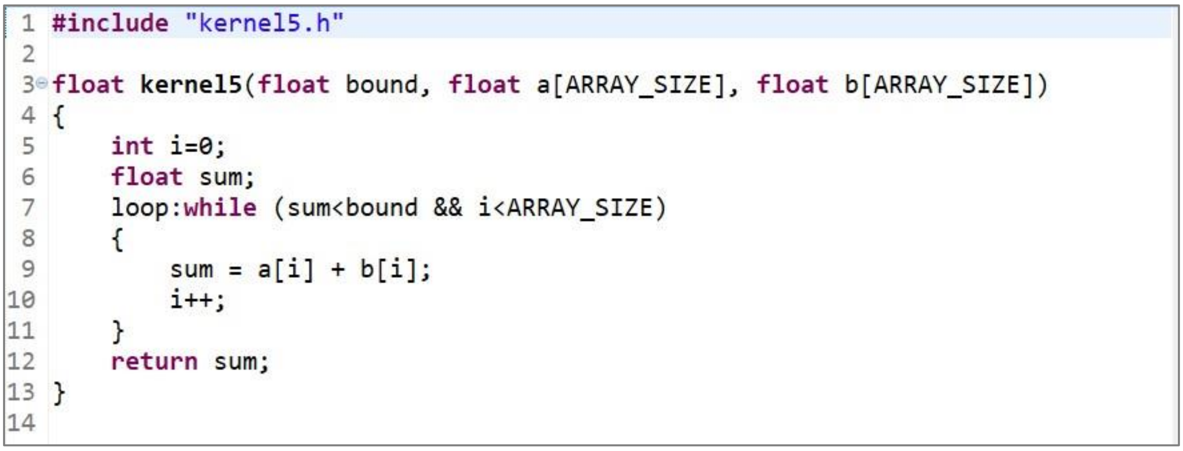
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	6	5111	5	5110	5	N	-
Optimized	6	1027	3	1024	3	N	1

Utilization Estimates: The optimized version uses 3 (50%) less DSP48Es and 93 (34%) less FFs, but 69 (25%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	6	1	272	~0	279	~0	0	0
Optimized	0	0	3	~0	179	~0	348	~0	0	0

Kernel-5

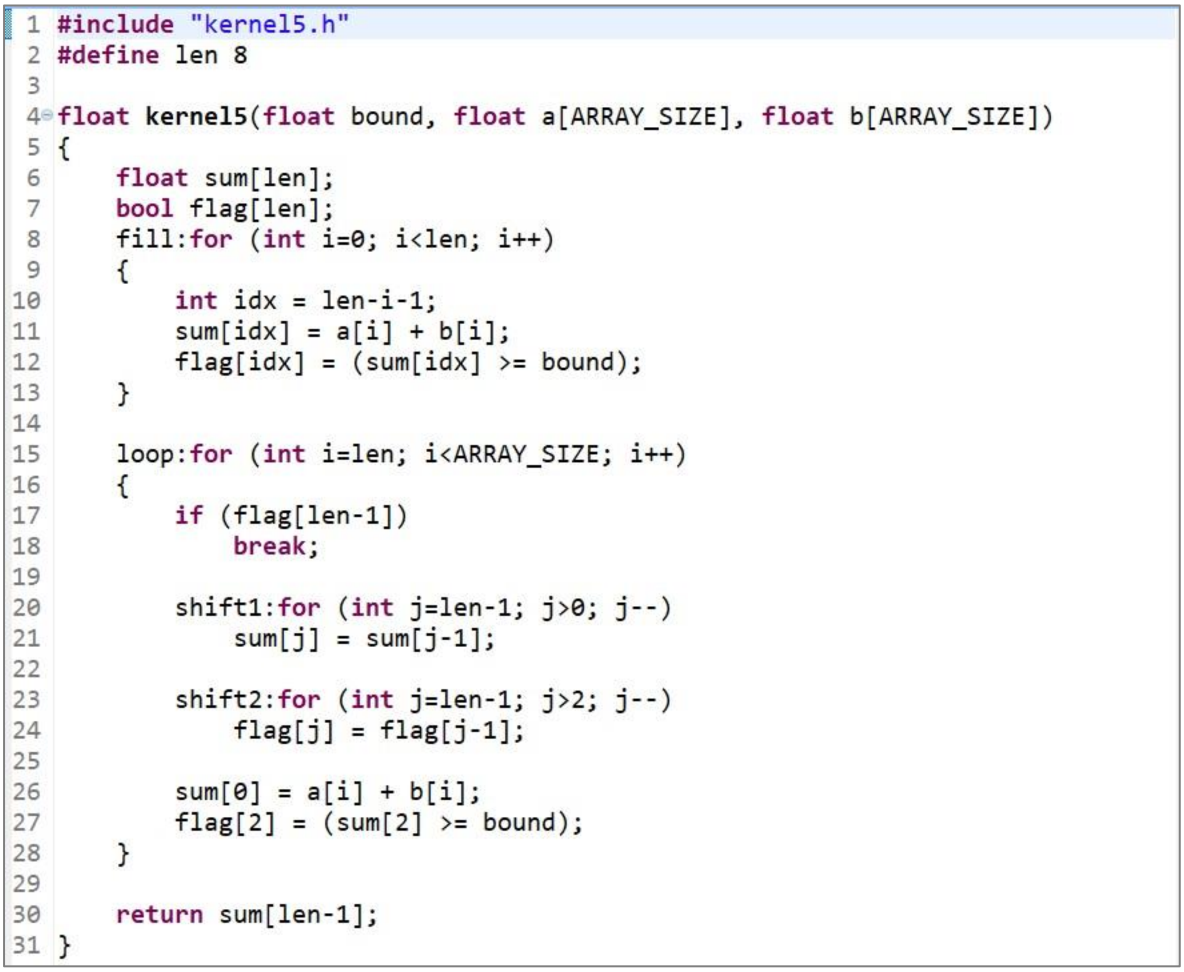
The code for the loop kernel is shown below. The code was rewritten for the optimized version.



```c
#include "kernel5.h"

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

The optimized version of the code is shown below.



```c
#include "kernel5.h"
#define len 8

float kernel5(float bound, float a[ARRAY_SIZE], float b[ARRAY_SIZE])
{
    float sum[len];
    bool flag[len];
    fill:for (int i=0; i<len; i++)
    {
        int idx = len-i-1;
        sum[idx] = a[i] + b[i];
        flag[idx] = (sum[idx] >= bound);
    }

    loop:for (int i=len; i<ARRAY_SIZE; i++)
    {
        if (flag[len-1])
            break;

        shift1:for (int j=len-1; j>0; j--)
            sum[j] = sum[j-1];

        shift2:for (int j=len-1; j>2; j--)
            flag[j] = flag[j-1];

        sum[0] = a[i] + b[i];
        flag[2] = (sum[2] >= bound);
    }

    return sum[len-1];
}
```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. The code was rewritten to remove the control dependency on *sum* which decides the exit condition of the loop. This was done using a shift queue of length 8 to store the *sum* at the end, and check the exit condition using the oldest value which is ready. The shifting loop was unrolled using the unroll directive and *sum* was partitioned using the partition directive.
3. Next, another queue was used to store the exit condition *flag* and this computation runs a little behind the sum computation (3 cycles: based on the comparison latency), effectively using a 6-length queue. The shifting loop was unrolled and *flag* was partitioned, like *sum*.
4. Another loop was added before the main loop to initialize the shift queues for *sum* and *flag*. This loop was pipelined using the pipeline directive, achieving an iteration interval of 1.
5. Finally, a stall of 3 cycles was detected by Vivado due to an intra loop dependency on the *flag* queue, which does not exist as long as the values are written in sequence. Hence, it was disabled using the dependence directive.
6. Hence, it was possible to achieve an initiation interval of 1.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The code has variable loop boundaries; hence, we assume a trip count of N (range: 1 to 1024) to calculate the loop latency (7N: naïve) and total latency (7N+1: naïve) based on the synthesis schedule. The estimates for the optimized version are provided by Vivado HLS, as shown below, for the main loop and the *init* loop.

The optimized version has the similar trip count and iteration latency in different conditions. However, the loop latency is 7 times because of pipelining and the new design, which led to an improvement in the max total latency from 7168 to 1039 cycles.

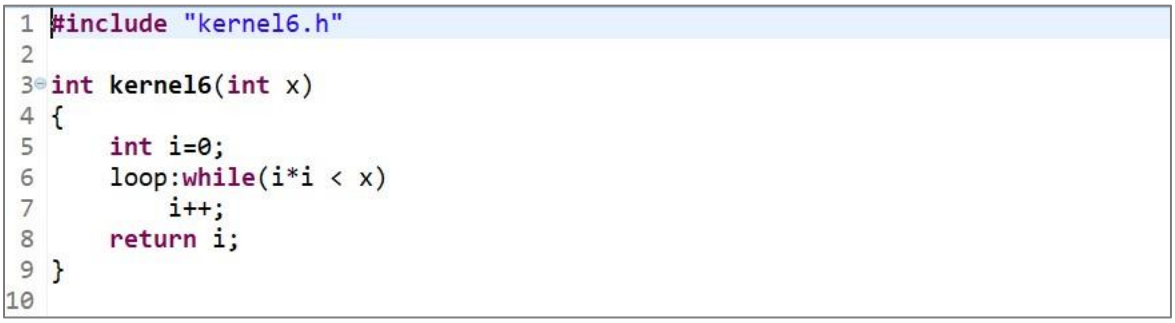
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	8	7168	7	7168	7	N	-
Optimized	24	1039	6	1021	7	1~1016	1
(init loop)	-	-	14	14	8	8	1

Utilization Estimates: The optimized version uses the same number of DSP48Es, but 1037 (225%) more FFs and 530 (110%) more LUTs compared to the naïve version. The performance improvement came at a high cost in area.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	2	~0	462	~0	478	~0	0	0
Optimized	0	0	2	~0	1499	~0	1008	~0	0	0

Kernel-6

The code for the loop kernel is shown below. The code was rewritten for the optimized version.

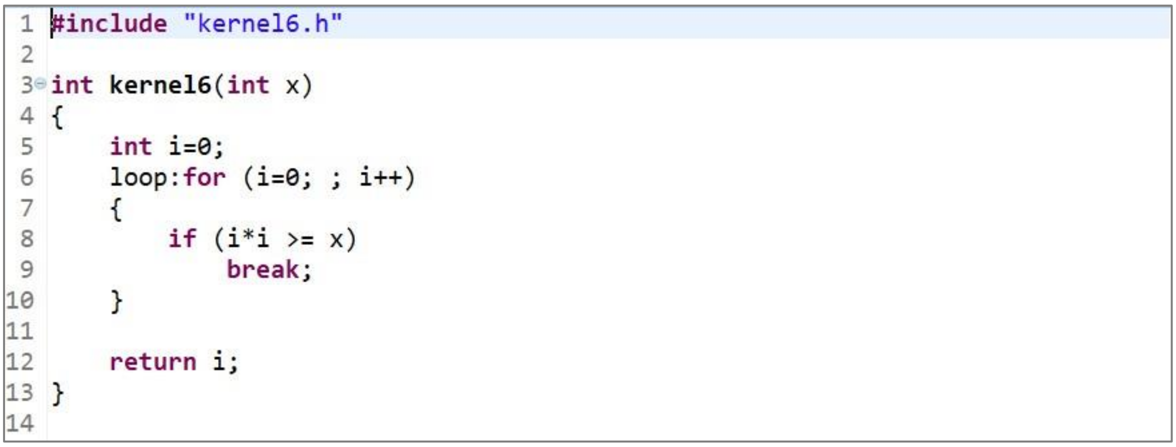


```c
#include "kernel6.h"

int kernel6(int x)
{
    int i=0;
    loop:while(i*i < x)
        i++;
    return i;
}

```

The optimized version of the code is shown below.



```c
#include "kernel6.h"

int kernel6(int x)
{
    int i=0;
    loop:for (i=0; ; i++)
    {
        if (i*i >= x)
            break;
    }

    return i;
}

```

Optimization Steps

The naïve implementation performs just fine (see table). The following observations and changes led to an optimized design:

1. The naïve implementation is decent because the loop has an iteration latency of just 1.
2. The loop can be pipelined using the pipeline directive, but this is not really necessary.
3. The code was rewritten in an attempt to apply directives and remove the control dependency. But even without it, the performance is optimal because of cheap operations within the loop.
4. Hence, it was possible to achieve an initiation interval of 1.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The code has variable loop boundaries; hence, we assume a trip count of N (range: 0 to inf) to calculate the loop latency (N: naïve, N: optimized) and total latency (N+1: naïve, N+1: optimized) based on the synthesis schedule. Both the versions are similar in design and they give the same performance.

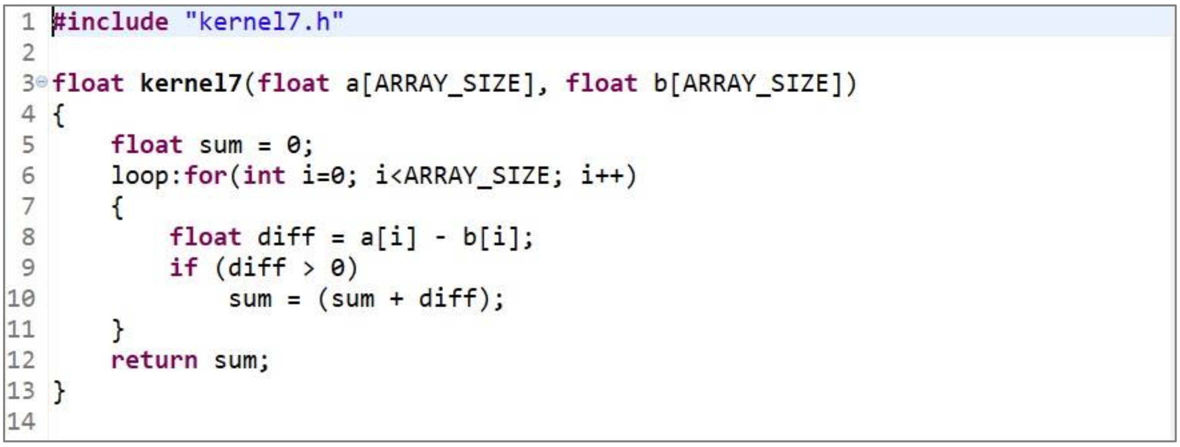
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	1	(N+1)	0	(N)	1	N	-
Optimized	1	(N+1)	0	(N)	1	N	1

Utilization Estimates: The optimized version uses the same number of DSP48Es, 1 (3%) more FF and 6 (7%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	3	~0	34	~0	102	~0	0	0
Optimized	0	0	3	~0	35	~0	108	~0	0	0

Kernel-7

The code for the loop kernel is shown below. The same code was used for the optimized version too.



```
#include "kernel7.h"

float kernel7(float a[ARRAY_SIZE], float b[ARRAY_SIZE])
{
    float sum = 0;
    loop:for(int i=0; i<ARRAY_SIZE; i++)
    {
        float diff = a[i] - b[i];
        if (diff > 0)
            sum = (sum + diff);
    }
    return sum;
}

```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. There is a control dependency on *diff* followed by a data dependency on *sum* which stalls the pipeline. This prevents from achieving a perfect initiation interval.
3. The problem cannot be solved by simple predication as it will involve a float multiplication with the predicate which is a costly operation, and hence, worsen the performance.
4. It could be possible to improve the performance at a high cost of area by using a queue to store the *diff*. It will require complex logic, an inefficient loop to fill the queue and an unrolled loop to shift the queue items. But it will still have the control dependency due to the top item.
5. Hence, it was only possible to achieve an initiation interval of 4.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The optimized version has the same trip count and iteration latency since the code was unchanged. However, the loop latency is 2.5 times better because of pipelining, which led to an improvement in the total latency from 10241 to 4104 cycles.

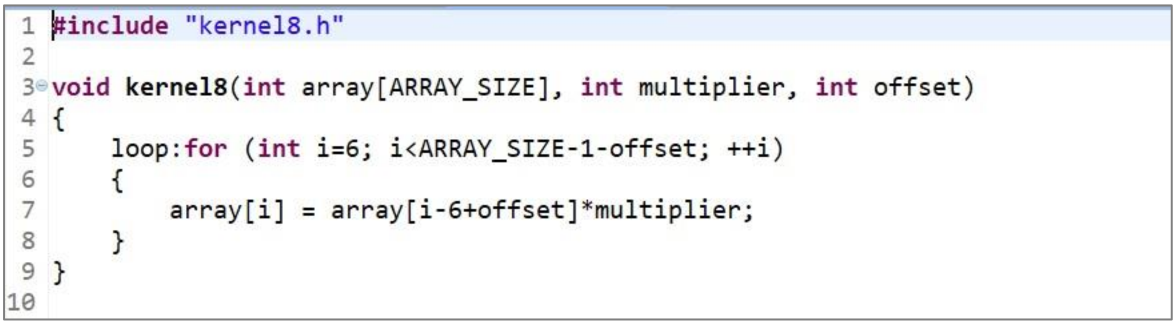
Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	10241	10241	10240	10240	10	1024	-
Optimized	4104	4104	4102	4102	11	1024	4

Utilization Estimates: The optimized version uses the same number of DSP48Es, 63 (14%) more FFs and 41 (8%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	2	~0	457	~0	500	~0	0	0
Optimized	0	0	2	~0	520	~0	541	~0	0	0

Kernel-8

The code for the loop kernel is shown below. The same code was used for the optimized version too.



```
#include "kernel8.h"

void kernel8(int array[ARRAY_SIZE], int multiplier, int offset)
{
    loop:for (int i=6; i<ARRAY_SIZE-1-offset; ++i)
    {
        array[i] = array[i-6+offset]*multiplier;
    }
}

```

Optimization Steps

The naïve implementation does not give very good results (see table). The following observations and changes led to an optimized design:

1. The loop can be pipelined using the pipeline directive. But this is not sufficient.
2. The iterations are not independent. There may be a RAW data dependency on the *array* when the indices are within the iteration latency range. This cannot be known beforehand. Some complex logic of prediction and speculation might help at high cost in terms of area.
3. Hence, it was only possible to achieve an initiation interval of 4.

Synthesis Comparison

The synthesis reports for both the implementations can be compared in terms of timing and area:

Performance Estimates: The code has variable loop boundaries; hence, we assume a trip count of N (range: 1 to 1017) to calculate the loop latency (4N: naïve, 4N: optimized) and total latency (4N+1: naïve, 4N+1: optimized) based on the synthesis schedule. The optimized version simply uses the pipeline directive which could not improve the performance because of the dependencies.

Implementation	Total Latency		Loop Latency		Iteration Latency	Trip Count	Initiation Interval
	Min	Max	Min	Max			
Naïve	5	4069	4	4068	4	N	-
Optimized	5	4069	4	4068	4	N	4

Utilization Estimates: The optimized version uses the same number of DSP48Es, 1 (0.7%) more FF and 5 (2%) more LUTs compared to the naïve version.

Implementation	BRAM_18K		DSP48E		FF		LUT		URAM	
	Unit	%	Unit	%	Unit	%	Unit	%	Unit	%
Naïve	0	0	3	~0	132	~0	250	~0	0	0
Optimized	0	0	3	~0	133	~0	255	~0	0	0

Conclusion

Finally, the design optimizations led to a perfect performance improvement with *II=1* in some *kernels (1, 2, 4, 5)*, a little performance improvement with *II > 1* in other *kernels (3, 7)* and no performance improvement with *II = latency* in the last *kernel (8)*. Just one *kernel (6)* performed good enough without any optimizations in the naïve implementation because of a 1-cycle latency. All the improvements, except for *kernel 5*, were achieved with only a minor cost in terms of area.

Please note that all the optimized kernels have been simulated and synthesized successfully. The optimizations were mostly achieved by rewriting the code logic, using the pipeline directive and occasionally, using the dependence directive. There was only one opportunity to use the unroll directive, and consequently, the partition directive in *kernel 5*. The use of these directives could also improve the performance for some *kernels (3, 7)*, but it will require very complex logic and high area cost. Hence, this approach was avoided.

Along with this report, the project files, naïve and optimized, are attached as two separate zip-files. The C++ files and HTML synthesis reports, naïve and optimized, are also attached separately.

Submitted by: Kushagra Shah (316002)

Dated: 10/05/2021