More Loop Optimizations

Kang Zhao

zhaokang@bupt.edu.cn

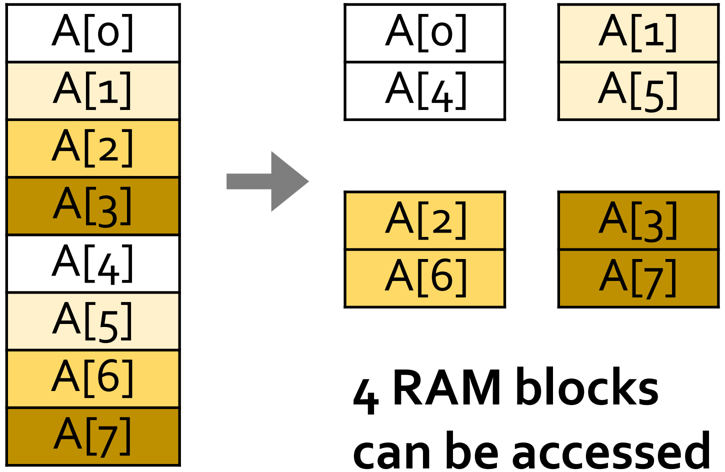
Hardware Specialization with HLS

- **Where does performance gain come from? Specialization!**
- Data type specialization
  - Arbitrary-precision fixed-point, custom floating-point
- Interface/communication specialization
  - Streaming, memory-mapped I/O, etc.
- **Memory specialization**
  - Array partitioning, data reuse, etc.
- **Compute specialization**
  - Unrolling, pipelining, dataflow, multithreading, etc.
- Architecture specialization
  - Pipelined, recursive, hybrid, etc.

The Three Musketeers
(i) Array partition
(ii) Loop unroll
(iii) Loop pipeline

Array Partition – Memory Parallelism

- **Initially, an array is mapped to one (or more) block(s) of RAM** (or BRAM on FPGA)
  - One block of RAM has at most **two ports**
  - At most **two** read/write operations can be done in **one clock cycle – Parallelism is 2 (too low)**
- **An array can be partitioned and mapped to multiple blocks of RAMs**

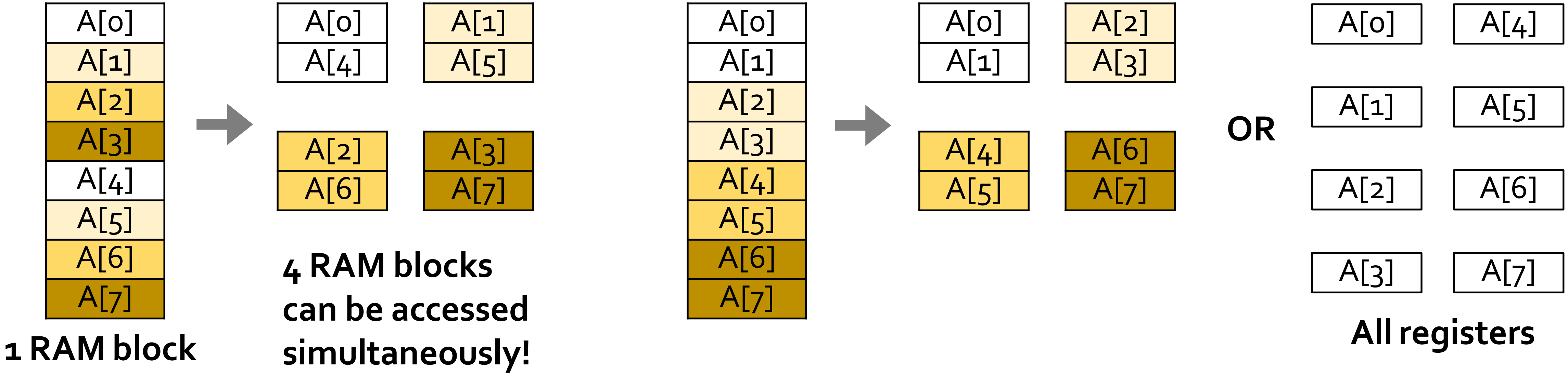


1 RAM block

4 RAM blocks can be accessed simultaneously!

Array Partition – Memory Parallelism

- **Initially, an array is mapped to one (or more) block(s) of RAM** (or BRAM on FPGA)
  - One block of RAM has at most **two ports**
  - At most **two** read/write operations can be done in **one clock cycle – Parallelism is 2 (too low)**
- **An array can be partitioned and mapped to multiple blocks of RAMs**
  - Can also be partitioned into individual elements and mapped to registers
    - Only if your array is small otherwise the tool will give up



Loop Unrolling

- **Loop unrolling to expose higher parallelism and achieve shorter latency**
  - Pros
    - Decrease loop overhead
    - Increase parallelism for scheduling
  - Cons
    - Increase operation count, which may negatively impact area, power, and timing

Original Loop

```
for (int i = 0; i < N; i++)
#pragma HLS unroll
  A[i] = B[i] + C[i];
```

N x m cycles

Assume A[i] = B[i] + C[i] takes m cycle

→

Unrolled Loop

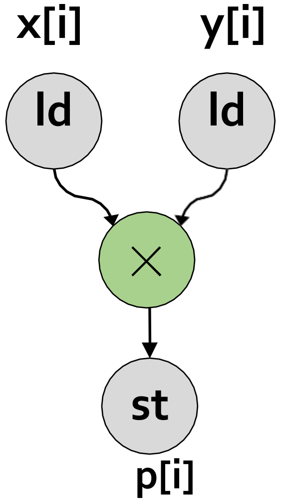
```
A[0] = B[0] + C[0];
A[1] = B[1] + C[1];
A[2] = B[2] + C[2];
...
```

m cycle

Only if A, B, and C are fully partitioned!

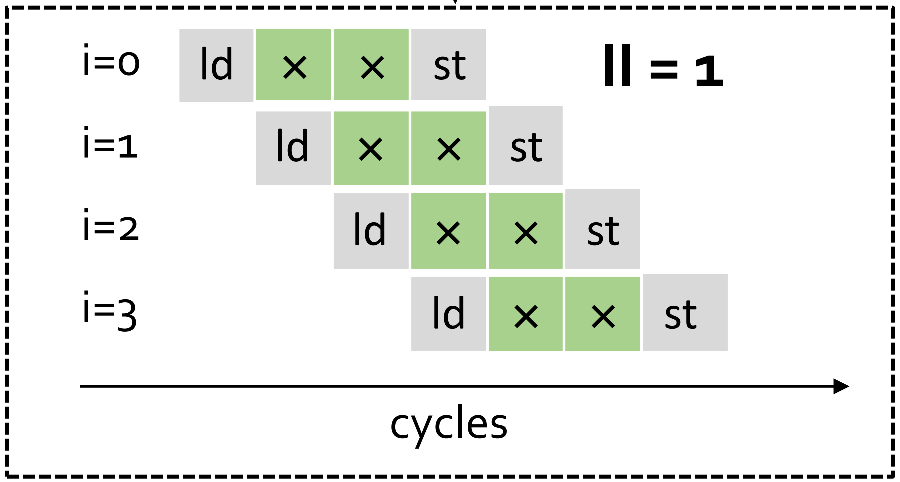
Loop Pipelining

- **Loop pipelining is one of the most important optimizations for high-level synthesis**
  - Allows a new iteration to begin processing before the previous iteration is complete
  - Key metric: **Initiation Interval (II)** in # cycles



ld – Load
st – Store

```
for (i = 0; i < N; ++i)
#pragma HLS pipeline
    p[i] = x[i] * y[i];
```



Put-together: Pipeline + Unroll +Partition

- The three techniques are frequently used together to boost computation efficiency

```
for (int i = 0; i < N; i++) {

  for (int j = 0; j < M; j++) {

    A[i][j] = B[i][j] * C[i][j];
  }
}
```

A

RAM block 1

A[0][0]	A[0][1]	…	A[0][N-1]
A[1][0]	A[1][1]	…	A[1][N-1]
…	…	…	…
A[M-1][0]	A[M-1][1]	…	A[M-1][N-1]

B

RAM block 2

B[0][0]			

C

RAM block 3

C[0][0]			

Put-together: Pipeline + Unroll +Partition

- **The three techniques are frequently used together to boost computation efficiency**

```
for (int i = 0; i < N; i++) {

  for (int j = 0; j < M; j++) {
#pragma HLS unroll
    A[i][j] = B[i][j] * C[i][j];
  }
}
```

Compute in parallel

RAM block 1

A[0][0]	A[0][1]	...	A[0][N-1]
A[1][0]	A[1][1]	...	A[1][N-1]
...	...	...	...
A[M-1][0]	A[M-1][1]	...	A[M-1][N-1]

Compute in parallel

RAM block 2

B[0][0]			

Compute in parallel

RAM block 3

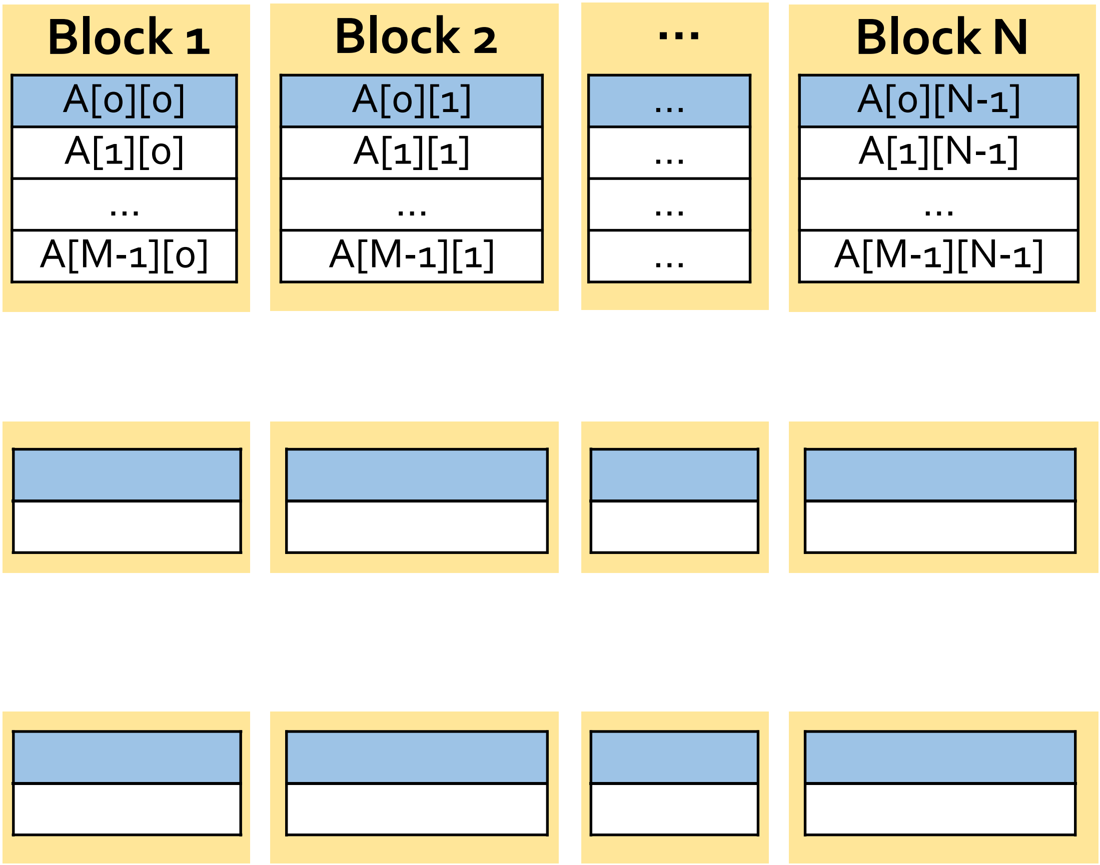
C[0][0]			

Memory ports limited by 2 → Need to partition

Put-together: Pipeline + Unroll +Partition

- **The three techniques are frequently used together to boost computation efficiency**

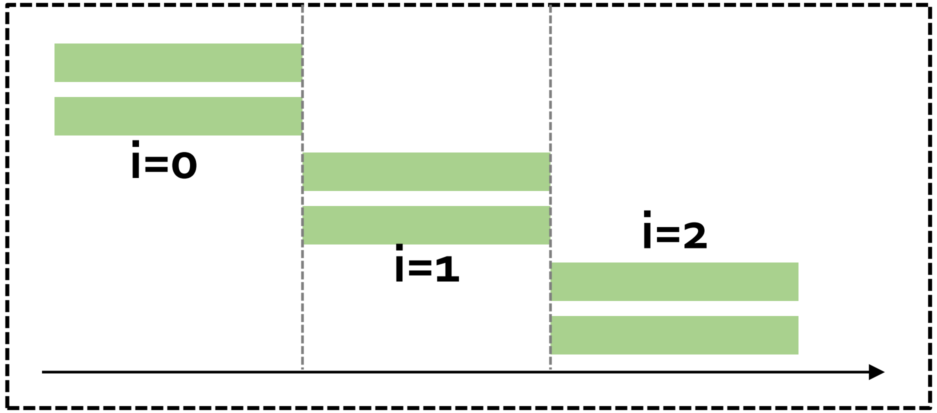
```
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2
complete #pragma HLS array_partition variable=C
dim=2 complete

for (int i = 0; i < N; i++) {

  for (int j = 0; j < M; j++) {
#pragma HLS unroll
    A[i][j] = B[i][j] * C[i][j];
  }
}
```



Put-together: Pipeline + Unroll +Partition

- **The three techniques are frequently used together to boost computation efficiency**

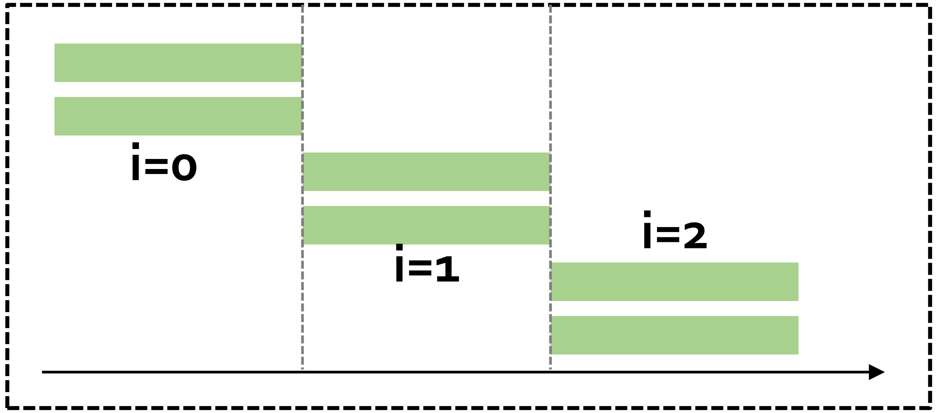
```
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete
#pragma HLS array_partition variable=C dim=2 complete

for (int i = 0; i < N; i++) {

  for (int j = 0; j < M; j++) {
#pragma HLS unroll
    A[i][j] = B[i][j] * C[i][j];
  }
}
```

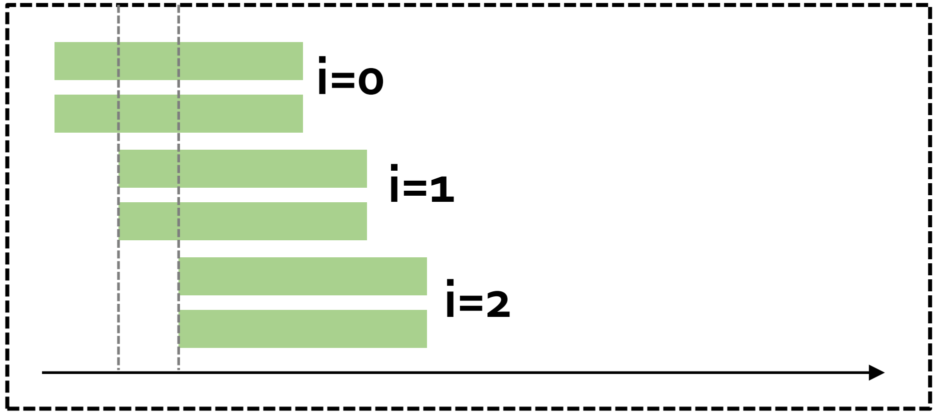


Put-together: Pipeline + Unroll +Partition

- **The three techniques are frequently used together to boost computation efficiency**

```
#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2
complete #pragma HLS array_partition variable=C
dim=2 complete

for (int i = 0; i < N; i++) {
#pragma HLS pipeline II=1
  for (int j = 0; j < M; j++) {
#pragma HLS unroll
    A[i][j] = B[i][j] * C[i][j];
  }
}
```



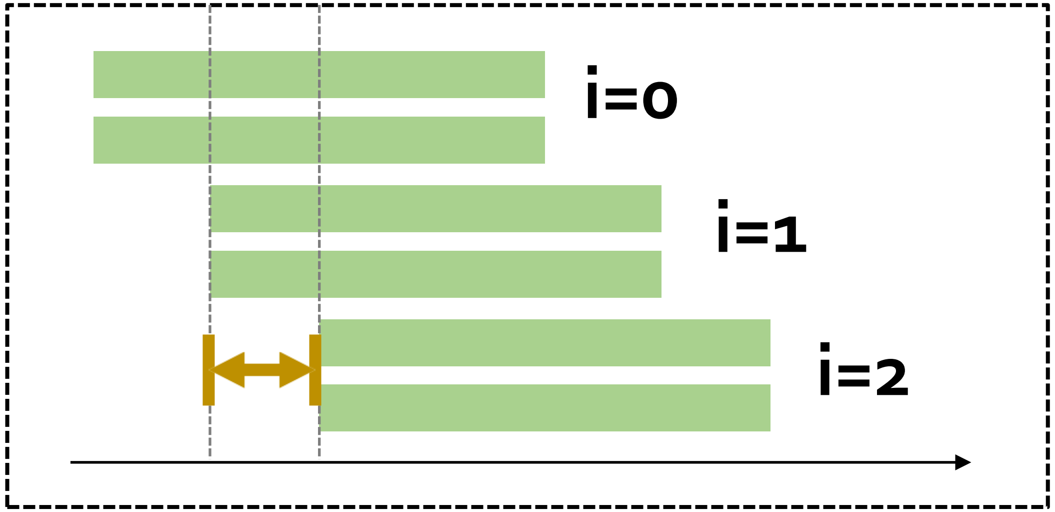


More Loop Optimizations

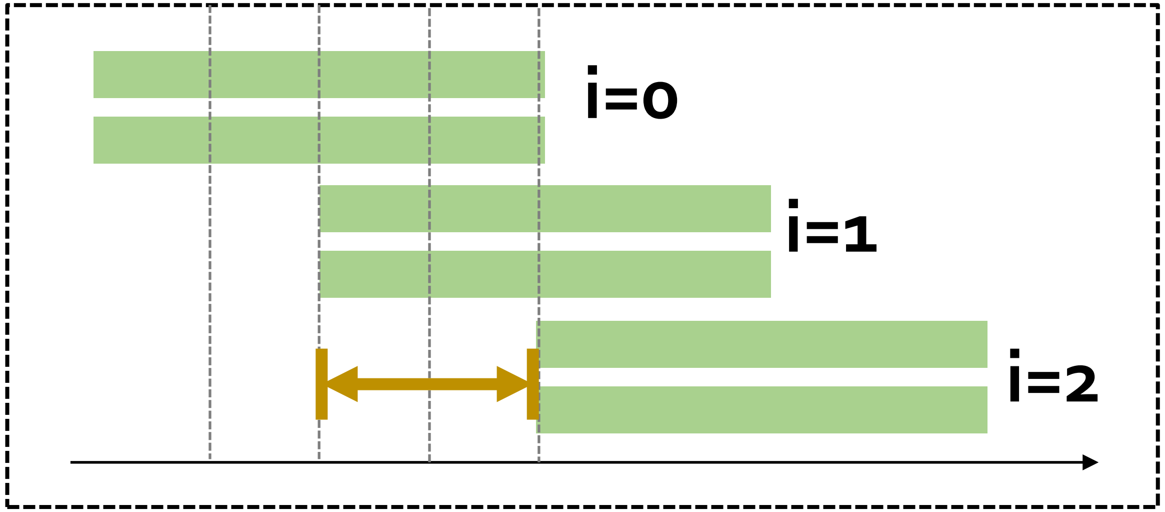
- Initial Interval (II) Violation
- **Pipeline and Unroll over Functions**
- **Loop-carried Dependency**
  - Loop reorder
  - Remove false dependency
- **Loop Tiling**
- **Loop Fusion**
  - Parallel loops fused into single loop
  - Wrap loops into functions

The most important factor about pipelining

- **Initial Interval (II) – II violation**



II = 1



II = 2

Causes to II violation

- **Not enough resource (e.g., memory not partitioned)**

```
void test(int sum[10], int A[10][100], int B[10][100])
{
    for(int i = 0; i < 10; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 100; j++) {
            sum[i] += A[i][j] * B[i][j];
        }
    }
}
```

- Pipeline effective region: “current” region with curly bracket
- All the loops within { } are automatically unrolled

Causes to II violation

- **Not enough resource (e.g., memory not partitioned)**

```
void test(int sum[10], int A[10][100], int B[10][100])
{
    for(int i = 0; i < 10; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 100; j++) {
            sum[i] += A[i][j] * B[i][j];
        }
    }
}
```

WARNING: [HLS 200-885] **Unable to schedule 'load' operation ('A_load_2', top.cpp:63) on array 'A' due to limited memory ports**. Please consider using a memory core with more ports or partitioning the array 'A'.
Resolution: For help on HLS 200-885 see ...
INFO: [HLS 200-1470] Pipelining result : **Target II = 1, Final II = 50, Depth = 52, ...**

Causes to II violation

- **Manual array partition**

```
void test(int sum[10], int A[10][100], int B[10][100])
{

#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete

    for(int i = 0; i < 10; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 100; j++) {
            sum[i] += A[i][j] * B[i][j];
        }
    }
}
```

Causes to II violation

- [Caution!] If array is from DRAM – copy to local buffer (BRAM) first!

```
void test(int sum[10], int A[10][100], int B[10][100])
{
#pragma HLS interface m_axi port=A offset=slave bundle=mem
#pragma HLS interface m_axi port=B offset=slave bundle=mem
#pragma HLS interface m_axi port=sum offset=slave bundle=mem

#pragma HLS array_partition variable=A dim=2 complete
#pragma HLS array_partition variable=B dim=2 complete

// copy A to A_local, B to B_local
    for(int i = 0; i < 10; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 100; j++) {
            sum_local[i] += A_local[i][j] * B_local[i][j];
}}}
```

A, B, and sum are from DRAM; cannot partition DRAM!

More Loop Optimizations

- **Initial Interval (II) Violation**
- **Pipeline and Unroll over Functions**
- **Loop-carried Dependency**
  - Loop reorder
  - Remove false dependency
- **Loop Tiling**
- **Loop Fusion**
  - Parallel loops fused into single loop
  - Wrap loops into functions

Pipeline and Unroll over Functions

- **Pipeline over function: the function must be inlined**
- **Unroll over function: the function will be duplicated**

Pipeline over Function

- **If a function inside pipeline region – the function must be inlined**

```
void test(int sum[10], int A[10][100], int B[10][100])
{

#pragma HLS array_partition variable=A dim=1 complete
#pragma HLS array_partition variable=B dim=1 complete

    for(int i = 0; i < 10; i++) {
#pragma HLS pipeline II=1
        sum[i] += foo(A[i], B[i]);
    }
}
```

```
int foo(int A[100], int B[100])
{
#pragma HLS inline

    int add = 0;
    for(int i = 0; i < 10; i++)
        add += A[i] * B[i];
    return add;
}
```

Unroll over Function

- **Unroll a loop involving a function – the function will be duplicated**

```
void test(int sum[10], int A[10][100], int B[10][100])
{

#pragma HLS array_partition variable=A dim=1 complete
#pragma HLS array_partition variable=B dim=1 complete
#pragma HLS array_partition variable=sum complete

    loop_L1: for(int i = 0; i < 10; i++) {
#pragma HLS unroll
        sum[i] = foo(A[i], B[i]);

    }
}
```

foo is duplicated by 10 copies which run in parallel

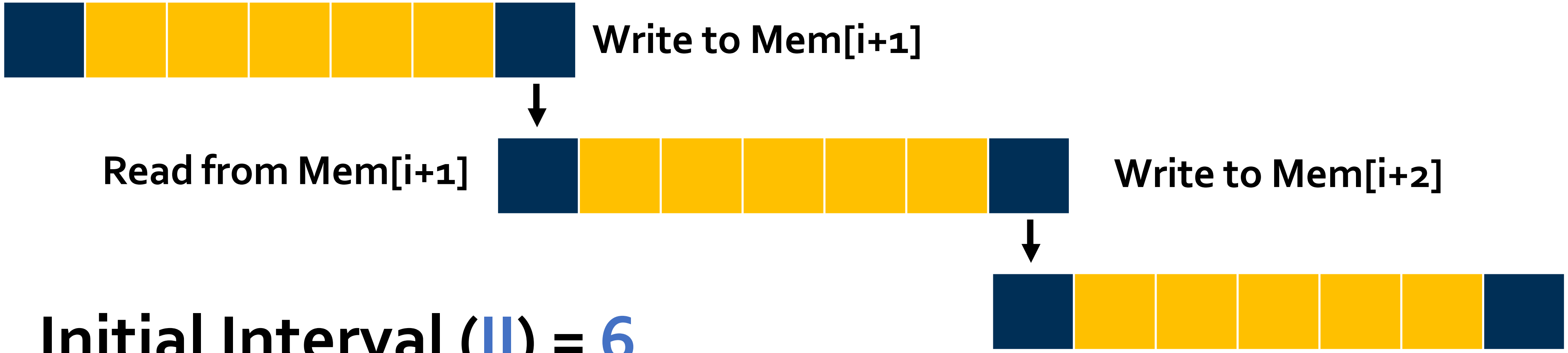
```
int foo(int A[100], int B[100])
{
    int add = 0;
    for(int i = 0; i < 10; i++)
        add += A[i] * B[i];
    return add;
}
```

More Loop Optimizations

- **Initial Interval (II) Violation**
- **Pipeline and Unroll over Functions**
- **Loop-carried Dependency**
  - Loop reorder
  - Remove false dependency
- **Loop Tiling**
- **Loop Fusion**
  - Parallel loops fused into single loop
  - Wrap loops into functions

Loop-carried Dependency

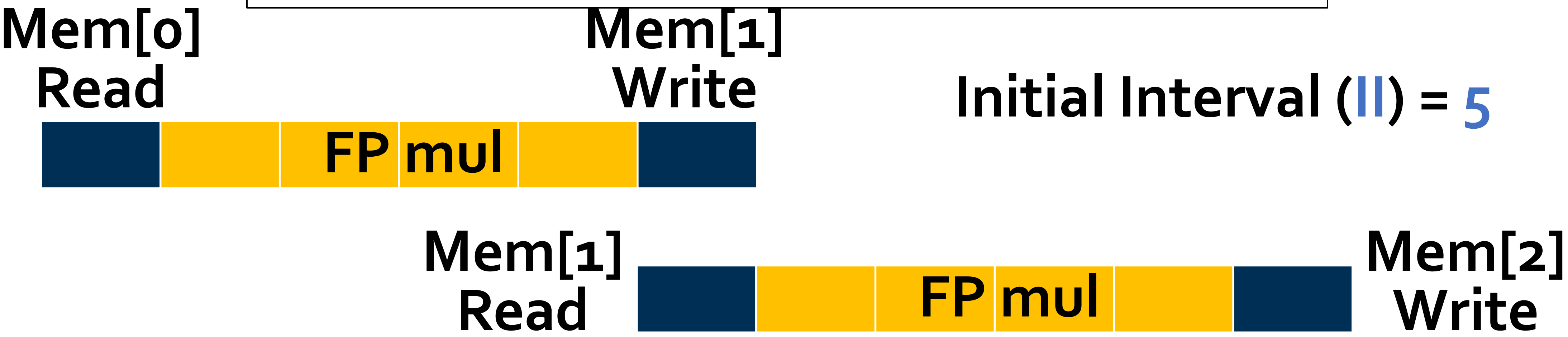
- **An iteration of a loop depends on a result produced by a previous iteration, which takes multiple cycles to complete**
- **Can be true or false dependency**
  - HLS tends to be conservative



Initial Interval (II) = 6

If True Dependency?

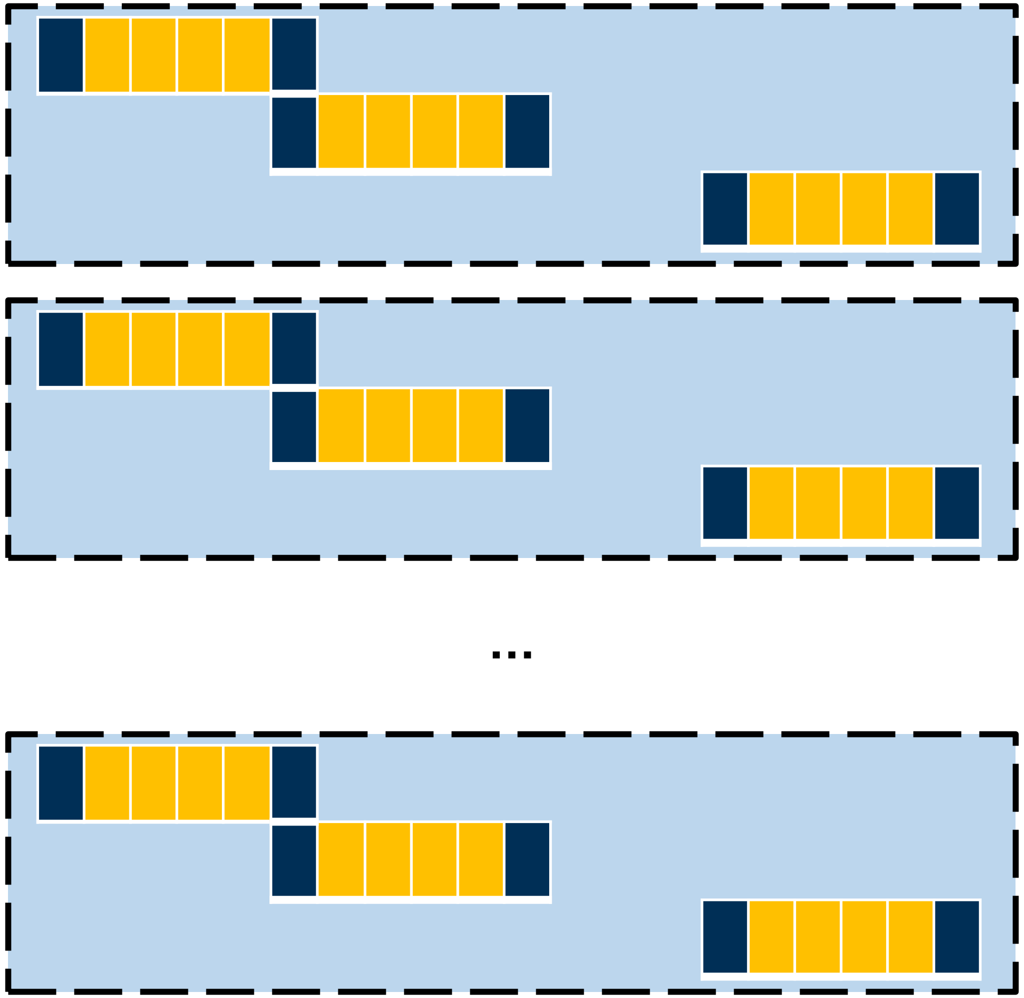
```
void test(float w, float mem[100])
{
   for(int i = 1; i < 100; i++) {
#pragma HLS pipeline
      float a = mem[i-1];
      mem[i] = a * w;
   }
}
```



Initial Interval (II) = 5

If True Dependency with another dimension…

```
void test(float w, float mem[10][100])
{

    for(int j = 0; j < 10; j++)
    {   for(int i = 1; i < 100; i++) {
        #pragma HLS pipeline
            float a = mem[j][i-1];
            mem[j][i] = a * w;
        }
    }
}
```



If True Dependency with another dimension…

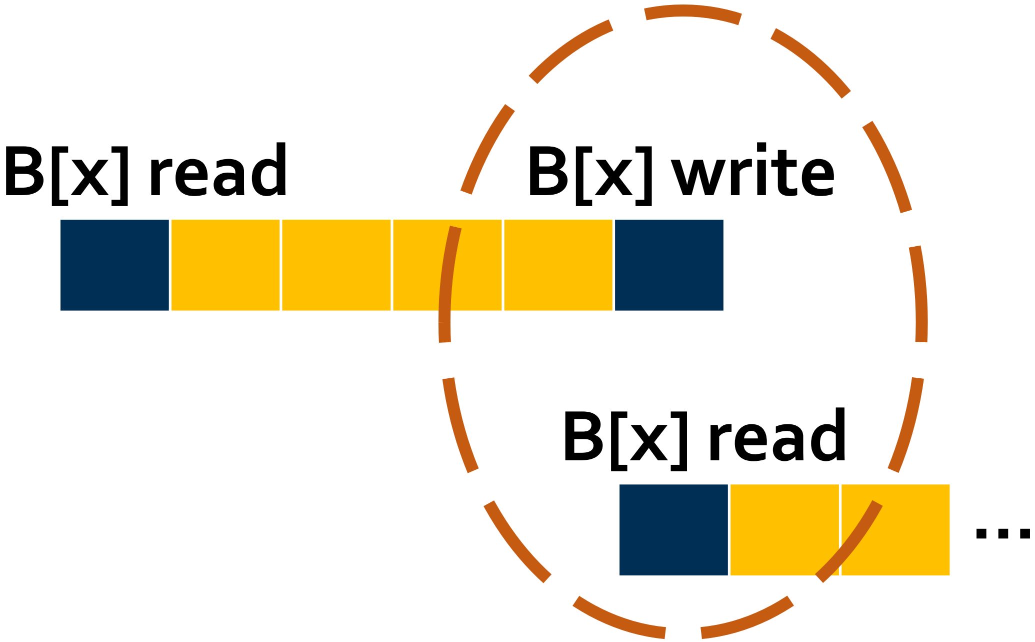
```
void test(float w, float mem[10][100])
{
#pragma HLS array_partition
variable=mem dim=1 complete

   for(int i = 1; i < 100; i++) {
#pragma HLS pipeline
      for(int j = 0; j < 10;    {
      j++)  float a = mem[j][i-1];
            mem[j][i] = a * w;
       }
     }
}
```

Reorder the two loops

If False Dependency? Remove it!

- **But only do it when you're sure it's safe**

```
void test(int* A, float B[100])
{

    for(int i = 0; i < 100; i++) {
        int x = A[i];
        B[x] *= B[x];
    }
}
```



Not sure if they're the same B[x]…

II = 3

If False Dependency? Remove it!

- **But only do it when you're sure it's safe**

```
void test(int* A, float B[100])
{
#pragma HLS DEPENDENCE variable=B inter false
    for(int i = 0; i < 100; i++) {
        int x = A[i];
        B[x] *= B[x];
    }
}
```

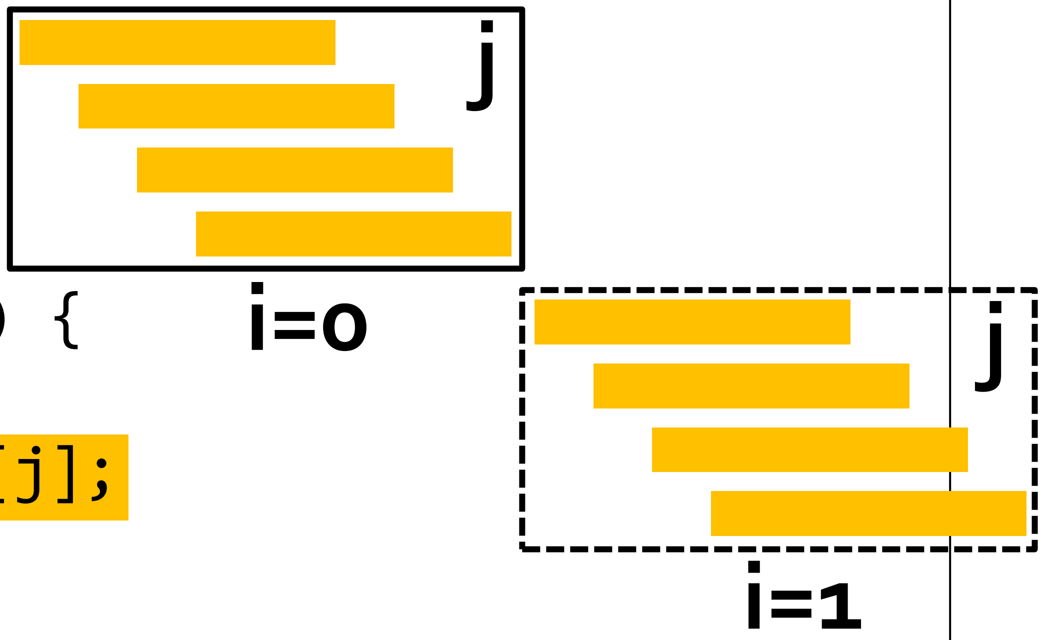
But if you know for sure: e.g., array A is monotonically increasing

II = 1

Effective Region of Unroll and Pipeline

- **Both Pipeline and Unroll must act on a loop, i.e., within a loop region**

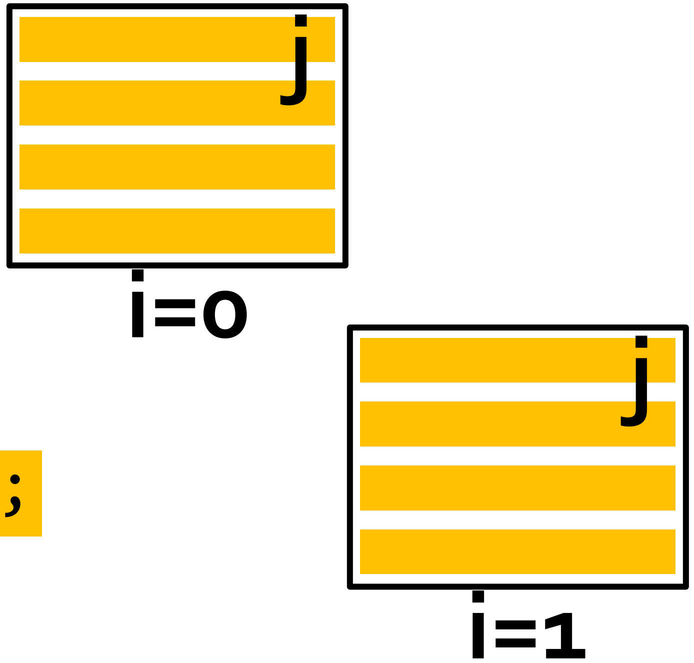
```
void top( ports ) {

    for(int i = 0; i < 1024; i++) {

        for(int j = 0; j < 1024; j++) {
#pragma HLS pipeline
            sum[i] += A[i][j] * B[i][j];
        }
    }

}
```



Effective Region of Unroll and Pipeline

- **Both Pipeline and Unroll must act on a loop, i.e., within a loop region**

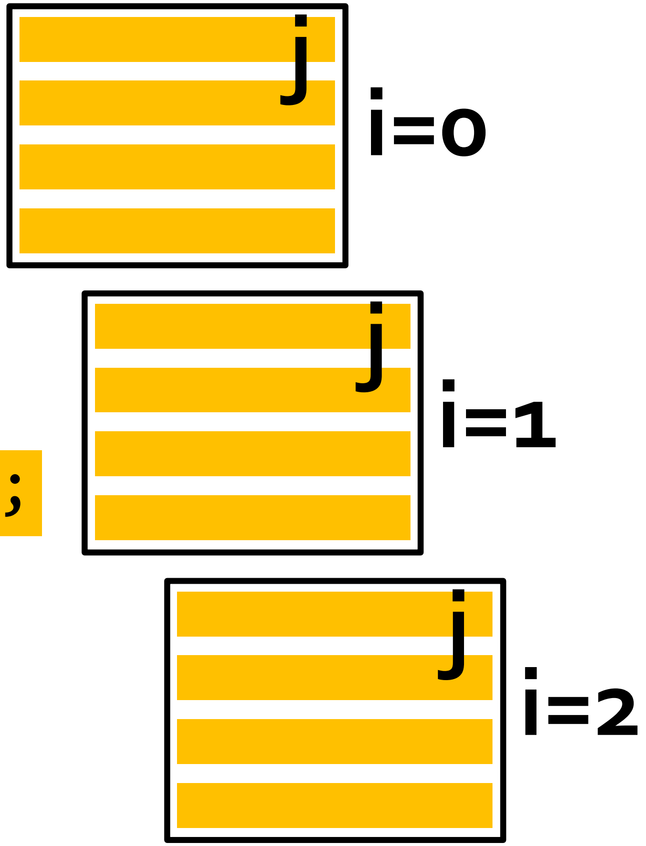
```
void top( ports ) {

    for(int i = 0; i < 1024; i++) {

          for(int j = 0; j < 1024; j++) {
#pragma HLS unroll
               sum[i] += A[i][j] * B[i][j];
          }
     }

}
```



Effective Region of Unroll and Pipeline

- **Both Pipeline and Unroll must act on a loop, i.e., within a loop region**

```
void top( ports ) {

   for(int i = 0; i < 1024; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 1024; j++) {
#pragma HLS unroll
            sum[i] += A[i][j] * B[i][j];
        }
    }
}
```



Effective Region of Unroll and Pipeline

- **Tried both pipeline and unroll together… NOT recommended**

```
void top( ports ) {

   for(int i = 0; i < 1024; i++) {
#pragma HLS pipeline
#pragma HLS unroll

       for(int j = 0; j < 1024; j++) {
           sum[i] += A[i][j] * B[i][j];
       }
    }
}
```

- The inner loops are all unrolled
- And the unrolled statements will be pipelined

- The inner loops are all unrolled
- But they won't paralyze unless wrapped as a function

More Loop Optimizations

- **Initial Interval (II) Violation**
- **Pipeline and Unroll over Functions**
- **Loop-carried Dependency**
  - Loop reorder
  - Remove false dependency
- **Loop Tiling**
- **Loop Fusion**
  - Parallel loops fused into single loop
  - Wrap loops into functions

Loop Tiling – Controlled Parallelism

- **When you have multiple loops, or the loop count is too big**

```
void test(int A[1024][1024], int B[1024][1024], int
sum[1024])
{
// array_partition omitted
    for(int i = 0; i < 1024; i++) {
#pragma HLS pipeline
        for(int j = 0; j < 1024; j++) {
            sum[i] += A[i][j] * B[i][j];
    }   }
}
```

Unable to pipeline, crashes, or hangs forever

Loop Tiling – Controlled Parallelism

- **Step 1: break them into tiles**
- **Step 2: parallelize within one tile**
- **Step 3: pipeline across different tiles**
  - We'll see another example of II violation

Step 1: Break into Tiles

```
void test(int A[1024][1024], int B[1024][1024], int
sum[1024])
{
    L1: for(int i = 0; i < 1024; i++) {
        L2: for(int j = 0; j < 1024; j += 16) {
            L3: for(int jj = 0; jj < 16; jj++) {
                sum[i] += A[i][j + jj] * B[i][j + jj];
            }

        }
    }
}
```

Let these 16 multiplications execute in parallel

Step 2: Parallelize with one tile

```
void test(int A[1024][1024], int B[1024][1024], int sum[1024])
{
#pragma HLS array_partition variable=A dim=2 factor=16 cyclic
#pragma HLS array_partition variable=B dim=2 factor=16 cyclic

    L1: for(int i = 0; i < 1024; i++) {
        L2: for(int j = 0; j < 1024; j += 16) {

            L3: for(int jj = 0; jj < 16; jj++) {
#pragma HLS unroll
                sum[i] += A[i][j + jj] * B[i][j + jj];
}}}}
```

Partition using factor and "cyclic" or "block"

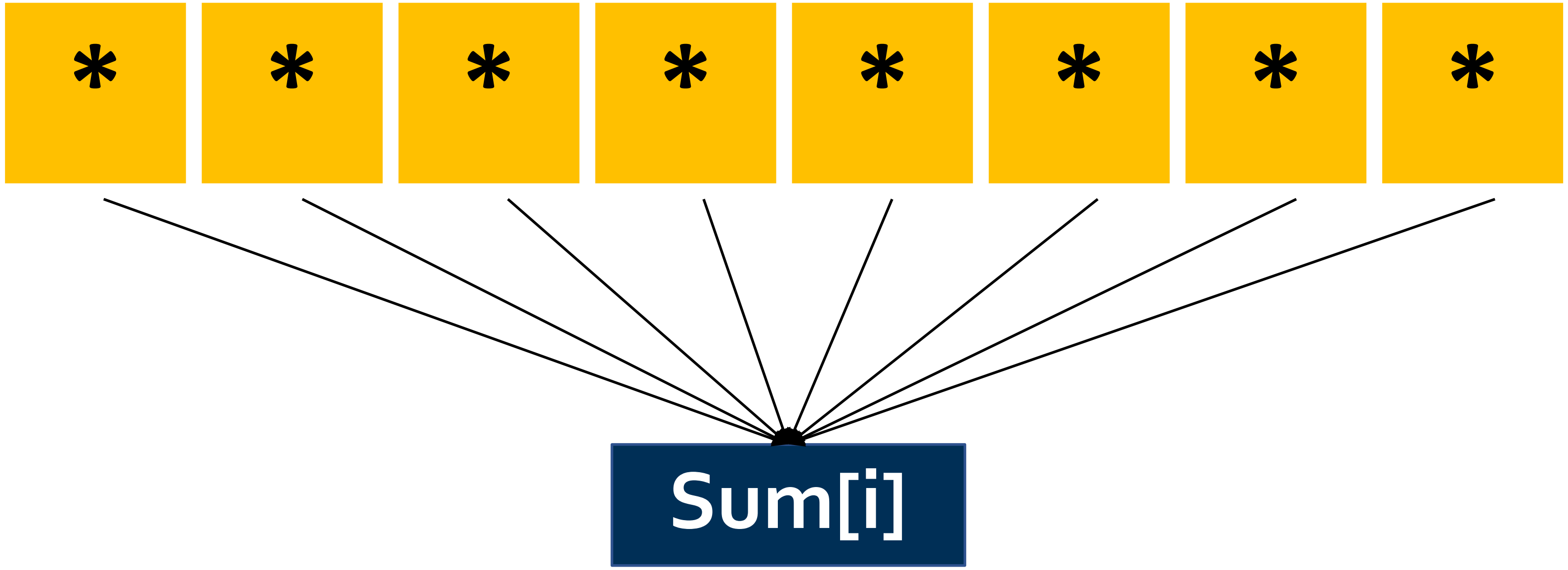
Step 3: Pipeline across Different Tiles

```
void test(int A[1024][1024], int B[1024][1024], int sum[1024])
{
#pragma HLS array_partition variable=A dim=2 factor=16 cyclic
#pragma HLS array_partition variable=B dim=2 factor=16 cyclic

     L1: for(int i = 0; i < 1024; i++) {
         L2: for(int j = 0; j < 1024; j += 16) {
 #pragma HLS pipeline
             L3: for(int jj = 0; jj < 16; jj++) {
#pragma HLS unroll
                sum[i] += A[i][j + jj] * B[i][j + jj];
}}}}
```

Partition using factor and "cyclic" or "block"

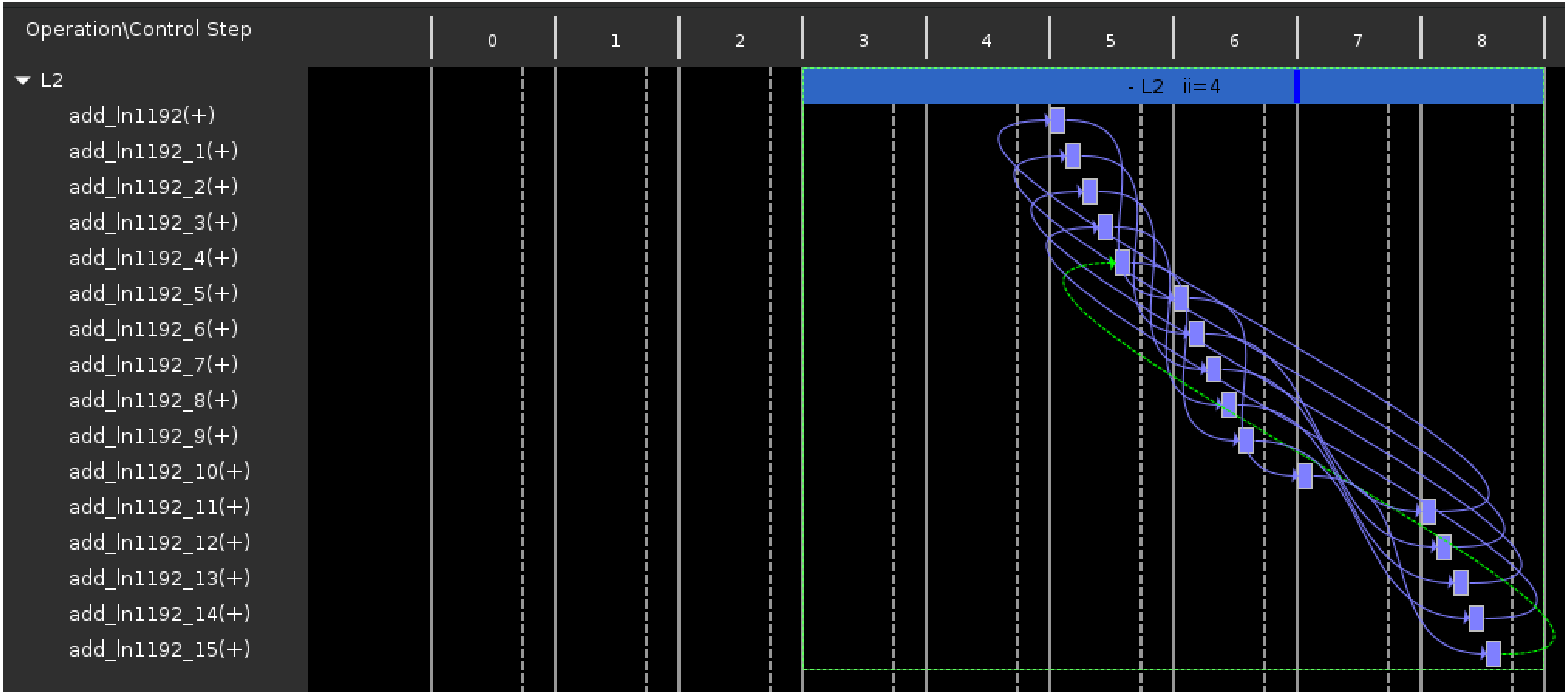
[Caution!] You May Run into II Violation

- **Not because of multiplication but because of the accumulation**



- **16 multiplications can be done in 1 cycle but not the fixed_point or floating_point accumulation…**

This is the II Violation in Vitis HLS Scheduler



Resolve the II Violation

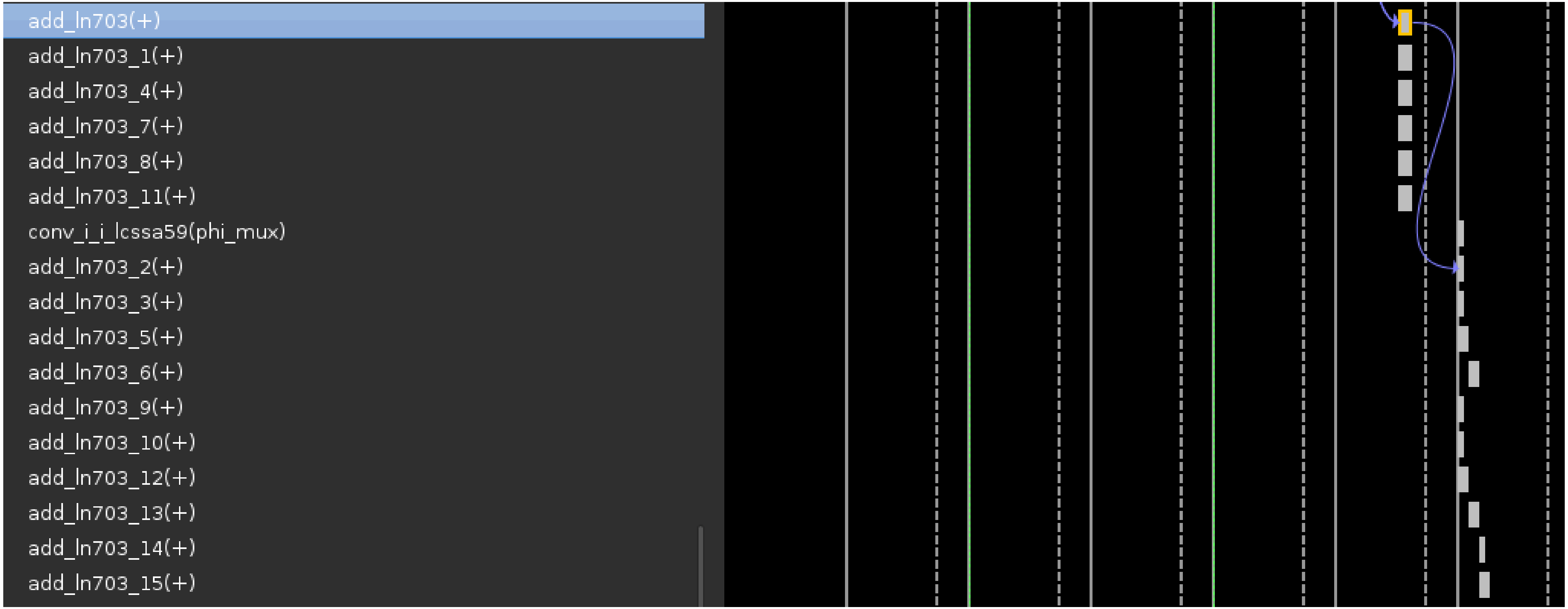
```
void test(FIX_TYPE A[1024][1024], FIX_TYPE B[1024][1024], FIX_TYPE sum[1024])
{
#pragma HLS array_partition variable=A dim=2 factor=16 cyclic
#pragma HLS array_partition variable=B dim=2 factor=16 cyclic

    L1: for(int i = 0; i < 1024; i++) {
        L2: for(int j = 0; j < 1024; j += 16) {
#pragma HLS pipeline

            FIX_TYPE sum_local[16];
            L3: for(int jj = 0; jj < 16; jj++)
                sum_local[jj] = A[i][j + jj] * B[i][j + jj];

            L4: for(int k = 0; k < 16; k++)
                sum[i] += sum_local[k];
}}}
```

```
L3: for(int jj = 0; jj < 16; jj++)
    sum[i] = A[i][j + jj] * B[i][j + jj];
```

Use a partitioned local buffer (registers)

This is the schedule after II Violation Resolved

- **Seems that HLS is smart enough to build an efficient adder tree**



More Loop Optimizations

- **Initial Interval (II) Violation**
- **Pipeline and Unroll over Functions**
- **Loop-carried Dependency**
  - Loop reorder
  - Remove false dependency
- **Loop Tiling**
- **Loop Fusion**
  - Parallel loops fused into single loop
  - Wrap loops into functions

Loop Fusion

- **When we have multiple loops, and they are independent:**

```
for(int i = 0; i < I1; i++)
    Do_Homework(i);

for(int j = 0; j < I2; j++)
    Watch_A_Movie(j);
```

Merge into one

```
for(int i = 0; i < max(I1, I2); i++)
    if(i < I1) Do_Homework(i);
    if(i < I2) Watch_A_Movie(i);
```

Wrap them up

```
// definition
Do_Homework_Wrapper() { first_for_loop}
Watch_A_Movie_Wrapper() { second_for_loop}

// function call
Do_Homework_Wrapper();
Watch_A_Movie_Wrapper();
```