

Kernel 1

II = 1

```
#define PARTITION_FACTOR 8
void kernel1( int array[ARRAY_SIZE] )
{
#pragma HLS ARRAY_PARTITION variable=array type=cyclic
factor=PARTITION_FACTOR
    int i;
    for(i=0; i<ARRAY_SIZE; i++)
    {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=PARTITION_FACTOR
        array[i] = array[i] * 5;
    }
}
```

In the loop, we can pipeline the array loads and stores by reading the array value in cycle 0 and performing the arithmetic and store operation in cycle 1. Notice that Vitis has optimized the multiplication by 5 into a shift and add.

For further optimization, we partition the array by PARTITION_FACTOR and unrolling the loop by the same factor. This enables a speed up of PARTITION_FACTOR at the expense of having PARTITION_FACTOR more hardware. Developers can choose their desired level of PARTITION_FACTOR to tradeoff extra hardware for latency.

Total Kernel Latency = 130
Of which loop latency = 128

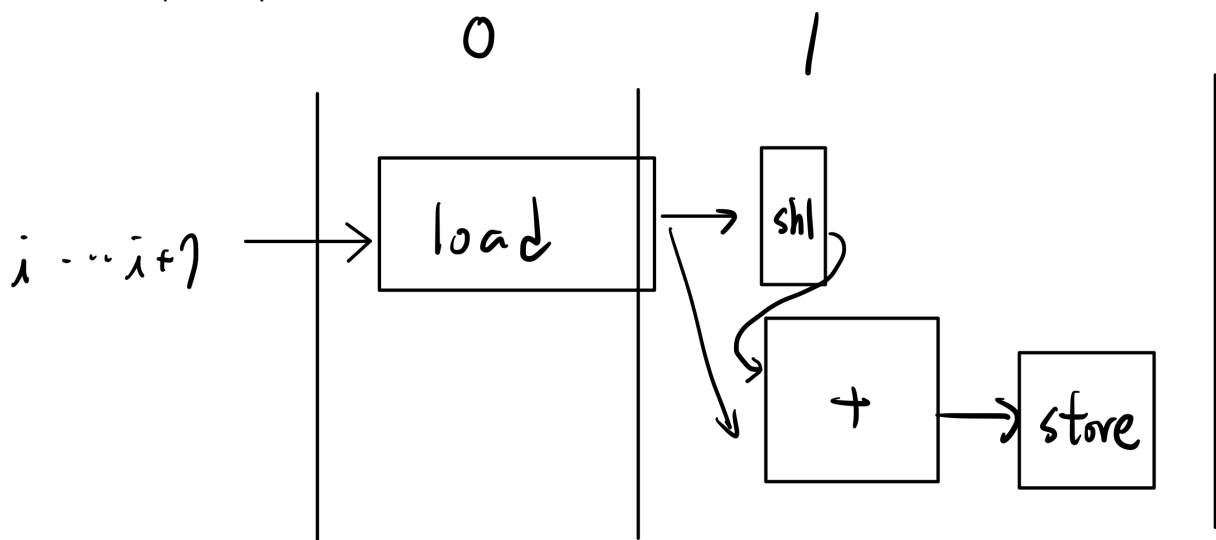


Figure 1: Dataflow graph of the optimized kernel 1.

Kernel 2

II = 1

```
void kernel2( int array[ARRAY_SIZE] )
{
    int i;
    int a = array[0], b = array[1], c = array[2];
    int tmp = a * b, res;
    for(i=3; i<ARRAY_SIZE; i++) {
#pragma HLS PIPELINE
        res = c + tmp;
        array[i] = res;
        tmp = b * c;
        b = c;
        c = res;
    }
}
```

For the first optimization, we can observe that array[0], array[1], and array[2] uniquely determine the rest of the array. Therefore, there is no need to read from array inside the loop, so the optimized code stores every variable used in the computation with a register (a, b, and c).

For the second optimization, from the scheduling diagram, we can see that the multiply option takes the most time within a cycle. Instead of performing a multiply then an addition for each array index, we can compute the multiplication in the previous cycle. This leads to the current pipelined version, where in each cycle, addition is done for index i and multiplication is done for index $i+1$.

Total Kernel latency = 1027 cycles
Of which loop latency = 1023 cycles

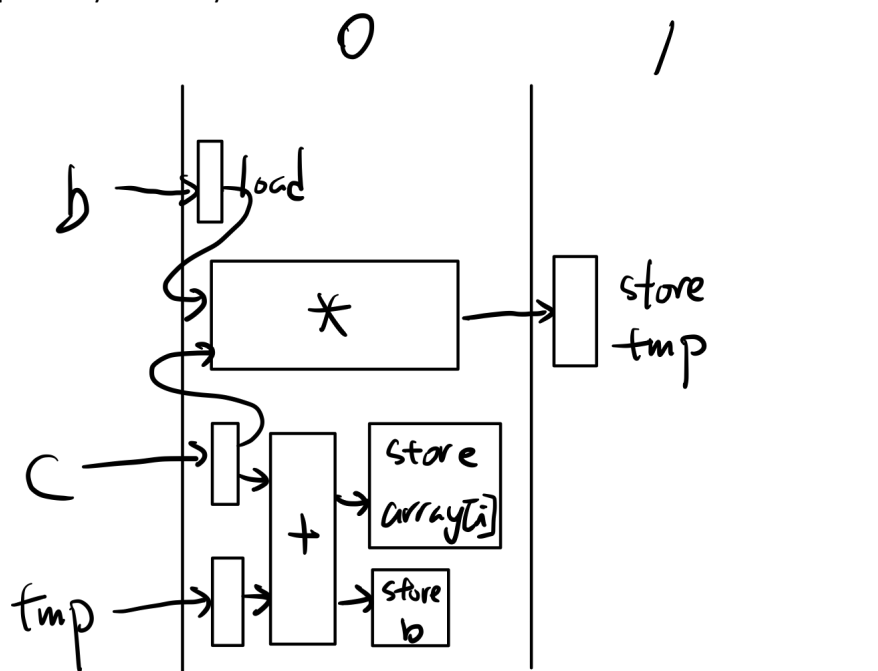


Figure 2: Dataflow graph of the optimized kernel 2.

Kernel 3

II = 4

```
void kernel3(float hist[ARRAY_SIZE], float weight[ARRAY_SIZE], int
index[ARRAY_SIZE])
{
    for (int i=0; i<ARRAY_SIZE; ++i) {
#pragma HLS PIPELINE II=4
        hist[index[i]] = hist[index[i]] + weight[i];
    }
}
```

In this kernel, there's not much we can do because each time we need to read the previous value of `hist[index[i]]` before adding `weight[i]` to it. Vitis makes use of some kind of forwarding path so as soon as we have the `fadd` result, a new loop iteration can be started.

Total Kernel latency = 4102 cycles
Of which loop latency = 4100 cycles

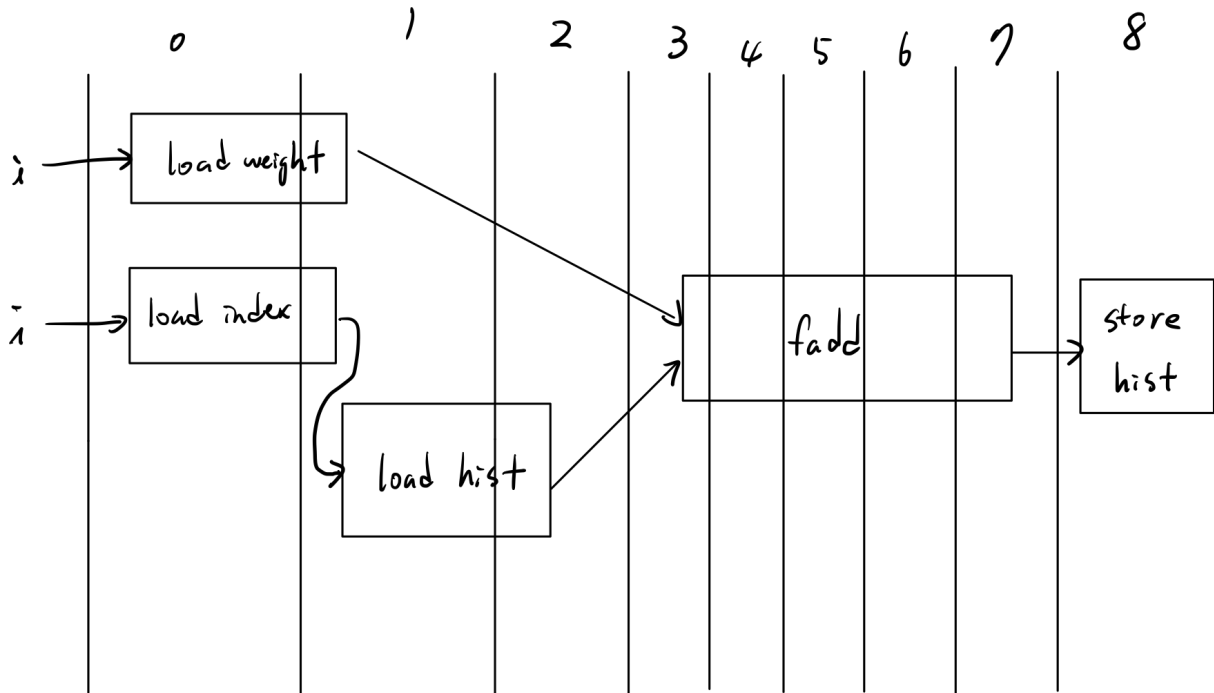


Figure 3: Dataflow graph of the optimized kernel 3.

Our failed attempt to pipeline `index[i]`

We tried to pipeline `index[i]` by reading the value in a previous cycle, but the total kernel latency went up (4106) even though the loop latency went down (4096). This is perhaps because in the following program, the post-loop code increases kernel latency.

```
void kernel3(float hist[ARRAY_SIZE], float weight[ARRAY_SIZE], int
index[ARRAY_SIZE])
{
    int idx0 = index[0], idx1;
```

```
    for (int i=0; i<ARRAY_SIZE-1; ++i) {  
#pragma HLS PIPELINE II=4  
        idx1 = index[i+1];  
        hist[idx0] = hist[idx0] + weight[i];  
        idx0 = idx1;  
    }  
    hist[idx0] = hist[idx0] + weight[ARRAY_SIZE-1];  
}
```

Kernel 4

II = 1

```
void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
    int tmp = 0;
    for (int i=offset+1; i<ARRAY_SIZE-1; ++i)
    {
#pragma HLS PIPELINE
        tmp = tmp + index[i] * (array[i+1] - array[i]);
    }
    array[offset] = array[offset] + tmp;
}
```

From the original code, we notice that `offset` is a loop invariant, meaning that its value does not change between different iterations of the loop. Therefore, instead of writing to `array[offset]`, we can simply write to a register `tmp`. With this optimization, it is possible to obtain an II = 1.

Since the number of iterations depends on `offset`, we can manually compute the approximate cycles needed for the loop as $\text{num_iterations} + 3 = (\text{ARRAY_SIZE} - \text{offset} - 1) + 3 = \text{ARRAY_SIZE} - \text{offset} + 2$ cycles. The loop prologue and kernel prologue is 1 cycle, while the epilogue takes 2 more cycles. In total, we estimate the kernel latency to be about $\text{ARRAY_SIZE} - \text{offset} + 5$ cycles. This is close enough to the average latency (1020) provided by Vitis co-simulation.

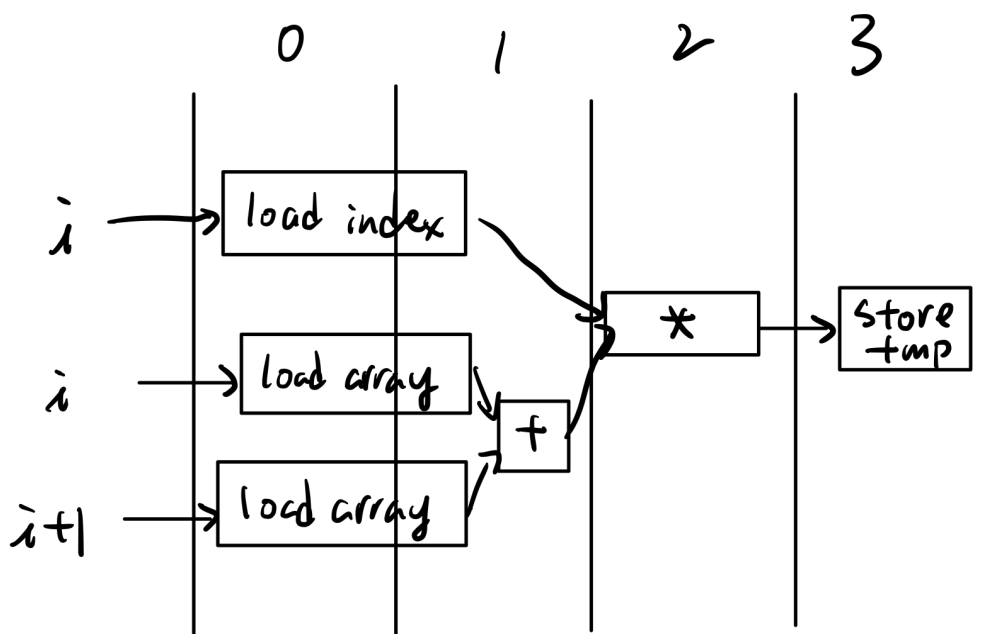


Figure 3: Dataflow graph of the optimized kernel 4.

Since there are no interloop dependencies, we can further reduce the cycle count by partitioning `array` and `index`. Doing so increases the loop interval to 6, but maintains II=1. The co-simulation tool reports an average kernel latency of 265, about a quarter of the latency of the previous version.

```
#define PARTITION_FACTOR 4
void kernel4(int array[ARRAY_SIZE], int index[ARRAY_SIZE], int offset)
{
#pragma HLS ARRAY_PARTITION variable=array type=cyclic
factor=PARTITION_FACTOR
#pragma HLS ARRAY_PARTITION variable=index type=cyclic
factor=PARTITION_FACTOR
    int tmp = 0;
    for (int i=offset+1; i<ARRAY_SIZE-1; ++i)
    {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=PARTITION_FACTOR
        tmp = tmp + index[i] * (array[i+1] - array[i]);
    }
    array[offset] = array[offset] + tmp;
}
```

Kernel 5

$$\mathbb{I}\mathbb{I} = 1$$

```
#define PARTITION_FACTOR 8

float kernel5(float bound, float a[ARRAY_SIZE], float b[ARRAY_SIZE])
{
    bool done = false;

#pragma HLS ARRAY_PARTITION variable=a type=cyclic factor=PARTITION_FACTOR
#pragma HLS ARRAY_PARTITION variable=b type=cyclic factor=PARTITION_FACTOR
    float sum;
    for (int i=0; i < ARRAY_SIZE; ++i) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=PARTITION_FACTOR
        if (done || (sum = a[i] + b[i]) ≥ bound) {
            done = true;
        }
    }
    return sum;
}
```

In this kernel, we refactored the code by removing the interloop dependencies of the break statement by comparing all sums with the bound. If any sum is larger or equal to the bound, the `done` flag is set to `true`, and the solution sum is set to the current sum. Since the boolean operator `||` is short-circuited in C, `sum` will not change when `done` is `true`. An `II` of 1 was reached since there were no dependencies between loops.

Moreover, the arrays are partitioned with a factor of 8, allowing for more parallelism. Without partitioning, Vitis reports the loop latency as 1031 cycles, with the kernel latency as 1033 cycles. With partitioning, the loop latency is reduced to 136 cycles, with the kernel latency as 138 cycles.

The main drawback of this optimization is that the latency is much higher if the return value is in the first few values. For example, in the original implementation, if $a[0] + b[0] \geq \text{bound}$, the latency would be about 8~10 cycles, while our optimization would return only after all sums are finished.

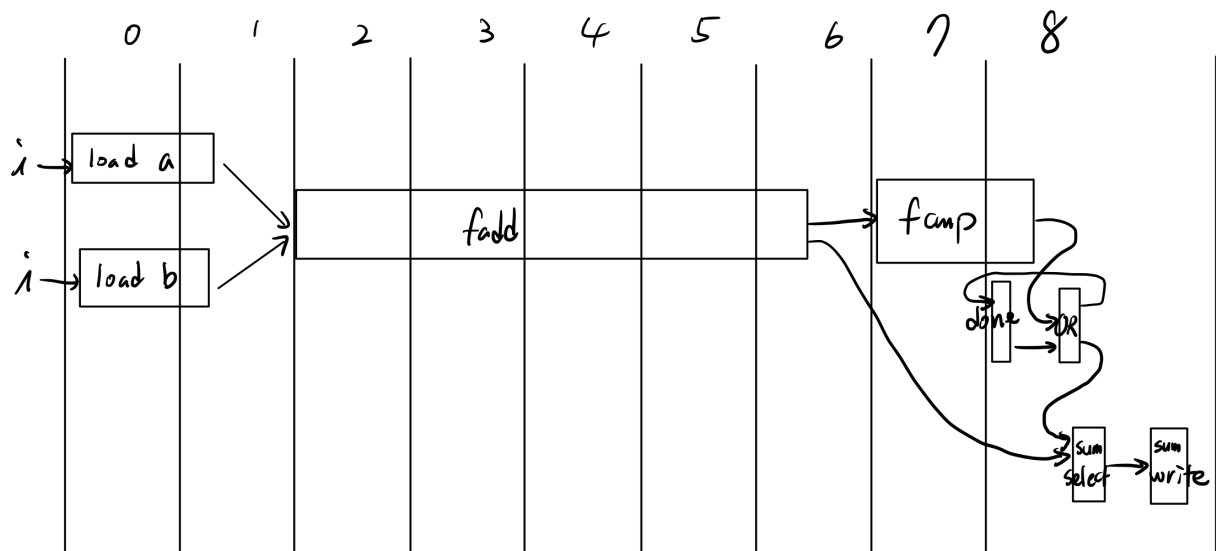


Figure 5: Dataflow graph of the optimized kernel 5 for PARTITION FACTOR = 1.