Course Project: HLS Code Refactoring for High Performance

Edonis Bajraktaraj 19-923-473

May 21, 2025

Kernel 1

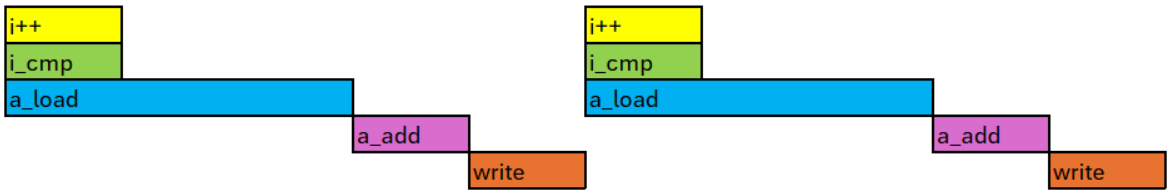

Figure 1: Scheduling of Kernel 1 with directives turned off

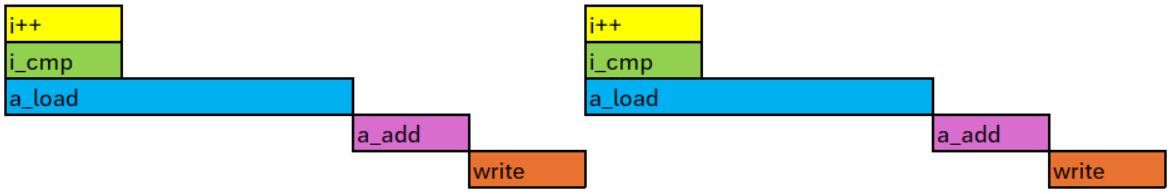

Figure 2: Scheduling of Kernel 1 without any input on directives

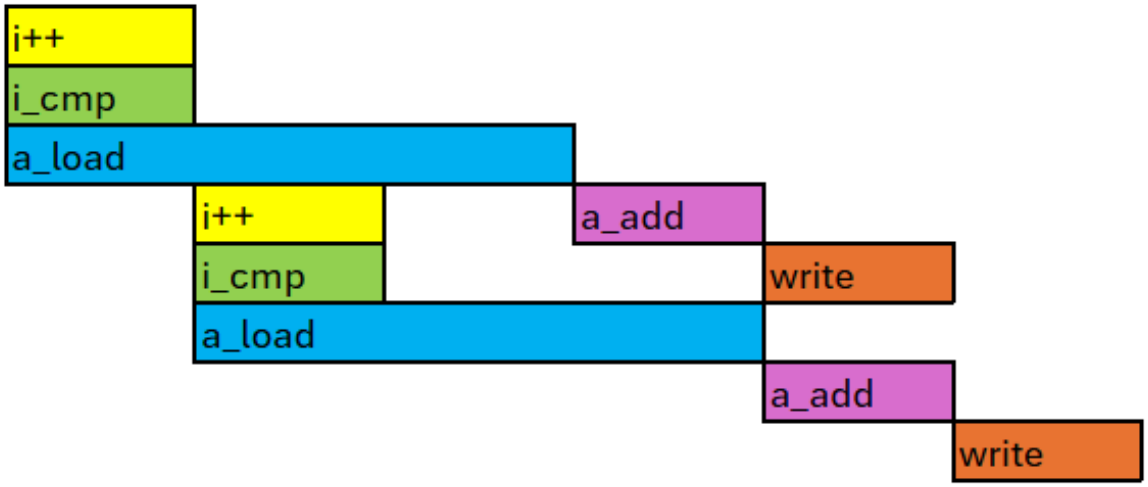

Figure 3: Scheduling of Kernel 1 with optimized code and directives, II=1

Figure 1 shows the scheduling of Kernel 1 with pipelining turned off. For Figure 2, we removed any instructions on directives and let vivado decide for itself. As seen in the chart, vivado does not employ any pipelining by itself, and the scheduling remains the same. It is important to note that

Vivado 2019 has been used for these tests.
For Figure 3 we finally apply the directives for static scheduling and achieve an iteration interval of 1 with an iteration latency of 2. In this case, we use pipelining with the following code:

```
#include "kernel1.h"

void kernel1( int a[ARRAY_SIZE] )
{
    int i;
    LOOP1: for(i=0; i<ARRAY_SIZE; i++) {
    #pragma HLS PIPELINE II = 1
        a[i] = a[i] * 5;
    }
}
```

Taking a look at the usage of resources and putting it all together we get the following table:

Kernel 1			
	FFs	LUTs	Latency
Baseline	38	116	4097
No Directives	38	116	4097
With Directived	29	128	4097

Kernel 2

Again, like in Kernel 1, as seen in Figure 4 and Figure 5, Vivado 2019 does not apply any directives by itself. Because of the long and many loads, we sit at an iteration latency of 5.

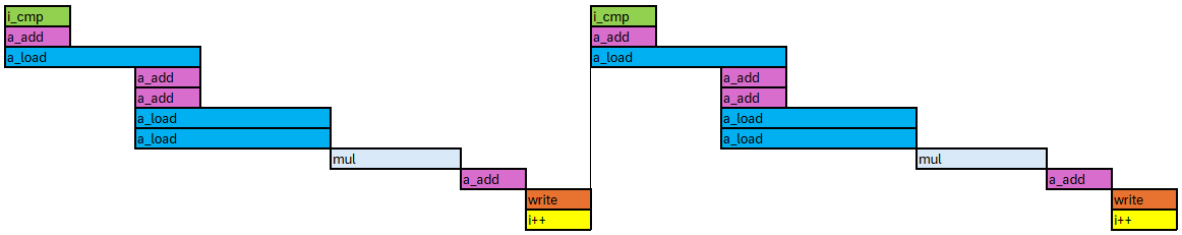

Figure 4: Scheduling of Kernel 2 with directives turned off

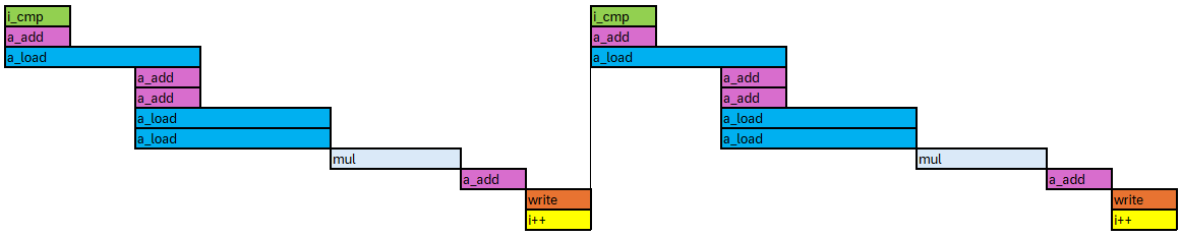

Figure 5: Scheduling of Kernel 2 without any input on directives

By simply applying pipelining, we can achieve an interation interval of 3. The loop-carried dependency of the memory loads hinders further improvement. However we can do better. We can break this dependency by keeping the last three values of the array $a[]$ in registers instead of always reading from memory. Now each iteration depends on local registers. This brings the iteration interval down to 1, reduces latency and iteration latency. This comes at the slight drawback of a bigger overhead

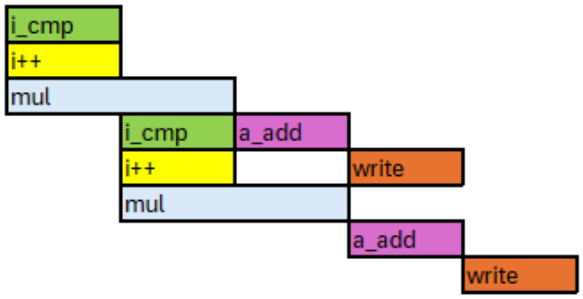

Figure 6: Scheduling of Kernel 2 with optimized code and directives, II=1

from the first loads of the array and a bigger area, since more resources are now needed to create these local registers. Figure 6 shows the new schedule.

This improvement was done with the following code:

```
#include "kernel2.h"

void kernel2( int a[ARRAY_SIZE] )
{
    int i;
    int a0 = a[0];
    int a1 = a[1];
    int a2 = a[2];
    int tmp;
    LOOP1: for(i=3; i<ARRAY_SIZE; i++) {
    #pragma HLS PIPELINE II = 1
        tmp = a2 + a1 * a0;
        a[i] = tmp;
        a0 = a1;
        a1 = a2;
        a2 = tmp;
    }
}
```

Taking a look at the usage of resources and putting it all together we get the following table:

Table 1: resources kernel 2

Kernel 2					
	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	3	146	226	10226	5
No Directives	3	146	226	10226	5
With Directives	3	192	254	2045	2

Kernel 3

Figure 7 shows the kernel 3 schedule with all directives turned off. Again, there is no difference from turning all the directives off and letting Vivado 2019 decide what directives to use. From now on, if there is no difference, I will not mention or show the kernel schedule in the case of not giving any input on the directives, since it is identical to turning all directives off.

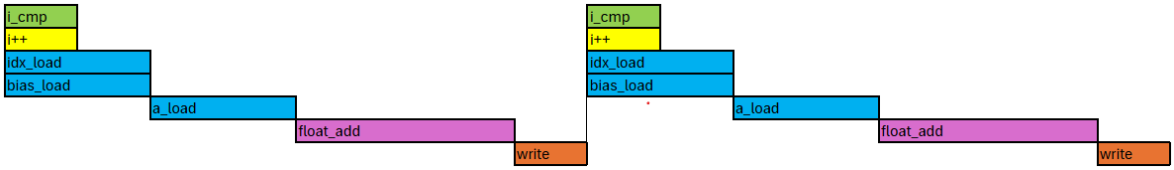

Figure 7: Scheduling of Kernel 3 with directives turned off

This code is a lot more problematic to optimize, and we need more knowledge of the arrays used. The iteration latency is 8, which is caused not only by the many loads but also by the adder, which, in this case, needs to add floats instead of integers. One could decrease the iteration latency by, if the array allows it, removing the float requirement. To decrease the interation interval, the most we are allowed to do is to use pipelining and achieve an iteration interval of 7. The address coming from $index[i]$ is unpredictable and so we always have to wait out the operation and cannot start to pipeline sooner. If we could guarantee no aliasing between the different $index[i]$, we could use the pragma HLS DEPENDENCE to remove the memory dependecne. But since we cannot guarantee this right now, an iteration interval of 7 is the best we can do statically.

```
#include "kernel3.h"

void kernel3( float a[ARRAY_SIZE], float bias[ARRAY_SIZE], int index[ARRAY_SIZE])
{
    LOOP1: for (int i=0; i<ARRAY_SIZE; ++i) {
    #pragma HLS PIPELINE
        a[index[i]] = a[index[i]]+ bias[i];
      }
}
```

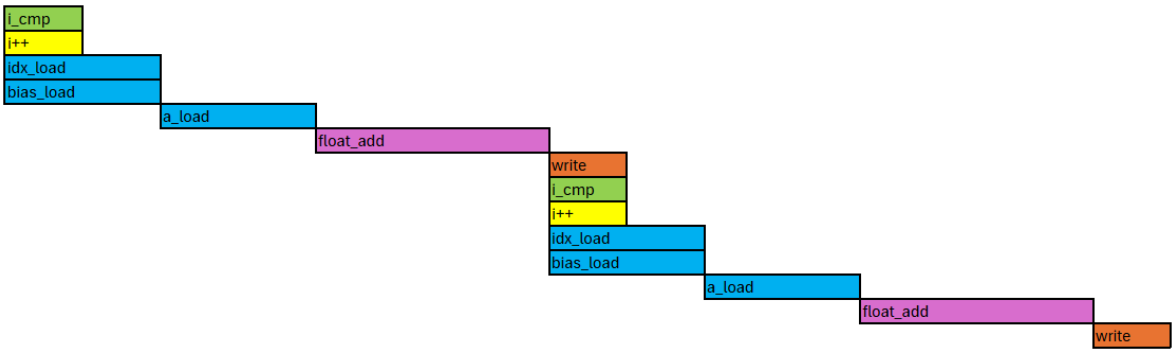

Figure 8: Scheduling of Kernel 3 with optimized code and directives, II=7

Taking a look at Table 2 we do see more usage in resources but less latency.

If we were to accept a massive overhead, we could bring the II down by using dynamic scheduling. By letting loads and stores issue speculatively at run time and checking for any true conflicts in hardware, one could effectively achieve and II of 1. For this, it is best to use LSQ. This allows us to add $a[index[i]]$ and $bias[i]$ and store the result immediately while the LSQ is keeping track of these loads and stores and only serializes two iterations if they really do conflict because of the $index[]$.

Table 2: Resources Kernel 3

Kernel 3					
	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	2	367	317	16384	8
No Directives	2	367	317	16384	8
With Directives	2	370	341	14336	8

Kernel 4

Figure 9 shows the schedule for kernel 4. At first glance, it looks like simply enabling pipelining would bring down the iteration interval to 1, but it only goes down to 4. The $a[targetIndex]$ forces a stall per iteration because Vivado conservatively serializes a load-after-store to the same address, even though it is just a single element.

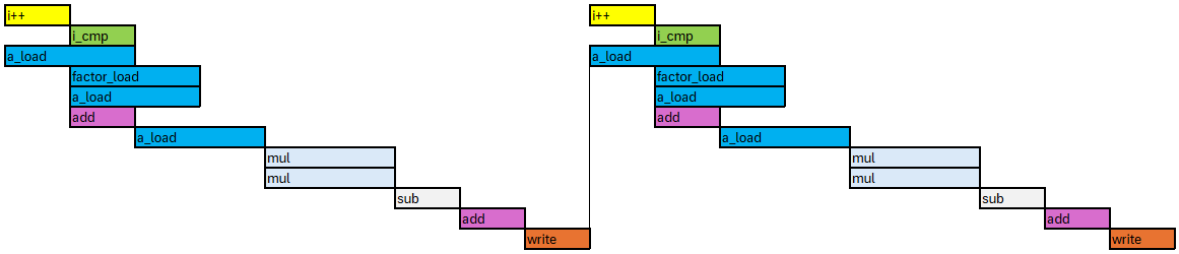

Figure 9: Scheduling of Kernel 4 with directives turned off

We can achieve a better II by forwarding this element of $a[targetIndex]$ to a scalar register in the loop that Vivado can then forward from stage to stage in the pipeline and achieve an II of 1. The resulting schedule is visible in Figure 10. This can be done with little change to the code:

```
#include "kernel4.h"

void kernel4(int a[ARRAY_SIZE], int factor[ARRAY_SIZE], int targetIndex)
{
        int tmp = a[targetIndex];
    LOOP1: for (int i=targetIndex+1; i<ARRAY_SIZE-1; ++i){
    #pragma HLS PIPELINE
        tmp = tmp - factor[i]*(a[i] + a[i+1]);
    }
    a[targetIndex] = tmp;
}
```

Using this method increases the resource usage and overhead but greately reduces the II and the iteration latency. We can also reduce the resources by rewriteing the line in the loop to contain one less multiplication. This can be seen in Table 3. One

Table 3: Resources Kernel 4

Kernel 4					
	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	6	273	279	2-10237	5
No Directives	6	273	279	2-10237	5
With Directives	3	180	309	3-2052	3

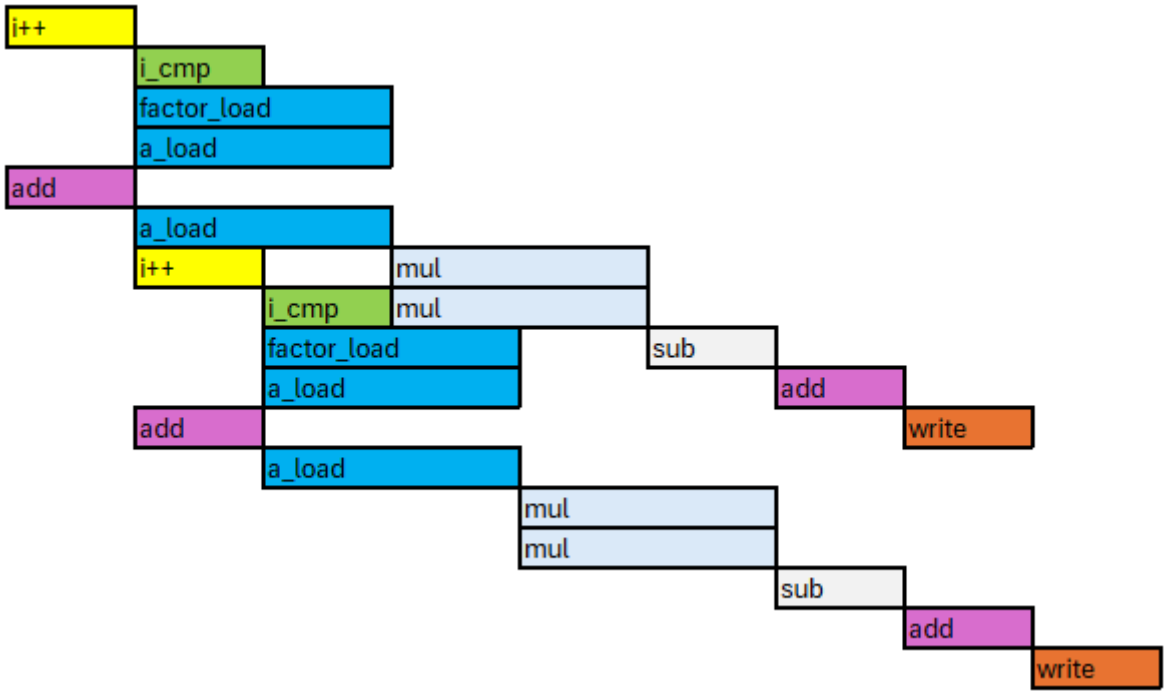

Figure 10: Scheduling of Kernel 4 with optimized code and directives, II=1

Kernel 5

Figure 11 shows the schedule for kernel 5.

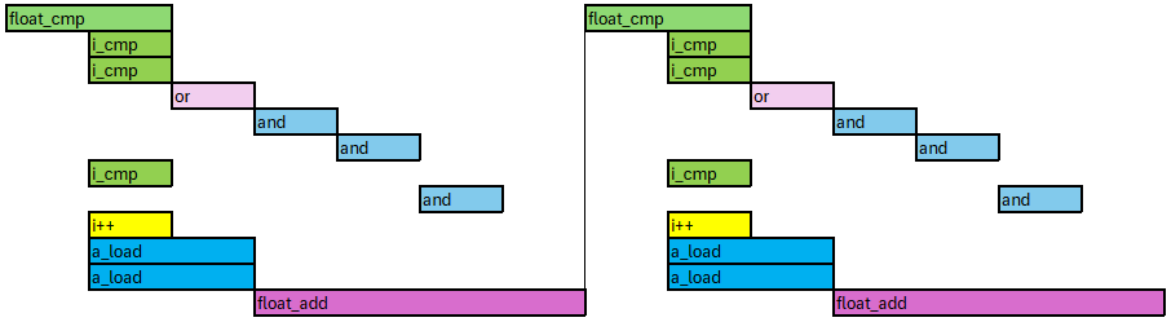

Figure 11: Scheduling of Kernel 5 with directives turned off

Here, simply using directives like pipelining only gives us an II of 7. We can first try and rewrite the while loop as a for loop and move the exit condition to a branch:

```
if (sum < max) {
  sum = array1[i] + array2[i];
} else {
  return sum;
}
```

However we still have an II of 7, since the compiler can not launch another iteration before checking if $sum < max$ is true. We introduce another mux by using a pure data-path select:

```
#include "kernel5.h"

float kernel5 (float max,
```

```
                float array1[ARRAY_SIZE], float
array2[ARRAY_SIZE]) {

    float sum = 0.0;
    LOOP1: for(int i = 0; i < ARRAY_SIZE; i++){
                #pragma HLS PIPELINE
            sum = sum < max ? array1[i] + array2[i] : sum;
    }
    return sum;
 }
```

This way the compare becomes just another mux in our pipeline and Vivado can treat it as a purely combinational select and we achieve an II of 1.

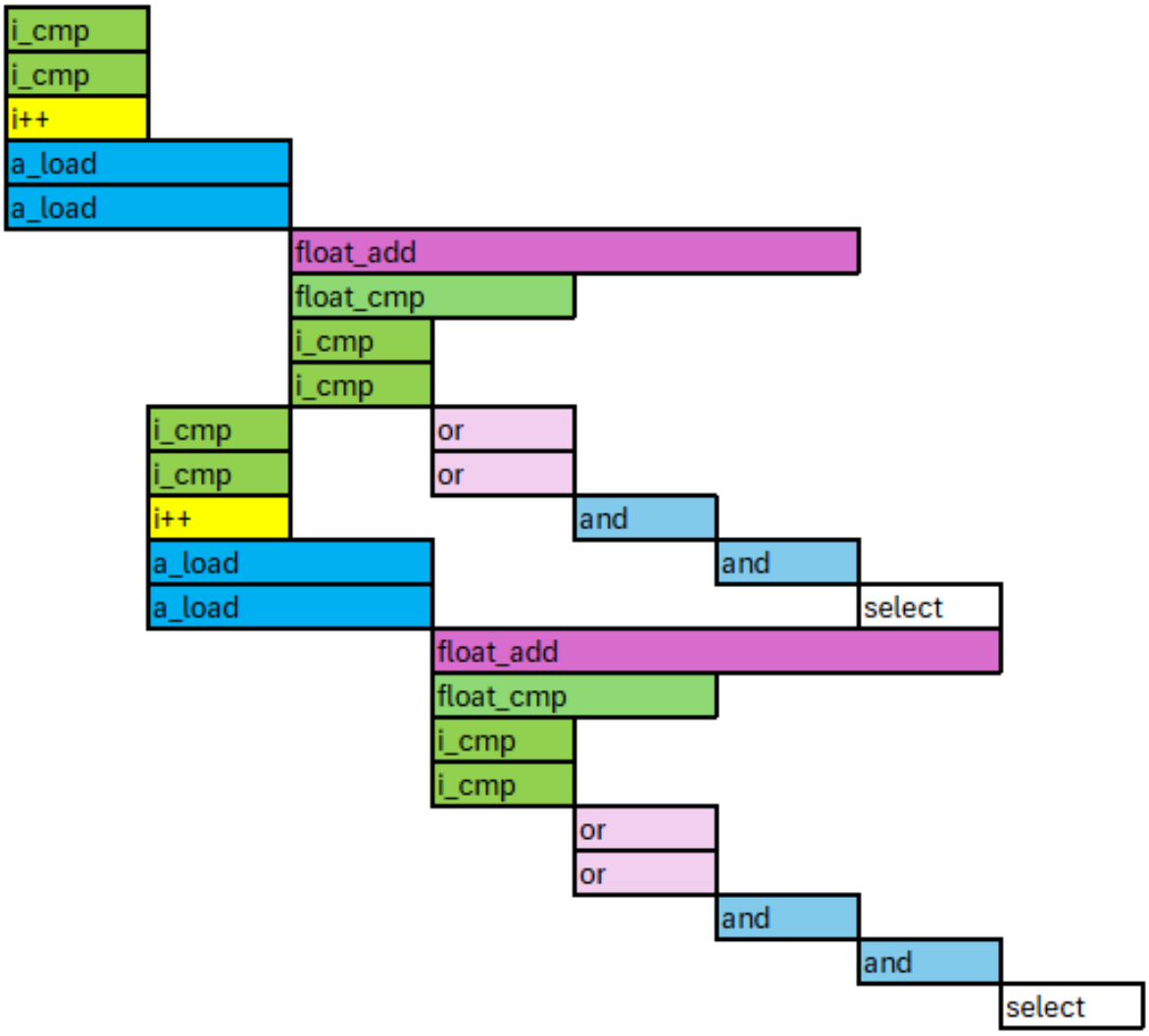

Figure 12: Scheduling of Kernel 5 with optimized code and directives, II=1

Table 4 shows the resource usage.

Kernel 6

Figure 13 shows the schedule of kernel 6.

Table 4: Resources Kernel 5

Kernel 5					
	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	2	462	478	2-14338	7
No Directives	2	462	478	3-14339	7
With Directives	2	606	568	2055	7

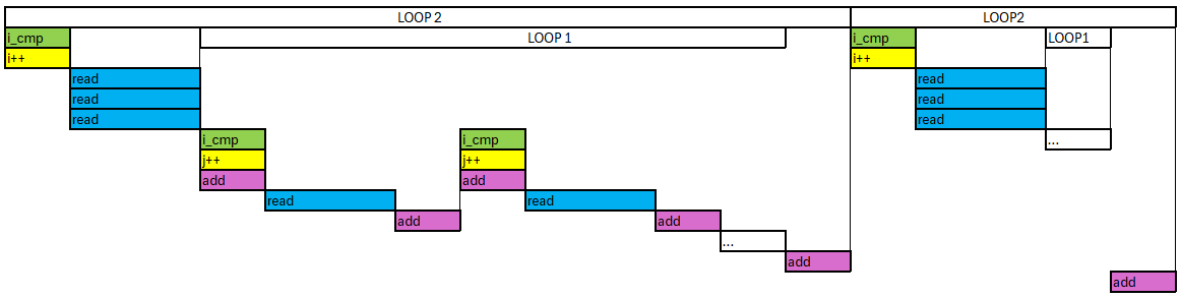

Figure 13: Scheduling of Kernel 6 with directives turned off

Using pipelining we can decrease the II of the inner loop to 1. However, the outer loop sees no such improvement. There is a read-after-write inter-iteration dependency that is caused by the *results*[*dst*]. Since we cannot guarantee that this final write might be shared by different iterations, we are forced to always wait it out. The other directives have no use in this kernel.

```
#include "kernel6.h"

void kernel6(int region_start[ARRAY_SIZE], int region_len[ARRAY_SIZE],
    int index_dst[ARRAY_SIZE], int results[2*ARRAY_SIZE]) {
        LOOP1: for (int i = 0; i < ARRAY_SIZE; i++) {
        #pragma HLS PIPELINE
            int start = region_start[i];  // Starting index of region
            int len   = region_len[i];    // Length of region
            int dst   = index_dst[i]; // Where to store the sum

            int sum = 0;
            LOOP2: for (int j = 0; j < len; j++) {
                #pragma HLS loop_tripcount min=0 max=2048
            #pragma HLS loop_flatten
                #pragma HLS PIPELINE
                sum += results[start + j];
            }
            results[dst] = sum;
        }
}
```

This is fixable by applying dynamic scheduling. Deploying LSQ will reduce the II for the same reasons as in Kernel 3.
Table 5 shows the resource usage.

Kernel 7

Figure 15 shows the schedule of kernel 7. Here, an inter-loop dependency hinders us from improving the II with pipelinging. We could try to unroll the loop by a factor of 7 and then use pipelining.

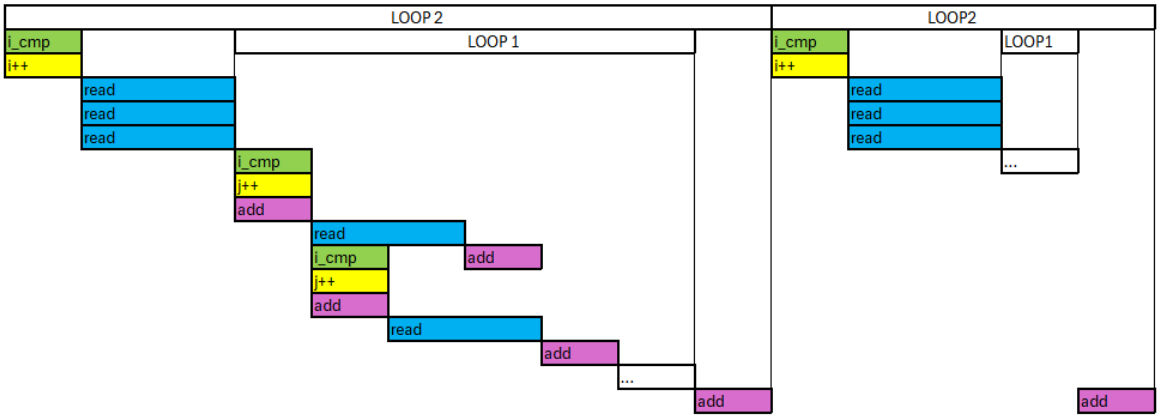

Figure 14: Scheduling of Kernel 6 with optimized code and directives, II=1 INNER LOOP

Table 5: Resources Kernel 6

Kernel 6					
	DPS	FFs	LUTs	Latency	Iteration Latency OUTER/INNER
Baseline	0	219	241	6145-839473	3-4099/2
No Directives	0	219	241	6145-839473	3-4099/2
With Directives	0	191	268	8193-4202497	4-2052/2

However, doing so blows up the resource usage and iteration latency. So instead of simply unrolling and pipelining, we could split the task into 2 loops by implementing a partial sum. LOOP 1 can be pipelined with an iteration interval of 1, and LOOP 2, since it is not too big, can be unrolled completely, thereby removing the loop completely. This causes an overhead but we can now achieven an II of 1. The implementation can be seen here:

```
#include "kernel7.h"

float kernel7(float array1[ARRAY_SIZE], float array2[ARRAY_SIZE])
{
        float ps[8] = {0.0f};
        float sum = 0.0f;

        LOOP1: for (int i = 0; i < ARRAY_SIZE; ++i) {
                #pragma HLS PIPELINE II=1

                float diff = array1[i] - array2[i];
                if (diff > 0.0f) {
                                int idx = i & (7);
                                ps[idx] += diff;
                }
        }

        LOOP2: for (int j = 0; j < 8; ++j) {
                #pragma HLS UNROLL
                sum += ps[j];
        }
return sum;
}
```

Figure 16 shows the schedule of the improved kernel.

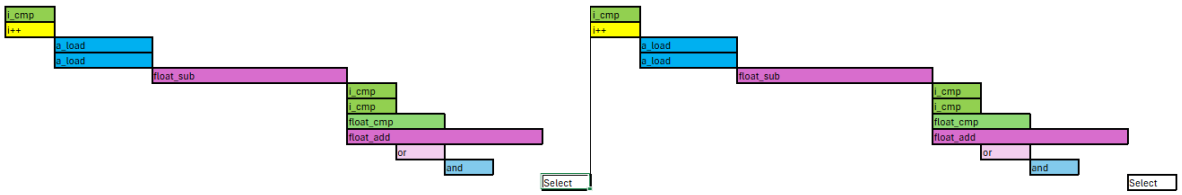

Figure 15: Scheduling of Kernel 7 with directives turned off

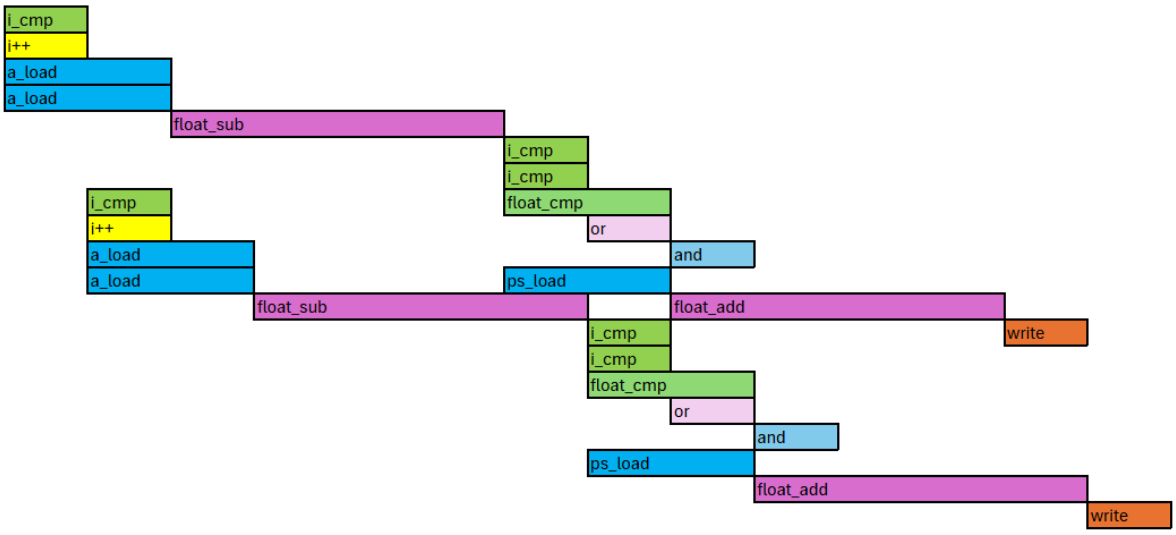

Figure 16: Scheduling of Kernel 7 with optimized code and directives, II=1

The resources have still risen and the iterarion latency has increased. But witht this, we have achieven an II of 1. Table 6 shows the change in resources.

Table 6: Resources Kernel 7

Kernel 7						
	BRAM	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	0	2	459	501	20480	10
No Directives	0	2	459	501	20480	10
With Directives	2	4	1009	1070		2103

Kernel 8

Figure 17 shows the schedule for kernel 8. The compiler must assume RAW memory dependence on a across iterations. We cannot guarantee that $a[i] and a[i-6+offset]$ never have any aliasing. So that means that pipelinging is not any help to us. On the contrary. It simply increases resource usage with no improvement in II.

One could try to implement some sort of shift-register and remove the loop carried dependency with the goal of decreasing the II. Doing this, we can achieve an II of 2. However, as seen in Table 7, this drastically increases the resource usage. Therefore, it is best to leave kernel 8 as is in baseline.

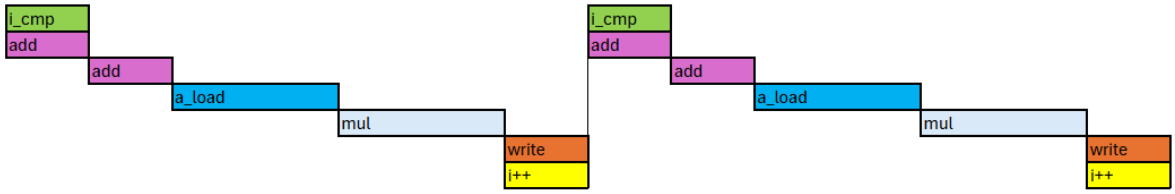

Figure 17: Scheduling of Kernel 8 with directives turned off

```
#include "kernel8.h"

// This is the code that tries to implement the mentioned shift-register

void kernel8(int a[ARRAY_SIZE], int factor, int offset)
{
    int r0 = a[0 + offset];
    int r1 = a[1 + offset];
    int r2 = a[2 + offset];
    int r3 = a[3 + offset];
    int r4 = a[4 + offset];
    int r5 = a[5 + offset];

  LOOP1: for (int i = 6; i < ARRAY_SIZE-1-offset; ++i) {
        #pragma HLS loop_tripcount min=0 max=2048
        #pragma HLS PIPELINE II=1

        int in_val = r0;
        r0 = r1;
        r1 = r2;
        r2 = r3;
        r3 = r4;
        r4 = r5;

        r5 = a[i-6+offset];
        a[i] = in_val * factor;
    }
}
```

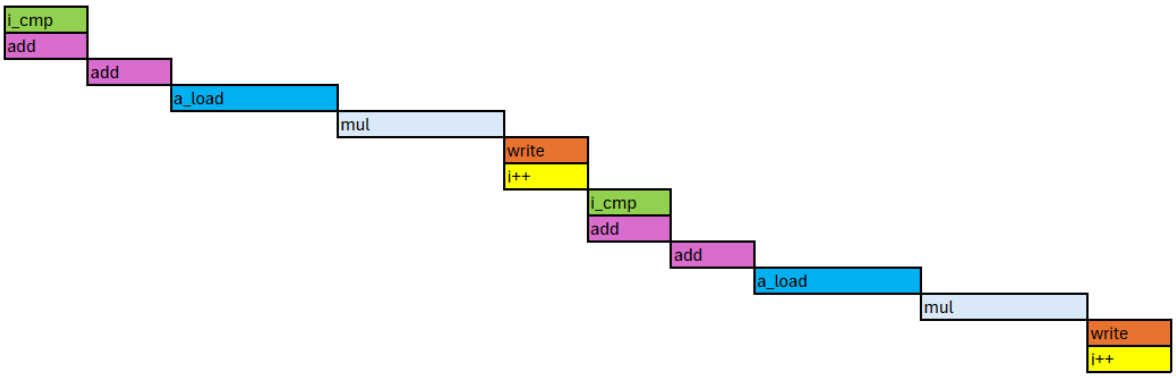

Figure 18: Scheduling of Kernel 8 with optimized code and directives, II=4

Table 7: Resources Kernel 8

Kernel 8					
	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	3	132	250	1-8193	4
No Directives	3	132	250	1-8193	4
Shift Register	3	422	546	0-4096	2
With Directives	3	133	255	2-8194	4

Kernel 9

Figure 19 shows the schedule for kernel 9. This kernel cannot be pipelined due to the read-after-write inter loop dependency. So simply using pipelingin we are left at an II of 6.

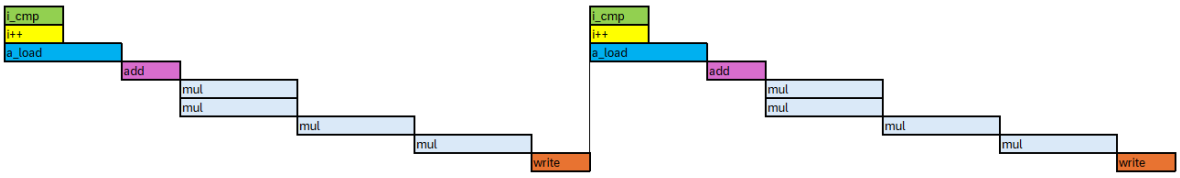

Figure 19: Scheduling of Kernel 9 with directives turned off

We can try to decouple the computing stage from the writing stage and create two seperate loops. The computed values are then stored in BRAM. Rach pipelined loop now only ever perfoems one mempry access per cycle to a single-ported RAM thus removing the loop-carried memory dependence with oth loops reaching an II of 1.

```
#include "kernel9.h"

void kernel9(int a[16*ARRAY_SIZE]){
int x = 0;
int idxa[ARRAY_SIZE];
int val[ARRAY_SIZE];
LOOP1: for (int i = 0; i < ARRAY_SIZE; ++i) {
        #pragma HLS PIPELINE II=1

        x += a[i];
        int idx = i * x * x * x * x;

        idxa[i] = idx;
        val[i] = x;
    }

LOOP2: for (int i = 0; i < ARRAY_SIZE; ++i) {
        #pragma HLS PIPELINE II=1

        a[idxa[i]] = val[i];
    }
}
```

We use two new BRAM slots and more resources in general, but we have decreased the latency by a large amount and brought the II down to 1. See Table 8.

Kernel 10

Figure 21 shows the schedule of kernel 10.

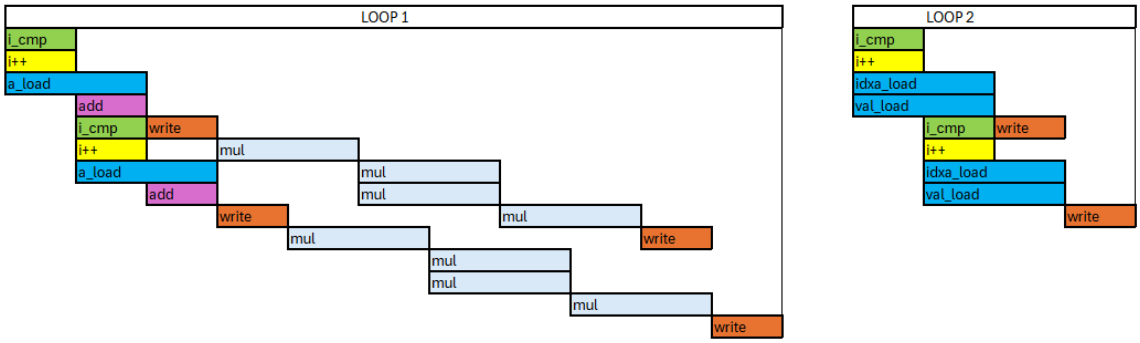

Figure 20: Scheduling of Kernel 9 with optimized code and directives, II=1

Table 8: Resources Kernel 9

Kernel 9						
	BRAM	DPS	FFs	LUTs	Latency	Iteration Latency
Baseline	0	11	223	225	769	6
No Directives	0	11	223	225	769	6
With Directives	2	11	463	455	264	6(LOOP1)/2(LOOP2)

With simple pipelining we can bring the iteration interval of the inner loop down to 1 (See Figure 22). This does not work for the outer loop. To pipeline, one must know exactly how many cycles the body takes. However, the inner loop's exit condition depends on $m[i]$ and $m[i]$ is not a compile-time constant. Also, pipelining, in cases of nested loops, works best for the innermost loops, since there is not a single simple back-edge for the outer loop's pipeline to target. One could faltten the inner loop but the resource usage would skyrocket, making this option less appealing.

```
#include "kernel10.h"

void kernel10(int a[ARRAY_SIZE], int b[ARRAY_SIZE], int m[ARRAY_SIZE])
{
    LOOP1: for (int i=0; i < ARRAY_SIZE; i++) {
    #pragma HLS PIPELINE off
        #pragma HLS loop_tripcount min=0 max=2048
            int s = 0;
            LOOP2: for (int j=0; j < m[i]; j++) {
            #pragma HLS PIPELINE
            #pragma HLS loop_flatten off
                #pragma HLS loop_tripcount min=0 max=2048
                s += a[j];
            }
            b[i] = s;
    }
}
```

Table 9 shows the resource usage.
One way for further improvements would be to use dynamic scheduling, specifically LSQ. At runtime, you could let $s+ = a[j]$ pipeline and the $b[i] = s$ store, overlapping different outer iteations. This would bring the II down to 1.

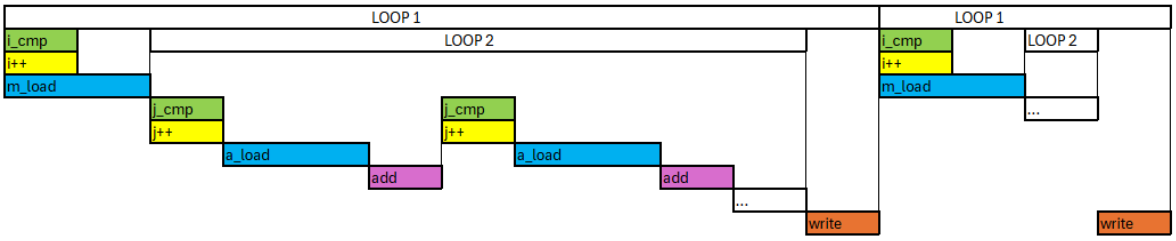

Figure 21: Scheduling of Kernel 10 with directives turned off

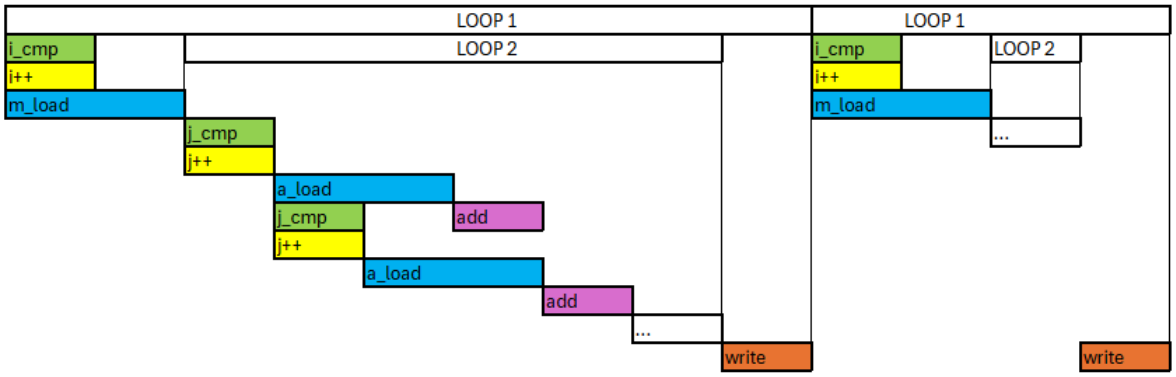

Figure 22: Scheduling of Kernel 10 with optimized code and directives, II=1 INNER LOOP

Table 9: Resources Kernel 10

Kernel 10					
	DPS	FFs	LUTs	Latency	Iteration Latency INNER/OUTER
Baseline	0	167	187	6144-8394752	2/3-4099
No Directives	0	167	187	6144-8394752	2/3-4099
With Directives	0	139	214	8192-4202496	2/4-2052