The 29th IEEE International Symposium on Field-Programmable Custom Computing Machines

FPGA High-Level Synthesis: Good Practices for Quality and Productivity

Virtual Tutorial
May 9 – 12, 2021
Duration: 3 hours
Speakers:
BSc. Andre B. Perina (ICMC-USP)
Dr. Leandro de Souza Rosa (EDPR-IIT)
Dr. Vanderlei Bonato (ICMC-USP)

ISTITUTO ITALIANO
DI TECNOLOGIA
EVENT-DRIVEN PERCEPTION
FOR ROBOTICS

Outline

- HLS challenges for FPGAs
- How does HLS work?
- Design Flow
  - Typical HLS flow
  - C versus OpenCL HLS
  - HW and SW emulations
- Tuning hardware from high-level description: practical activities using Vitis
  - Loops, Data type, and Arithmetic
  - Interfaces and memory organisation

Objectives

- Give a broad view about HLS
- Understand the effects on hardware caused by data type, arithmetic, loops, interfaces and memory organisation inferred from software-like input
- Provide to the designers a set of good practices to improve the final hardware quality while minimising the implementation efforts
- Explore Vitis HLS to tune hardware from high level descriptions (input)

HLS challenges for FPGAs

- FPGAs operate in a considerable low clock rate when compared to GPUs and CPUs from equivalent technologies
- FPGAs provide excellent performance when the spatial computing paradigm is well exploited
- FPGA devices are becoming larger and larger featuring a collection of hardcores and programmable resources - System-in-Package (SiP)
  - Intel® Agilex™ (10 nm)
  - Xilinx Versal™ ACAPs (7nm)
- The HLS challenge is to convert a software-like input into such hardware model

HLS challenges for FPGAs

Sw-like constructions are quite far from hw models

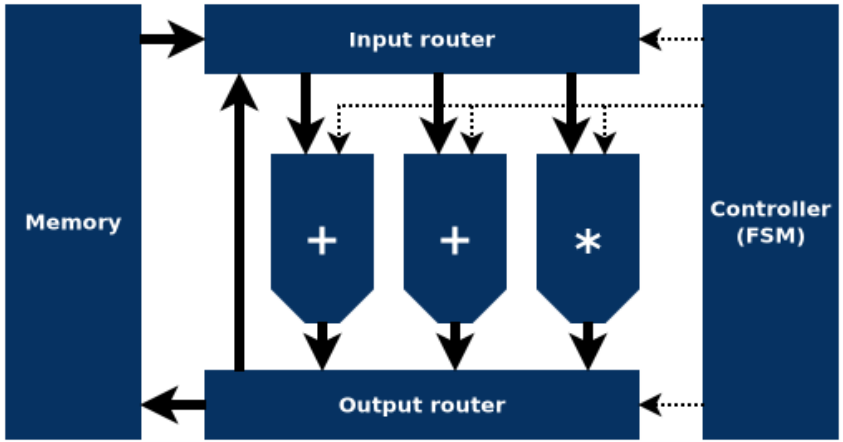
- Loops need to be unrolled and its computation pipelined
- Data types need to be converted to better fit the customized computing and storage units
- Spatial and temporal localities need to be exploited not for the hierarchical cache memory system, but for distributed RAM-Blocks and Registers
- Concurrency control mechanism exists in a much lower granularity
- Hw resources are not as abundant as memory in software systems!

How does HLS work?

How does HLS work?

Roughly speaking, an HLS-generated hardware is composed of:

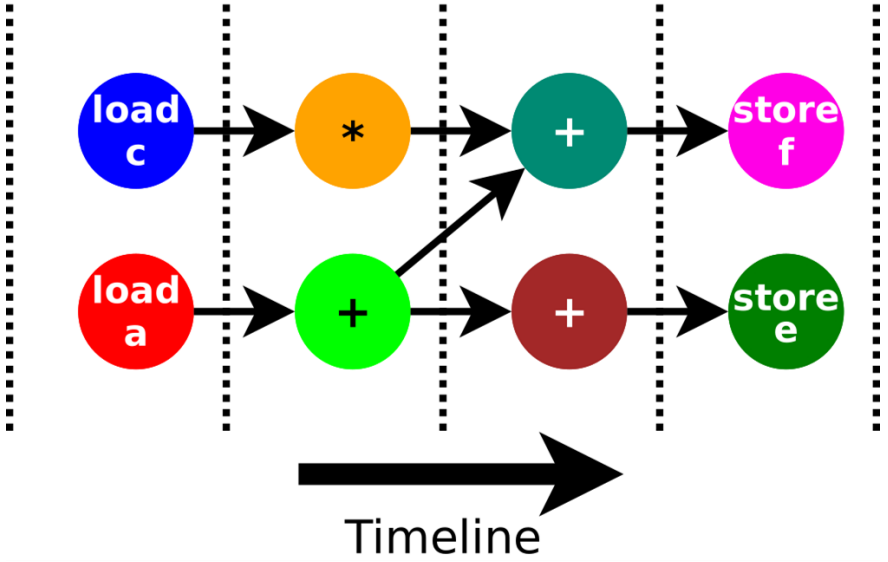
- A data storage system (registers, on-chip RAMs, DDR3...)
- Logic to distribute inputs and collect outputs
- Functional Units (FU) that perform simple operations
- A Finite State Machine (FSM) to control everything



How does HLS work?

The HLS is responsible for generating an execution timeline that is coordinated by the control FSM:

- Acquire inputs (e.g. loads)
- Activate FUs (e.g. add, mul)
- Collect results (e.g. store)

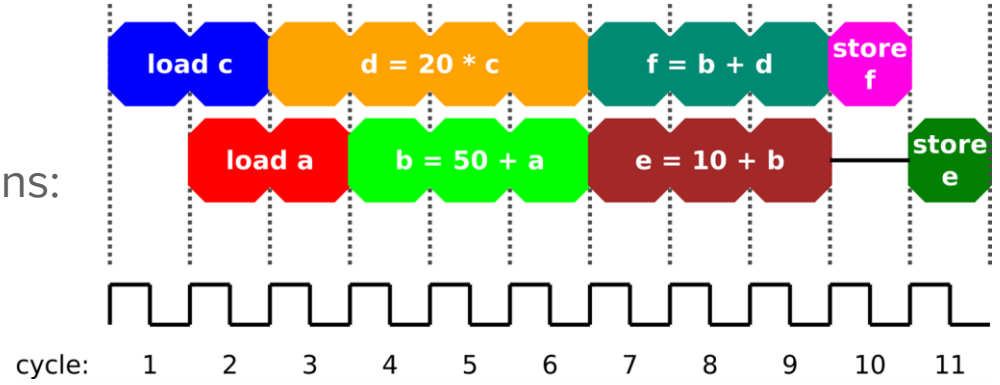


How does HLS work?

However, resources are usually constrained! For example, consider:

- Only one load may be issued per clock cycle (same for stores)
- Addition takes 3 cycles to calculate
- Multiply takes 4 cycles to calculate

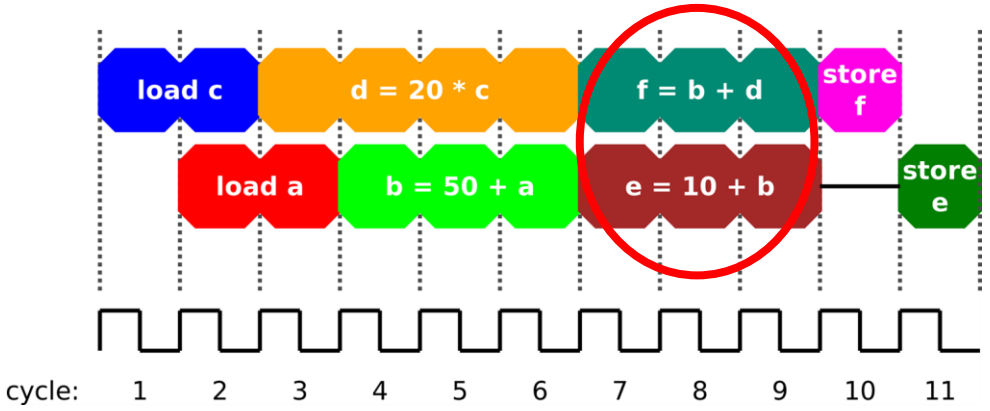
A valid scheduling under these conditions:



How does HLS work?

This scheduling would require 3 FUs:

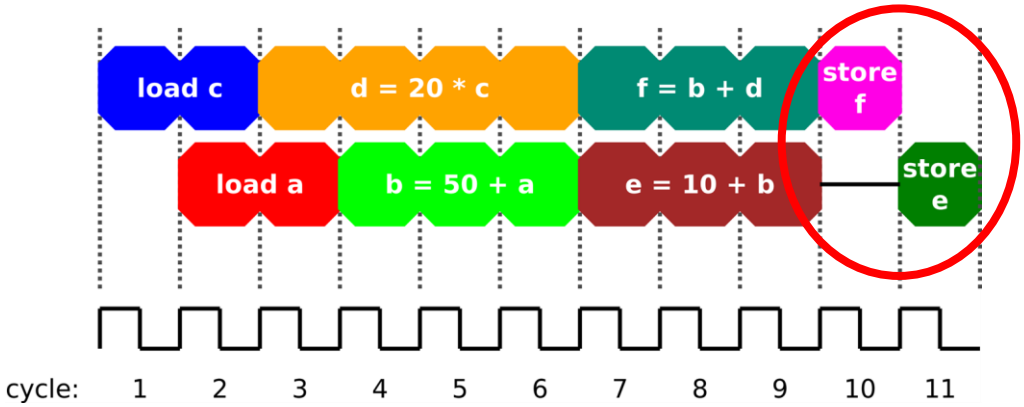
- 1 FU to execute the multiplication (orange)
- 2 FUs to execute the additions (green, lime and purple)
  - There are three additions, but only two are used in parallel



How does HLS work?

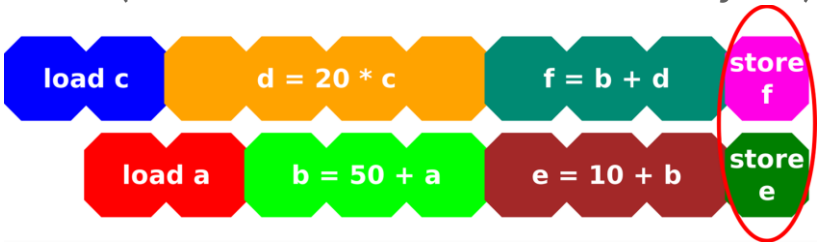
Bubbles might occur due to constraints

- For example, two stores cannot be allocated to a same clock cycle
  - It is possible to move the operations so that we can solve the bubble?

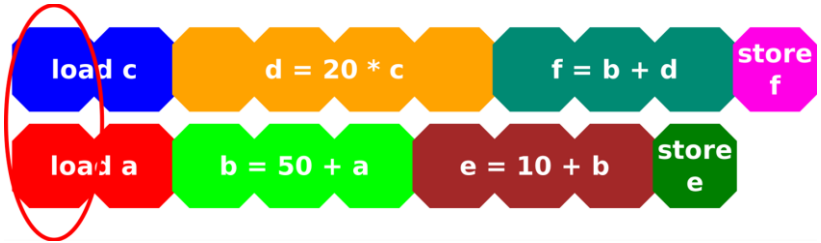


How does HLS work?

- Attempt 1: not possible (two stores in a same clock cycle)



- Attempt 2: not possible (two loads starting in a same clock cycle)

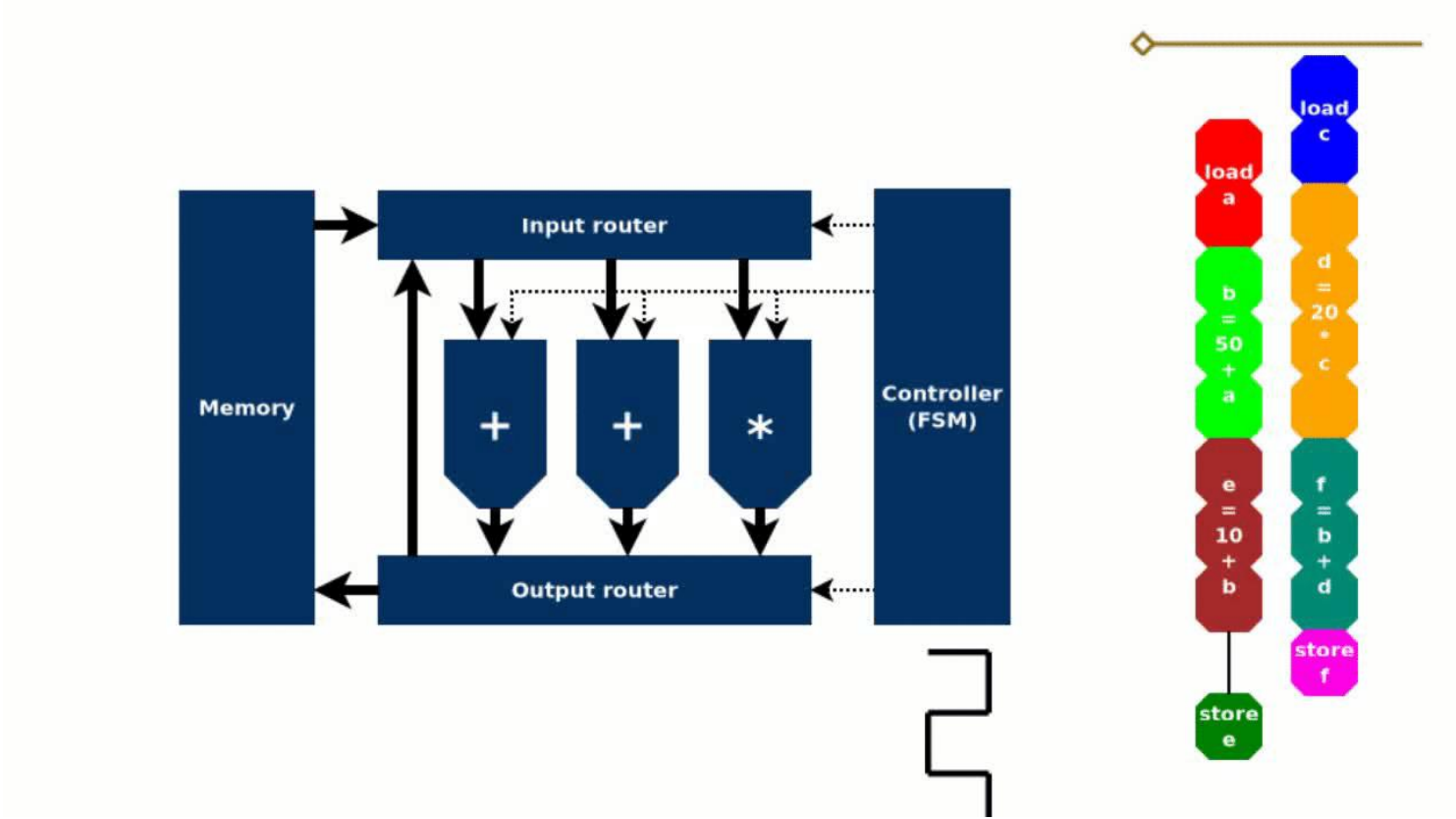


How does HLS work?

The HLS compilation process is mainly characterised by two steps:

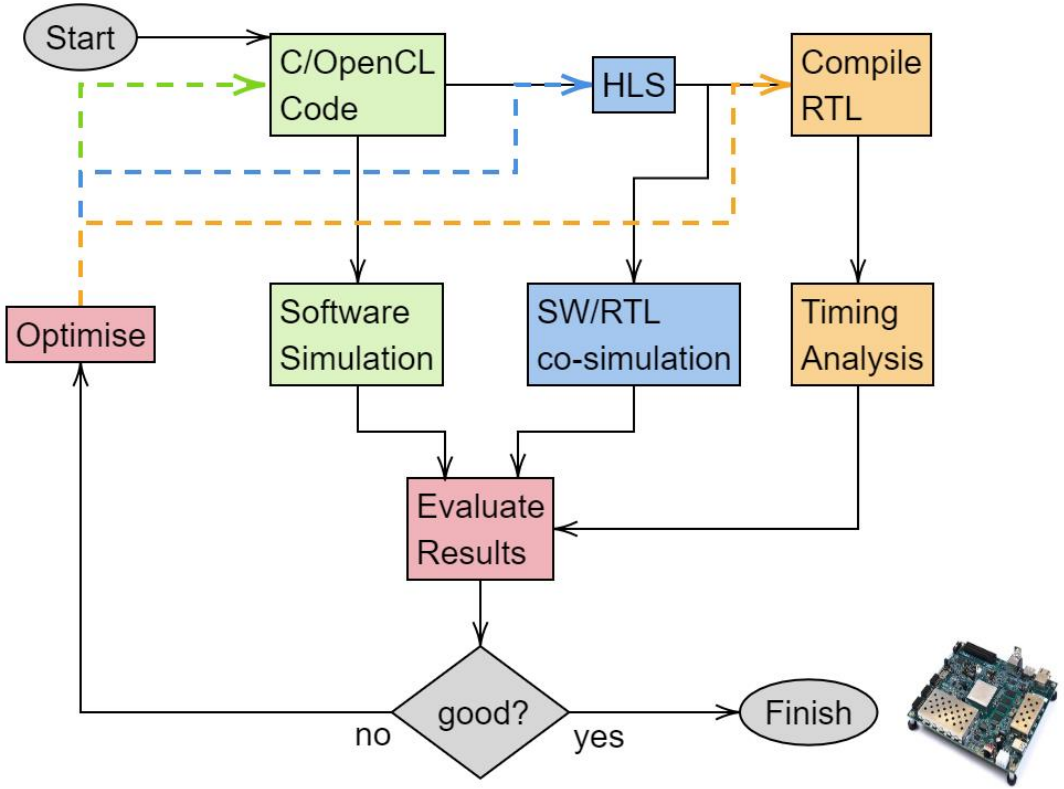
- Scheduling
  - Find out dependencies between operations in the software code
  - Attempt to find a parallel scheduling that respects the dependencies
  - Common scheduling algorithms: As-Soon-As-Possible/As-Late-As-Possible, System-of-Difference Constraints (SDC)
- Binding
  - Decide the types (and amount) of FUs
  - Assign each scheduled operation to one of the physical resources (FUs, memory ports)

How does HLS work?

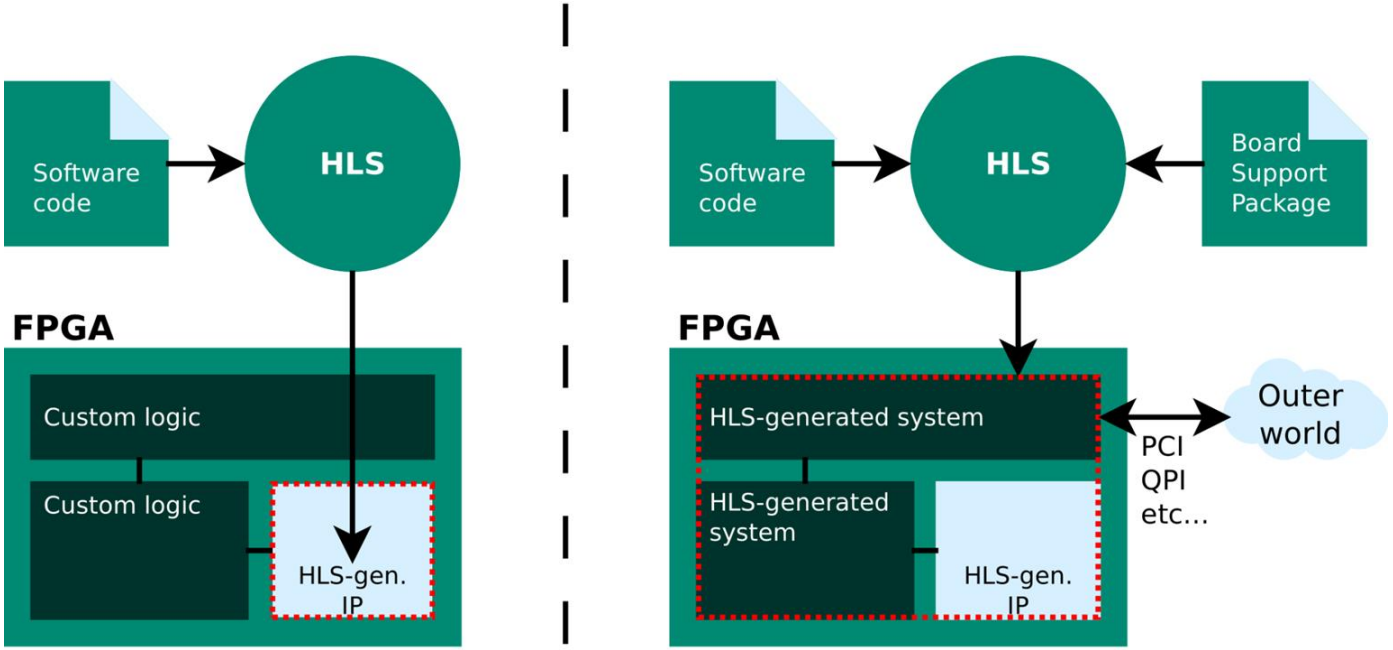


Design flow

Typical HLS flow



IP-level vs. System-level HLS



SW emulation x HW emulation

- Emulation is key to speed-up the development of hardware systems
- SW emulation
  - Code compiled to run in a CPU
  - Fast and can prove functionality
  - No HW information is in place
- HW emulation
  - Code compiled to a hardware model, usually RTL
  - Model runs in simulation tools
  - Cycle-accurate precision: can prove logic functionality, give performance and resource estimations
  - Not fast as SW emulation, but not slow as timing simulation

Factors impacting productivity in HLS

- RTL synthesis demands long compilation time
- It is desirable to refine the design as much as possible at HLS (avoiding RTL synthesis)
- Loops, Data type, and Arithmetic along with the compiler directives have significant impact on the hardware (size and performance)
- HLS compilers can also become a development time bottleneck
- To efficiently use an HLS compiler it is necessary to understand it, so you can know which options to use and when to use them

Our tutorial setup

Our setup - Xilinx Vitis

Vitis is Xilinx's umbrella framework that allows different FPGA development flows:

- Traditional RTL design
- HLS targeting IP generation
- HLS targeting system generation

https://www.xilinx.com/products/design-tools/vitis.html

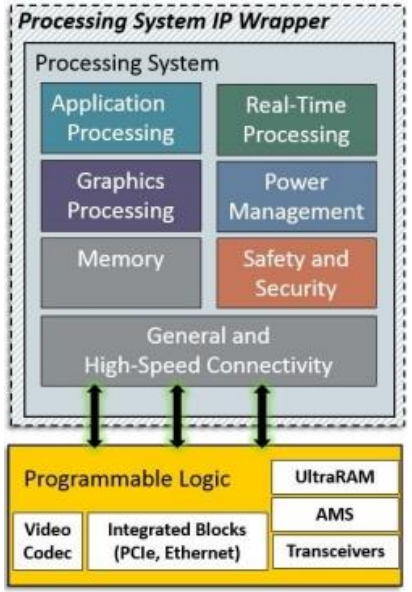
In this tutorial we will use the last approach

- Vitis will generate the whole management system around the HLS-generated kernel
- The OpenCL API is used on the host side to dispatch and manage the FPGA kernels

Our setup - Target platform

We will “target” the Xilinx Zynq UltraScale+ platform (ZCU104). Its core is an MPSoC:

- Processing System (PS): ARM processing units
- Programmable Logic (PL): FPGA component



Source: https://www.xilinx.com/products/intellectual-property/zynq-ultra-ps-e.html

Our setup - Preparation

Operating System used:

- Ubuntu 20.04 LTS

Tools required:

- Xilinx Vitis 2020.2
- ZCU104 Embedded Base Platform
- ZynqMP Common Image Package

Our setup - Preparation

Xilinx Vitis 2020.2

- Provides the full synthesis framework, from software to FPGA
- We will assume that Vitis is installed at `/opt/xilinx`

ZCU104 Embedded Base Platform

- Base files used by Vitis to generate the wrapper system around the HLS kernel
- Assuming here that these files are located at
  `/opt/xilinx/platforms/xilinx_zcu104_base_202020_1/`

Our setup - Preparation

ZynqMP Common Image Package

- Used by Vitis to generate a bootable Linux image for the platform
- Provides a cross-compilation mechanism to compile the host code for the PS
- Assuming here that the package is located at
  `/opt/xilinx/rootfs/xilinx-zynqmp-common-v2020.2/`
- and that the cross-compilation environment is located at
  `/opt/xilinx/petalinux`

Refer to the simplified installation guide for more info!

Hello world!

We will start with a simple “hello world” example that performs a vector add.

Two development approaches:

- Using Vitis GUI (Eclipse-based IDE)
- Using command line

Command line is more suitable for remote programming (e.g. via ssh). We will use that.

- Both command line and GUI development approaches produce the same results and can be (sort of) switched during development!

Hello world!

We will abstract most details from the Vitis programming flow

- Xilinx provides many learning resources for their tools (a good start is https://github.com/Xilinx/Vitis-Tutorials)

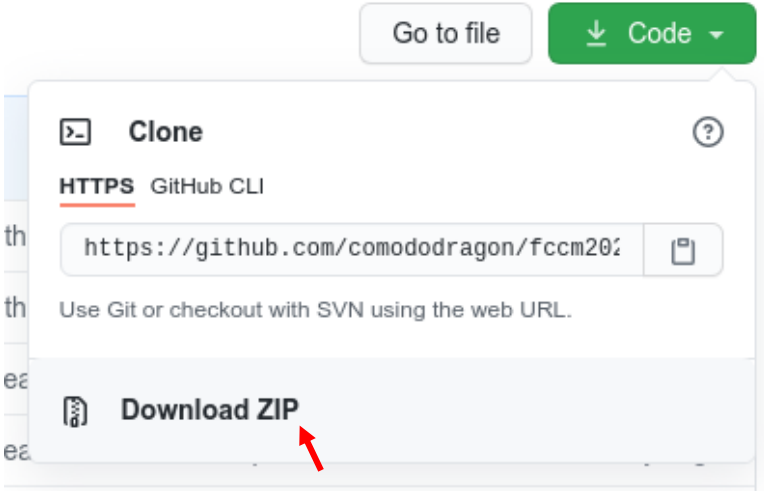
We will use a skeleton project that simplifies all tasks using a single Makefile.

Hello world!

First step: clone the git repository:

```
$ git clone https://github.com/comododragon/fccm2021-tutorial
```

Alternatively, you can access the link above and download the repo as a zip file:



Hello world! - Host code

The host code uses the OpenCL API to dispatch the kernel and manage its buffers.

```
int main(int argc, char** argv) {
    /* ... */

    cl::Buffer buffer_in1(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, size, in1, &err);
    cl::Buffer buffer_in2(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, size, in2, &err);
    cl::Buffer buffer_output(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, size, out, &err);

    krnl_vector_add.setArg(0, buffer_in1);
    krnl_vector_add.setArg(1, buffer_in2);
    krnl_vector_add.setArg(2, buffer_output);
    krnl_vector_add.setArg(3, size);

    q.enqueueMigrateMemObjects({buffer_in1, buffer_in2}, 0);
    q.enqueueTask(krnl_vector_add);
    q.enqueueMigrateMemObjects({buffer_output}, CL_MIGRATE_MEM_OBJECT_HOST);
    q.finish();

    /* ... */
}
```

More information about the OpenCL specification can be found at opencl.org

Hello world! - Kernel code

The kernel code is a simple vector-add loop wrapped on a C function

```
extern "C" {

void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE off
        out[i] = in1[i] + in2[i];
    }
}

}
```

Hello world! - Initial setup

But first, we must set up our environment as required by Vitis:

```
$ source ./setup.sh
```

This script initialises the shell to the Vitis build environment.

Hello world! - Build and run 101

Now we're good to go! To build the skeleton project:

```
$ make build TARGET=<TGT>
```

Where **<TGT>** may be **sw_emu**, **hw_emu** or **hw**:

- **sw_emu**: software emulation
- **hw_emu**: hardware emulation
- **hw**: full hardware synthesis

Hello world! - Trying the sw_emu

So, let's build and run the **sw_emu**:

```
$ make build TARGET=sw_emu
$ make run TARGET=sw_emu
```

QEMU will boot the SD card image

(`project/package.sw_emu/sd_card.img`).

Then run the following command to run the host code on the emulated platform:

```
$ cd /mnt/sd-mmcblk0p1/init_and_run.sh
```

Hello world! - Trying the sw_emu

The host software will prepare the buffers, execute the vector add on the PS side (since it is **sw_emu**), retrieve the results and validate. The message **TEST_PASSED** should print on screen:

```
Found Platform
Platform Name: Xilinx
INFO: Reading vadd.xclbin
Loading: 'vadd.xclbin'
INFO: [SW-EM 09-0] Unable to find emconfig.json. Using default
device.
Trying to program device[0]: xilinx_zcu104_base_202020_1
Device[0]: program successful!
TEST PASSED
INFO: host run completed.
```

The emulation can be terminated by pressing `CTRL+A` and `X`.

Hello world! - Trying the hw_emu

The following commands are available for **hw_emu** and **hw**:

- Generate the whole system, from HLS to bootable SD image:

```
$ make build TARGET=hw_emu
```

- Run only the HLS compiler, skip the rest:

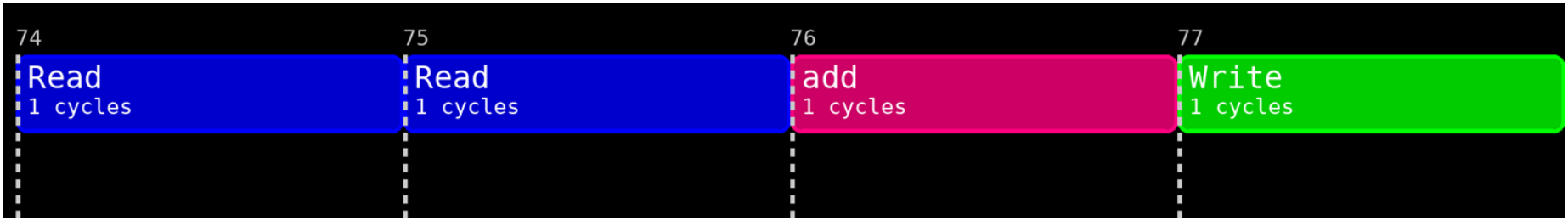
```
$ make hls TARGET=hw_emu
```

- Open the reports generated by the command above:

```
$ make report TARGET=hw_emu
```

Hello world! - Trying the hw_emu

When HLS compilation is performed, we can have a more precise information about the final design:



Scheduling of one iteration

Hello world! - Post-build analysis

Useful HLS-generated information (also generated for **hw** projects):

- `_x.hw_emu.xilinx_zcu104_base_202020_1/reports/vadd/hls_reports/vadd_csynth.rpt`
  - Textual report with latency details and resource estimates
- `_x.hw_emu.xilinx_zcu104_base_202020_1/vadd/vadd/vadd/solution/.autopilot/db/vadd.verbose.rpt`
  - An even more detailed (and hidden) textual report, including scheduling and binding information
- `_x.hw_emu.xilinx_zcu104_base_202020_1/logs/vadd/vadd_vitis_hls.log`
  - Log file generated during the HLS compilation. Detected issues are reported (e.g. failure to pipeline)

Hello world! - Post-build analysis

The information on these reports are also available in *_summary files that can be opened using vitis_analyzer (run **vitis_analyzer <FILE>** to open the GUI):

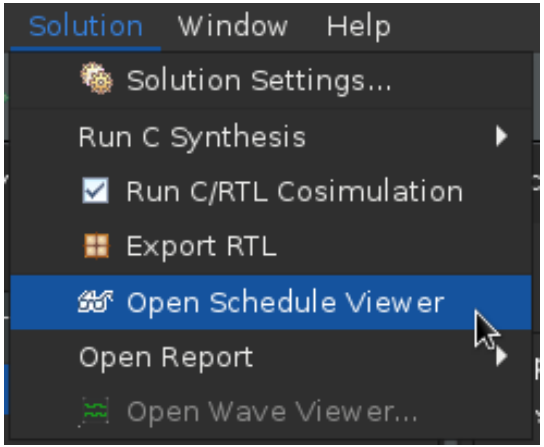
- `project/_x.hw_emu.xilinx_zcu104_base_202020_1/vadd.xo.compile_summary`
  - Contains information related to the HLS scheduling and binding
  - Similar to **`vadd_csynth.rpt`**, but structured
- `project/build_dir.hw_emu.xilinx_zcu104_base_202020_1/vadd.xclbin.link_summary`
  - Contains more information on the system generated around the kernel
- `project/vadd.xclbin.package_summary`
  - Contains information about the final SD card image generation

Hello world! - Post-build analysis

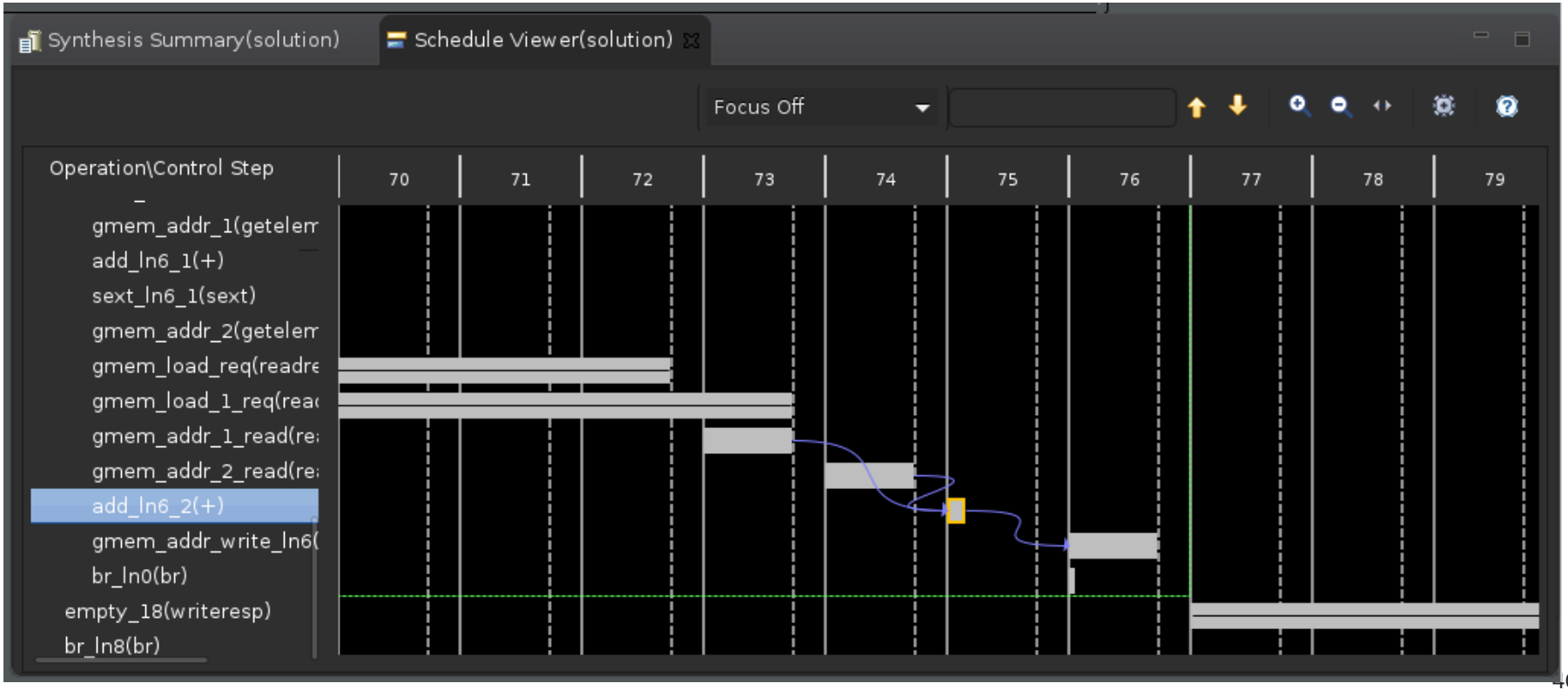
Projects generated in command line can be opened in the Vitis HLS IDE for a visual representation of the scheduling:

```
$ vitis_hls -p project/_x.hw_emu.xilinx_zcu104_base_202020_1
/vadd/vadd/vadd/
```

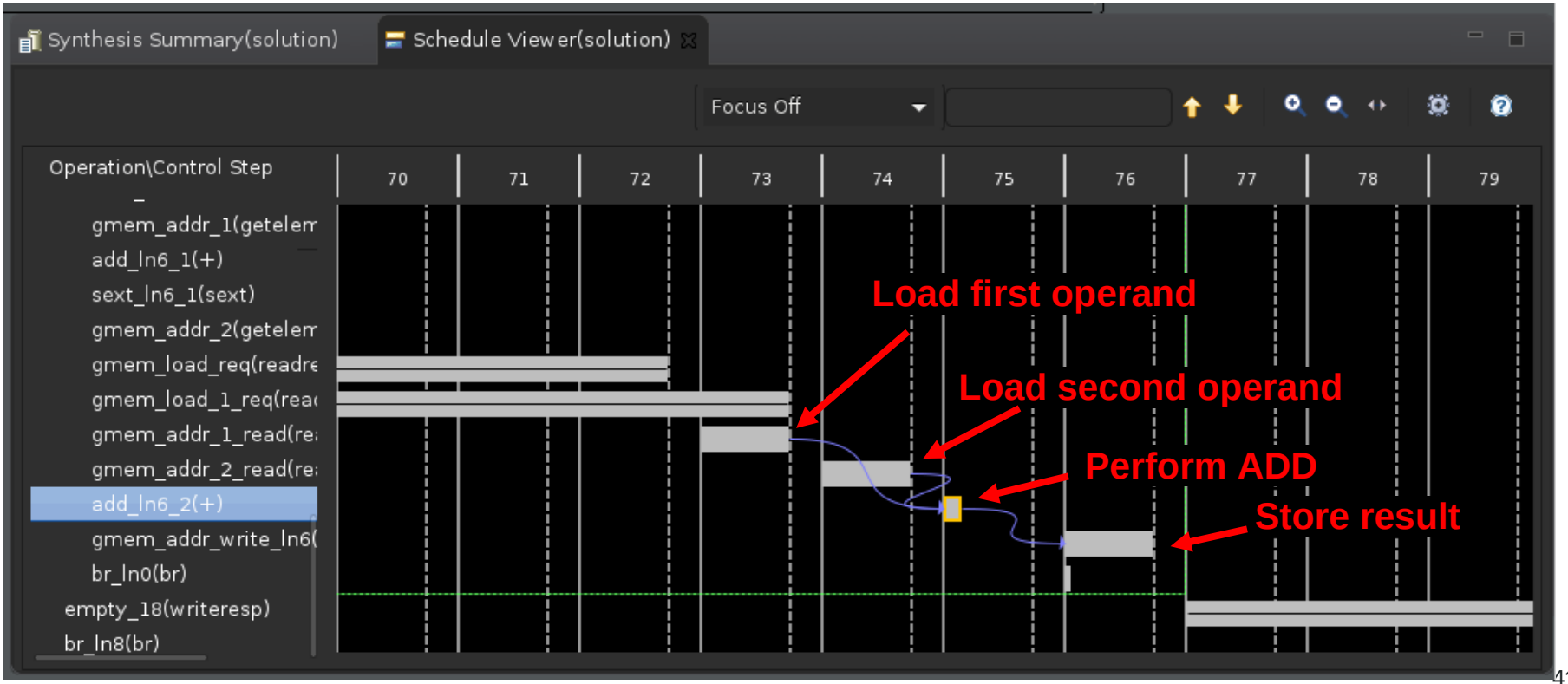
then, navigate to **Solution > Open Schedule Viewer**



Hello world! - Generated scheduling



Hello world! - Generated scheduling



Hello world! - Latency information

The **vadd_csynth.rpt** shows the latency information. Since the loop is not statically bounded, Vitis only shows the latency for a single iteration:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
?	?	?	?	?	?	none

	Latency (cycles)		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined
- LOOP_4_1	?	?	75	-	-	?	no

Hello world! - Latency information

We can use the **LOOP_TRIPCOUNT** pragma to suggest a trip count to Vitis

- Will not affect synthesis at all, only provide more latency information
- Pragma must be placed inside the associated loop's scope

```
void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT max=4096
#pragma HLS PIPELINE off
        out[i] = in1[i] + in2[i];
    }
}
```

Pragmas are used to inform the compiler about desired optimisations or to provide more information (e.g. inform false dependencies).

Hello world! - Latency information

More detailed latency information is now available considering a trip count from 0 to 4096:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
2	307270	13.334 ns	2.049 ms	3	307271	none

	Latency (cycles)		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined
- LOOP_4_1	0	307200	75	-	-	0 ~ 4096	no

Loops Pipeline

Loops

- Loops generally contain the bulk of the computations.
- Nested loops are common structures.
  - It is natural to optimize loops.
- Some important opt. for loops:
  - Loop pipeline
  - Loop unroll
  - Number of functional units

```
96  LANES: for (int i = 0; i < PAR_FACTOR; i ++ )
97  {
98    INSERTION_SORT_OUTER: for (int j = 0; j < K_CONST; j ++ )
99    {
100     #pragma HLS pipeline
101     pos = 1000;
102     INSERTION_SORT_INNER: for (int r = 0; r < K_CONST; r ++ )
103     {
104       #pragma HLS unroll
105       pos = ((knn_set[i*K_CONST+j] < min_distance_list[r]) && (pos > K_CONST)) ? r : pos;
106     }
107
108     INSERT: for (int r = K_CONST ;r > 0; r -- )
109     {
110       #pragma HLS unroll
111       if(r-1 > pos)
112       {
113         min_distance_list[r-1] = min_distance_list[r-2];
114         label_list[r-1] = label_list[r-2];
115       }
116       else if (r-1 == pos)
117       {
118         min_distance_list[r-1] = knn_set[i*K_CONST+j];
119         label_list[r-1] = i / (PAR_FACTOR / 10);
120       }
121     }
122   }
}
```

Digit Recognition benchmark from Rosetta [1]

[1] Zhou, Y., Gupta, U., Dai, S., Zhao, R., Srivastava, N., Jin, H., ... & Zhang, Z. (2018, February). Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 269-278).

Example: Loop **Without** Pipeline

```h
typedef double data_t;
#define N 5
void compute(data_t din, data_t coeff, data_t *dout);
```

```c
#include "compute.h"

void compute(data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
    }

    idx = (idx+1)%N;

    *dout = acc;
    return;
}
```

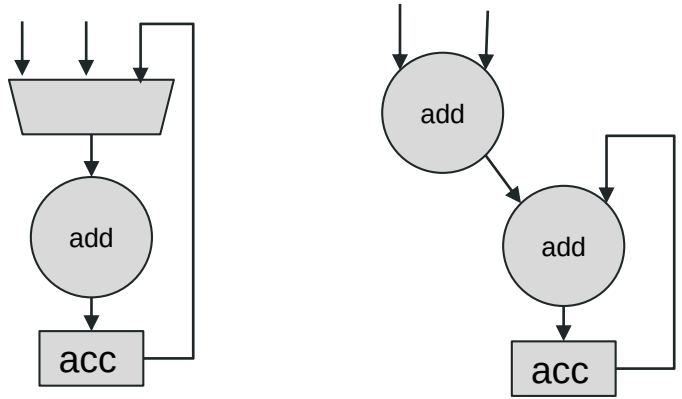
Clock Cycle	FU Add - Latency: X cycles
0*X	$temp^0$ = v[i]+coef
1*X	acc^0 = acc + temp
2*X	$temp^1$ = v[i]+coef
3*X	acc^1 = acc + temp
4*X	$temp^2$ = v[i]+coef
5*X	acc^2 = acc + temp
6*X	$temp^3$ = v[i]+coef
7*X	acc^3 = acc + temp
8*X	$temp^4$ = v[i]+coef
9*X	acc^4 = acc + temp

Example: Loop **Without** Pipeline

```h
typedef double data_t;
#define N 5
void compute(data_t din, data_t coeff, data_t *dout);
```

```c
#include "compute.h"

void compute(data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
    }

    idx = (idx+1)%N;

    *dout = acc;
    return;
}
```

- Can we do it faster with 2 Addition FUs?
  - No.
  - Adding another FU saves the routing and multiplexing resources, but costs resources for another FU.
    - It is worthy in some cases!



Example: Loop **With** Pipeline

```h
typedef double data_t;
#define N 5
void compute(data_t din, data_t coeff, data_t *dout);
```

```c
#include "compute.h"

void compute(data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
    }

    idx = (idx+1)%N;

    *dout = acc;
    return;
}
```

Clock Cycle	FU Add - Latency: X cycles
0*X	$temp^0 = v[i]+coef$
1*X	$acc^0 = acc + temp$
2*X	$temp^1 = v[i]+coef$
3*X	$acc^1 = acc + temp$
4*X	$temp^2 = v[i]+coef$
5*X	$acc^2 = acc + temp$
6*X	$temp^3 = v[i]+coef$
7*X	$acc^3 = acc + temp$
8*X	$temp^4 = v[i]+coef$
9*X	$acc^4 = acc + temp$

Example: Loop **With** Pipeline

```h
typedef double data_t;
#define N 5
void compute(data_t din, data_t coeff, data_t *dout);
```

```c
#include "compute.h"

void compute(data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
    }

    idx = (idx+1)%N;

    *dout = acc;
    return;
}
```

- Nothing changes with 1 FU.
- Can we do it faster with 2 Addition FUs?
  - Yes.

Clock Cycle	FU Add 0	FU Add 1
0*X	$temp^0$ = v[i]+coef	
1*X	$temp^1$ = v[i]+coef	acc^0 = acc + temp
2*X	$temp^2$ = v[i]+coef	acc^1 = acc + temp
3*X	$temp^3$ = v[i]+coef	acc^2 = acc + temp
4*X	$temp^4$ = v[i]+coef	acc^3 = acc + temp
5*X		acc^4 = acc + temp

Example: Loop **With** Pipeline

- Loop pipeline allows to explore instruction level parallelism between loop iterations.
- That is, it starts computing the next loop iteration before the previous ends.
- The speed-up depends on the number of FUs.

- Nothing changes with 1 FU.
- Can we do it faster with 2 Addition FUs?
  - Yes.

Clock Cycle	FU Add 0	FU Add 1
0*X	$temp^0$ = v[i]+coef	
1*X	$temp^1$ = v[i]+coef	acc^0 = acc + temp
2*X	$temp^2$ = v[i]+coef	acc^1 = acc + temp
3*X	$temp^3$ = v[i]+coef	acc^2 = acc + temp
4*X	$temp^4$ = v[i]+coef	acc^3 = acc + temp
5*X		acc^4 = acc + temp

Example Results

```h
typedef double data_t;
#define N 5
void compute(data_t din, data_t coeff, data_t *dout);
```

```c
#include "compute.h"

void compute(data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
    }

    idx = (idx+1)%N;

    *dout = acc;
    return;
}
```

No pipeline

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	55	55	11	-	-	5	no

Pipeline 1FU

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	41	41	10	8	1	5	yes

Pipeline 2FUs

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	31	31	8	6	1	5	yes

Understanding the Results

- Trip Count:
  - How many iterations the loop has.
- Iteration Latency (IL):
  - How many clock cycles for completing 1 iteration of the loop.
- Initial Interval (II):
  - How many clock cycles between starting two loop iterations.
- “Latency”:
  - The total number of clock cycles for completing all computations in the loop
  - “Latency” = IL + II * (trip count -1)
- A loop without pipeline:
  - II = IL

```
1  #include "compute.h"
2
3  void compute(data_t din, data_t coeff, data_t *dout){
4      static data_t v[N];
5      static int idx=0;
6
7      data_t acc = 0;
8      v[idx] = din;
9
10     loop1: for(int i = 0; i < N; i=i+1){
11         acc = acc + v[i] + coeff;
12     }
13
14     idx = (idx+1)%N;
15
16     *dout = acc;
17     return;
18 }
19
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	31	31	8	6	1	5	yes

Making Loop Pipelines Worse

- There are several code structures that make more difficult pipeline loops:
  - **Loop carried dependencies:** can constraint the minimal II, limiting the amount of parallelism even if FUs are available:
    - These dependencies are created when iterations use results produced in previous iterations.
    - Similar example with an intra-loop dependency added, using 2 FUs.

```
loop1: for(int i = 1; i < N; i=i+1){
    acc = acc + v[i-1] + coeff;
    v[i] = acc;
}
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	32	32	8	8	1	4	yes

Making Loop Pipelines Worse

- There are several code structures that make more difficult pipeline loops:
  - **Loop carried dependencies.**
  - **Dependencies between specific elements of arrays:** are hard to identify. More modern compilers give the designer the chance to remove such dependencies manually:

```
#pragma HLS PIPELINE II=1
#pragma HLS dependence variable=buff_A inter false
#pragma HLS dependence variable=buff_B inter false
if (col < cols) {
buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]
buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]
```

Making Loop Pipelines Worse

- There are several code structures that make more difficult pipeline loops:
  - **Loop carried dependencies.**
  - **Dependencies between specific elements of arrays.**
    - Identifying and solving such dependencies is a current topic of research:
      - Liu, J., Wickerson, J., Bayliss, S., & Constantinides, G. A. (2017). Polyhedral-based dynamic loop pipelining for high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 37(9), 1802-1815.
      - Zuo, W., Li, P., Chen, D., Pouchet, L. N., Zhong, S., & Cong, J. (2013, September). Improving polyhedral code generation for high-level synthesis. In 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS) (pp. 1-10). IEEE.

Making Loop Pipelines Worse

- There are several code structures that make more difficult pipeline loops:
  - **Loop carried dependencies.**
  - **Dependencies between specific elements of arrays.**
  - **Dynamic code**:
    - Unknowns during compilation time lead the compiler to be more conservative.
    - Without knowing the offset, the compiler cannot infer the pipeline

```
void compute(int offset, data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i] + coeff;
        v[i] = acc;
    }
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	31	31	8	6	1	5	yes

```
void compute(int offset, data_t din, data_t coeff, data_t *dout){
    static data_t v[N];
    static int idx=0;

    data_t acc = 0;
    v[idx] = din;

    loop1: for(int i = 0; i < N; i=i+1){
        acc = acc + v[i-offset] + coeff;
        v[i] = acc;
    }
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	40	40	8	8	1	5	yes

Making Loop Pipelines Worse

- There are several code structures that make more difficult pipeline loops:
  - **Loop carried dependencies.**
  - **Dependencies between specific elements of arrays.**
  - **Non-static code**:
  - **Number of FUs bottlenecks:**
    - Be sure to increase the number of FUs for all operations in the code, to avoid the creation of bottlenecks.
    - Memory can also be a resource bottleneck.

```
loop1: for(int i = 0; i < N; i=i+1){
    acc1 = acc1 + v[i] + coeff;
    acc2 = acc2 - v[i] - coeff;
}
```

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	42	42	11	8	1	5	yes

2 ADDs, 1 SUB

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop1	31	31	8	6	1	5	yes

2 ADDs, 2 SUBs

Loop Unroll

Loop Unroll

Loop unrolling copies the instructions of the loop body, reducing the iterations accordingly.

```
loop1: for(int i = 0; i < N; i=i+1){
    v2[i] = v[i] + coeff;
}
```

Unroll factor = 2

```
loop1: for(int i = 0; i < N; i=i+2){
    v2[i] = v[i] + coeff;
    v2[i+1] = v[i+1] + coeff;
}
```

Doing so, more opportunities for parallelism might be exposed.

The number of FUs must be adjusted.

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
Unroll factor = 2, nFU ADD = 1 → - loop1	7	7	6	2	1	2	yes
Unroll factor = 2, nFU ADD = 2 → - loop1	5	5	5	1	1	2	yes

Defining Pragmas Values

Design Space Exploration

- How to define the values for the pragma which gives us hardware with the best area-speed trade-offs?
  - Loop pipeline II
  - Loop unroll factor
  - Memory partition factor
  - Number of FUs
- We call this design space exploration (DSE), which has been a research topic for years.

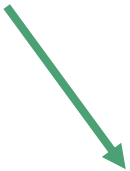
Design Space Exploration

- How to define the values for the directives which gives us hardwares with the best area-speed trade-offs?
- DSE approaches synthesise or estimate the hardware usage and speed (metrics), and use a variety of models to try to predict the best combinations or reduce the number of synthesis. Some works on the topic:
    - Schafer, B. C., & Wang, Z. (2019). High-Level Synthesis Design Space Exploration: Past, Present, and Future. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10), 2628-2639.
    - Ferretti, L., Ansaloni, G., & Pozzi, L. (2018, October). Lattice-traversing design space exploration for high level synthesis. In 2018 IEEE 36th International Conference on Computer Design (ICCD) (pp. 210-217). IEEE.
- HLS tools are always trying to develop their own DSE, using the knowledge they have on their own hardware/tools.

Practical examples - Loop Optimisations

Pipeline - Hello world!

What about this pragma?



```
extern "C" {

void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT max=4096
#pragma HLS PIPELINE off
        out[i] = in1[i] + in2[i];
    }
}

}
```

Usually Vitis will automatically apply pipeline to loops.
We used “**`#pragma HLS PIPELINE off`**” to deactivate the automatic pipeline

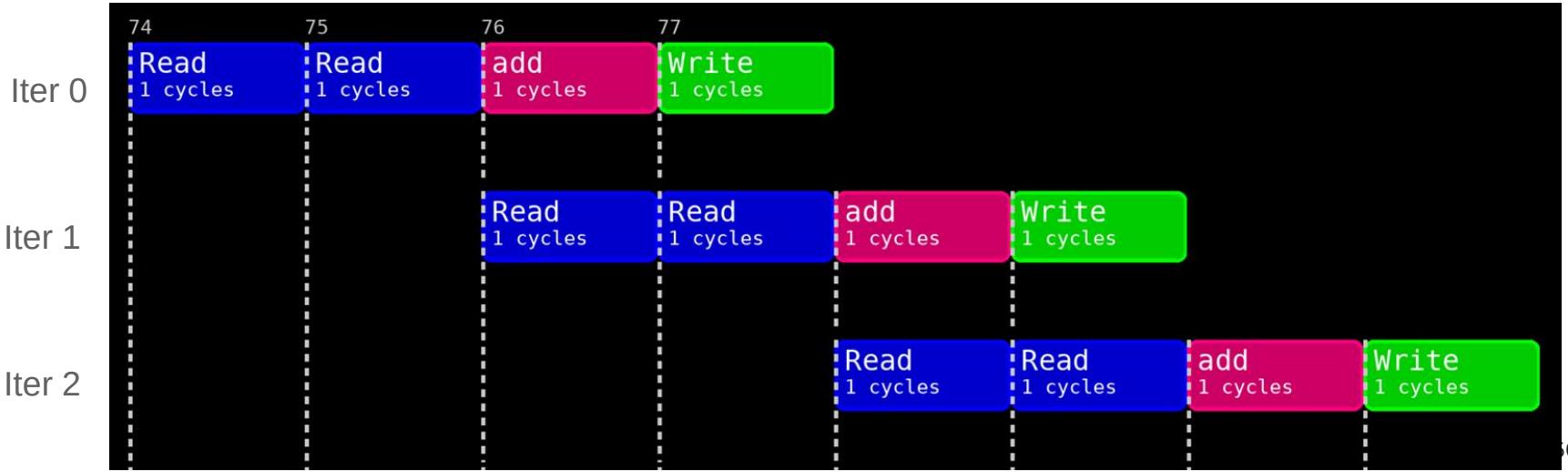
Pipeline - Hello world!

If we remove the pragma, the loop will be automatically pipelined.

During **hw_emu** or **hw** build, the following message is printed:

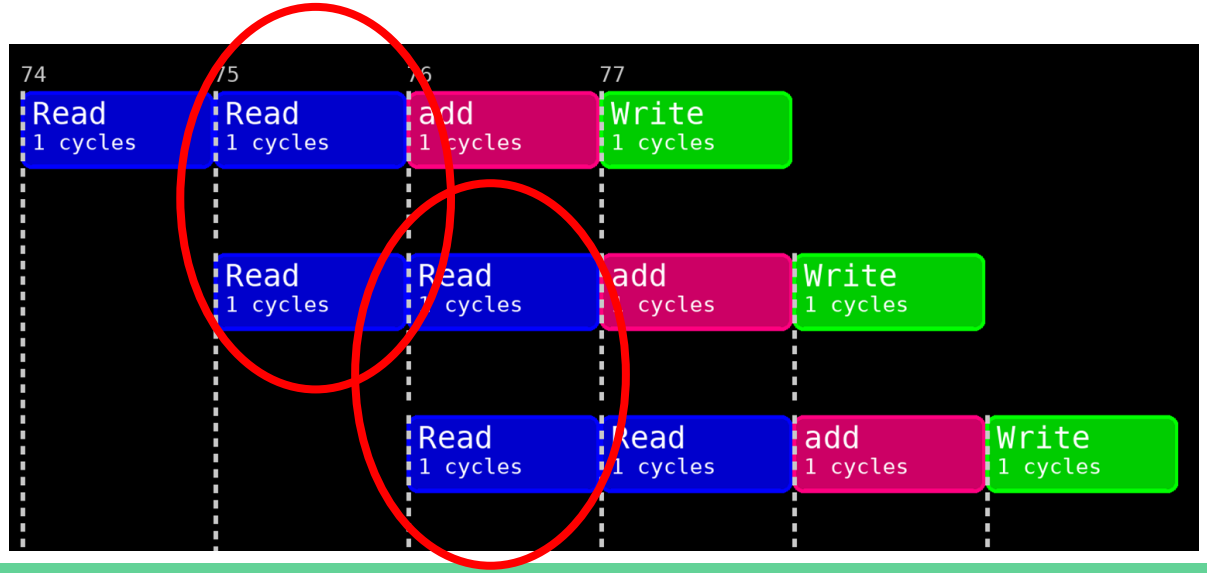
```
INFO: [v++ 200-1470] Pipelining result: Target II=1, Final II=2, Depth=75
```

This indicates that the loop was pipelined with an initiation interval of 2:



Pipeline - Hello world!

Initiation interval of 1 is not possible, since this would require two reads to happen simultaneously:



Pipeline - Hello world!

Initiation interval of 1 is not possible, since this would require two reads to happen simultaneously:

Such scheduling is not possible, since the input arrays are in a single-ported off-chip memory interface!

Vitis informs us about the problem during HLS:

```
WARNING: [v++ 200-885] Unable to schedule bus request on port 'gmem'
due to limited memory ports. Please consider using a memory core
with more ports or partitioning the array.
```

Pipeline - Hello world!

The result of automatic pipeline is similar to providing the PIPELINE pragma as follows:

```
void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}
```

The compiler will attempt to minimise the initiation interval towards the defined II, however there is no guarantee of success.

Pipeline - Hello world!

The **vadd_csynth.rpt** now reports a latency including pipeline:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
2	8334	13.334 ns	55.563 us	3	8335	none

	Latency (cycles)		Iteration	Initiation Interval		Trip	
Loop Name	min	max	Latency	achieved	target	Count	Pipelined
- LOOP_4_1	0	8264	75	2	1	0 ~ 4096	yes

Without pipeline: 307270 cycles

With pipeline: 8334 cycles

Pipeline - Loop-carried dependencies

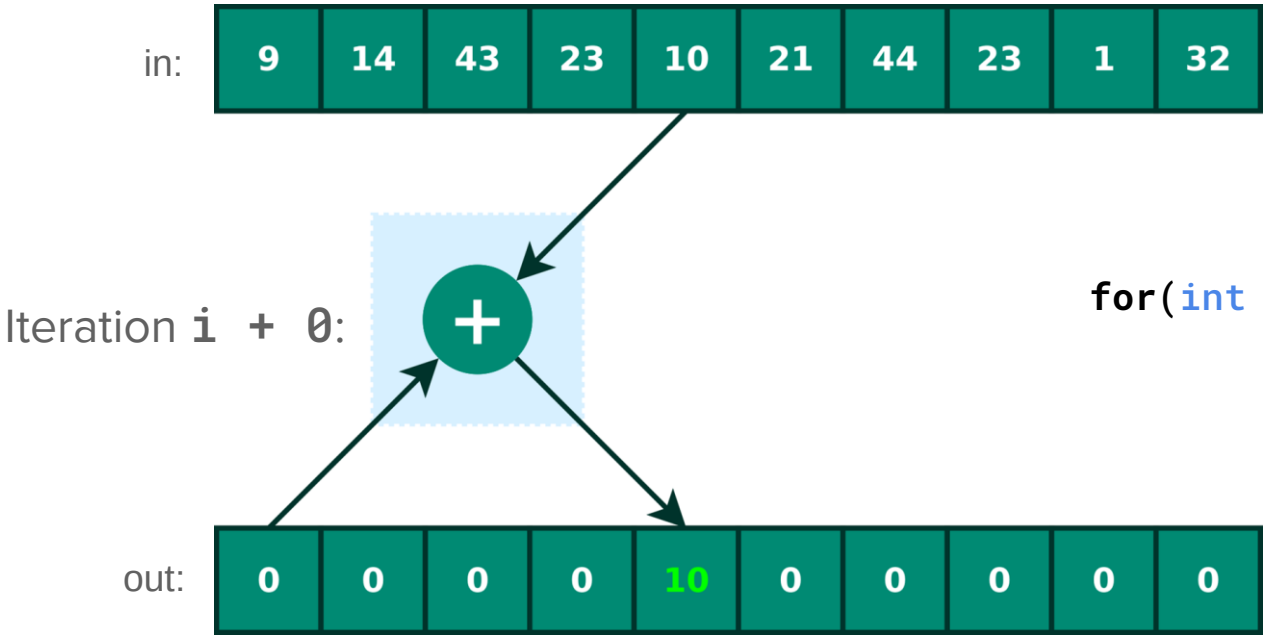
However, sometimes the code itself imposes restrictions to pipeline:

```
#define DISTANCE 4

extern "C" {

void example(unsigned int *in, unsigned int *out, int size) {
    for(unsigned int i = DISTANCE; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4092
        out[i] = in[i] + out[i - DISTANCE];
    }
}

}
```

The calculation of out[i] depends on out[i - 4]!

Pipeline - Loop-carried dependencies

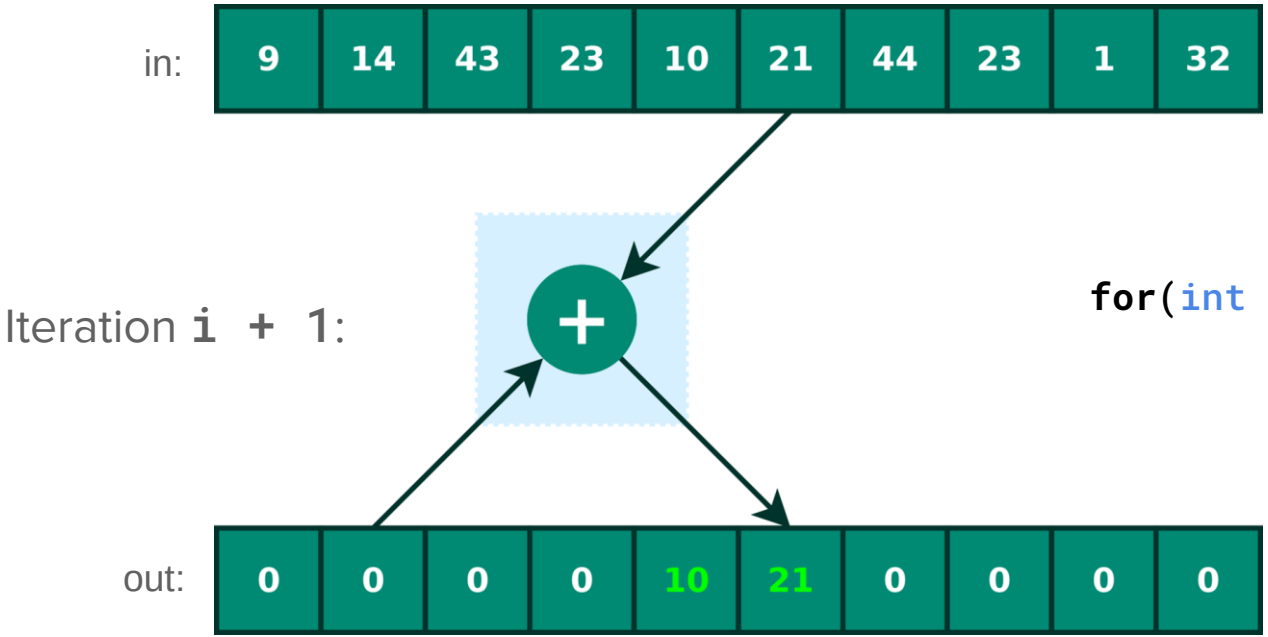
The read of **out[i]** at iteration **i + 4** cannot be scheduled before the **out[i]** is calculated at iteration **i**. This imposes a restriction on the initiation interval.




```
for(int i = 4; i < size; i++)
        out[4] = in[4] + out[0]
```

Pipeline - Loop-carried dependencies

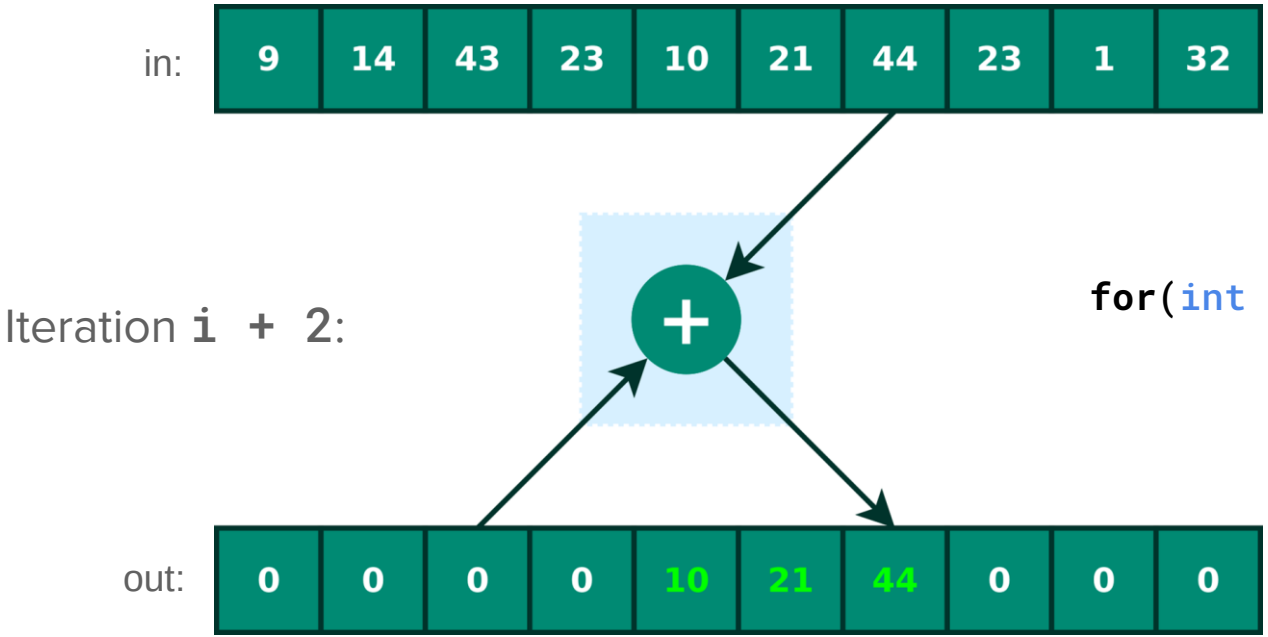
The read of **out[i]** at iteration **i + 4** cannot be scheduled before the **out[i]** is calculated at iteration **i**. This imposes a restriction on the initiation interval.




```
for(int i = 4; i < size; i++)
        out[5] = in[5] + out[1]
```

Pipeline - Loop-carried dependencies

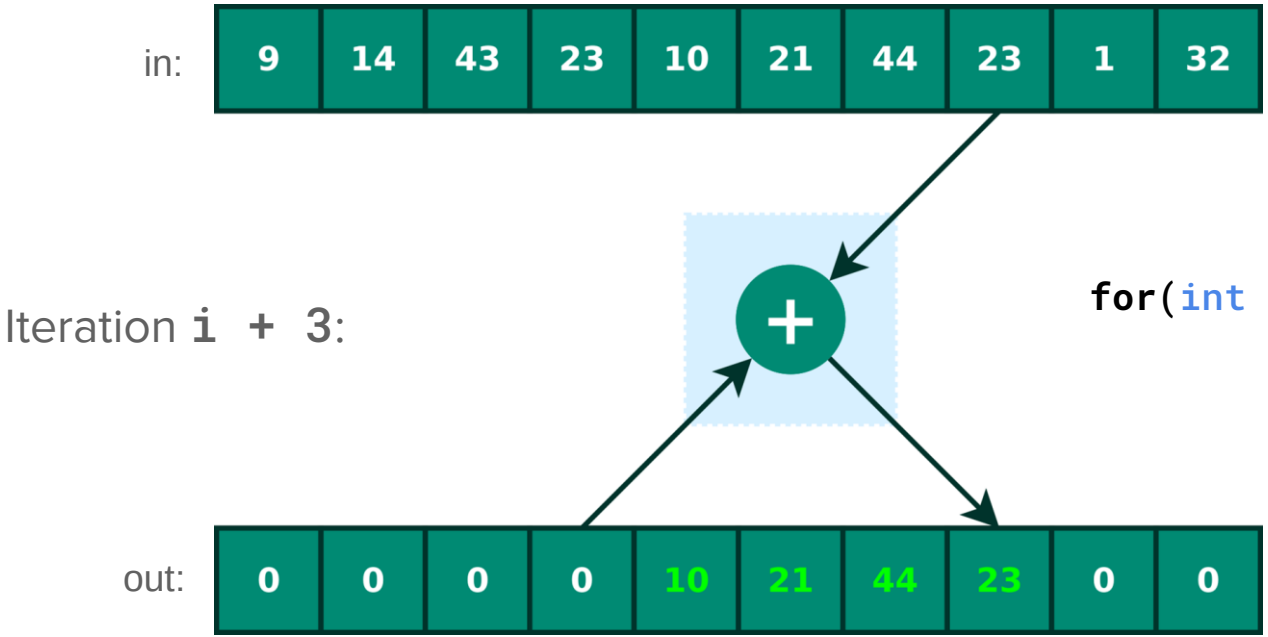
The read of **out[i]** at iteration **i + 4** cannot be scheduled before the **out[i]** is calculated at iteration **i**. This imposes a restriction on the initiation interval.




```
for(int i = 4; i < size; i++)
        out[6] = in[6] + out[2]
```

Pipeline - Loop-carried dependencies

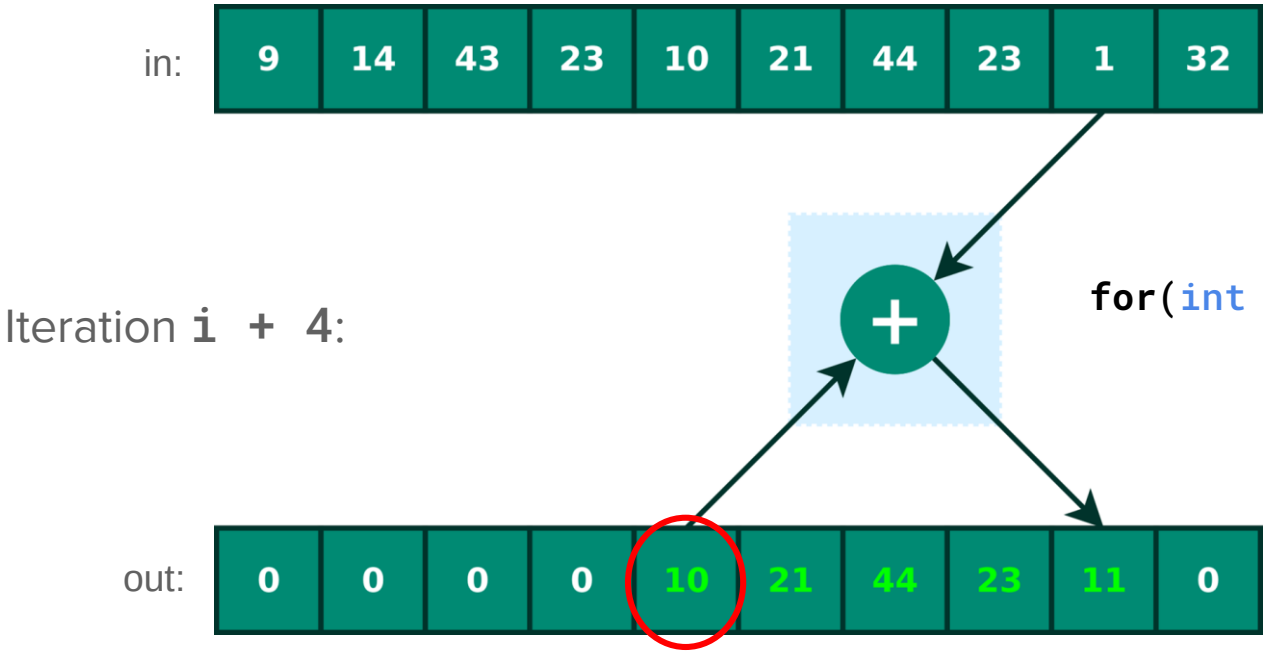
The read of **out[i]** at iteration **i + 4** cannot be scheduled before the **out[i]** is calculated at iteration **i**. This imposes a restriction on the initiation interval.




```
for(int i = 4; i < size; i++)
        out[7] = in[7] + out[3]
```

Pipeline - Loop-carried dependencies

The read of **out[i]** at iteration **i + 4** cannot be scheduled before the **out[i]** is calculated at iteration **i**. This imposes a restriction on the initiation interval.




```
for(int i = 4; i < size; i++)
        out[8] = in[8] + out[4]
```

Pipeline - Loop-carried dependencies

Indeed, Vitis prints several warnings of failed pipelining attempts and keeps increasing the II:

```
WARNING: [HLS 200-880] The II Violation in module 'example'
(loop'LOOP_6_1'): Unable to enforce a carried dependency constraint (II =
1,distance = 4, offset = 1) between bus response on port 'gmem'
(example.cpp:9) and bus request on port 'gmem' (example.cpp:9).
```

The message above repeats for II = 2, 3, 4, 19, 27, 31, 33 and 34. The final II reached is 35:

```
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 35,
Depth = 142, loop 'LOOP_6_1'
```

Pipeline - Loop-carried dependencies

There is no golden recipe for solving recurrence constraints. It could be a combination of:

- Use of on-chip buffers to reduce access to global memory
- Implicitly indicating false dependencies for the HLS compiler
- Code rewrite

Pipeline - Loop-carried dependencies

Another example of recurrence constraint, a simple histogram calculator:

```
extern "C" {

void hist(unsigned char *image, unsigned int *histogram, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=1024
        histogram[image[i]] += 1;
    }
}

}
```

Pipeline - Loop-carried dependencies

Another example of recurrence constraint, a simple histogram calculator:

```
extern "C" {

void hist(unsigned char *image, unsigned int *histogram, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=1024
        histogram[image[i]] += 1;
    }
}

}
```

The **histogram[image[i+1]]** read at iteration **i + 1** cannot be placed before the write at iteration **i** on **histogram[image[i]]**.
There is no way of assuming that the indexes will be different
(i.e. image[i + 1] != image[i])

Pipeline - Loop-carried dependencies

One solution: create a “cache” for histogram elements

- Keep the histogram count of a single pixel intensity on a register

Pipeline - Loop-carried dependencies

Histogram element "cache"

```
void hist(unsigned char *image, unsigned int *histogram, int size) {
    static unsigned int lHist[HISTOGRAM_SIZE];
    unsigned char prevElem = 0;
    unsigned int histCache = 0;

    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=1024
        unsigned char currElem = image[i];

        if(currElem == prevElem) {
            histCache++;
        }
        else {
            lHist[prevElem] = histCache;
            histCache = lHist[currElem] + 1;
            prevElem = currElem;
        }
    }

    lHist[prevElem] = histCache;

    for(unsigned int i = 0; i < HISTOGRAM_SIZE; i++)
        histogram[i] = lHist[i];
}
```

Histogram array is small, we can keep it completely on-chip

Update only the cache value as long pixel intensities are the same

When pixel intensities differ, store cached value, retrieve the next histogram element and increment

Histogram is transferred to off-chip memory

Pipeline - Loop-carried dependencies

II dropped to 2!

- Main reason is the use of local memory

II = 1 is apparently not possible, since the store at iteration **i+1** must be placed after the load at iteration **i** on the **else** block.

However, the **else** block guarantees that prevElem != currElem!

- Meaning that the read and write to/from the histogram can happen at the same time, since they will never point to the same place (efficiency)
- Read and write are independent memory ports

Pipeline - Loop-carried dependencies

Vitis did not detect such false dependency. We can manually inform it to the compiler using a pragma:

```
#pragma HLS DEPENDENCE variable=lHist intra RAW false
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=1024
        unsigned char currElem = image[i];

        if(currElem == prevElem) {
            histCache++;
        }
        else {
            lHist[prevElem] = histCache;
            histCache = lHist[currElem] + 1;
            prevElem = currElem;
        }
    }
```

Pipeline - Loop-carried dependencies

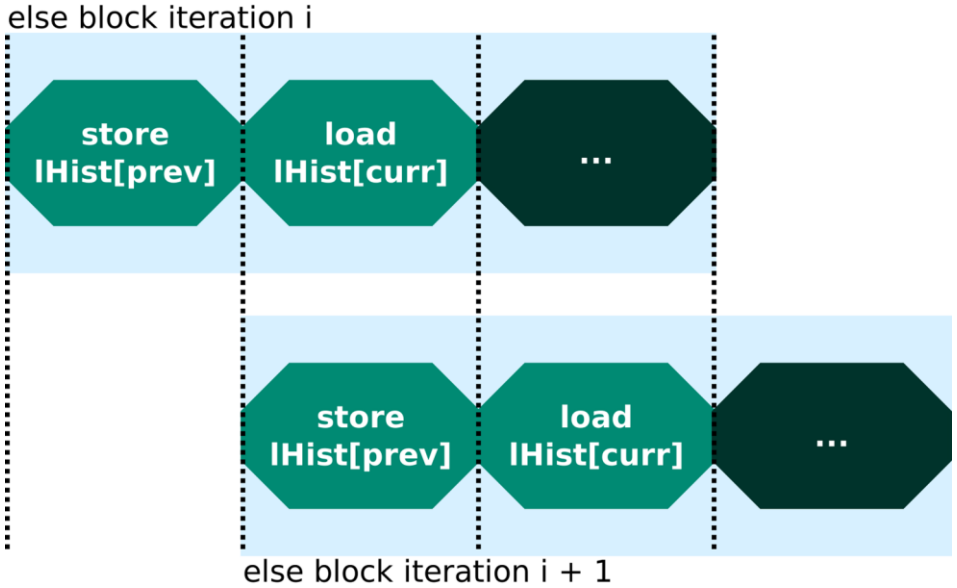
```
#pragma HLS DEPENDENCE variable=lHist intra RAW false
```

- **variable:** defines which variable the dependency information is about
- **intra/inter:** defines if the dependency information is within a loop iteration or between one or more iterations
- **RAW/WAR/WAW:** type of dependence, Read-After-Write, Write-After-Read or Write-After-Write
- **true/false:** informs if the dependency is true or false

Pipeline - Loop-carried dependencies

With this information, Vitis allocates the store and load operations to happen at the same clock cycle for different iterations (i.e. II = 1), since they will never point to the same place:

Please note that marking a true dependency as false may generate incorrect results!



For more information:
https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_optimization_techniques.html#mxr1539734225660

Pipeline - Final Remarks

Pipeline pragmas affect the loop body that they are inserted into.

- All sub-loops are fully unrolled
- If any sub-loop has a variable loop bound, pipeline is not possible

The pragma can also be inserted on the body of a function.

- In this case the function will execute multiple calls overlapped
  - If the function is not called as frequent as its initiation interval, stalls will happen!

Vitis has additional directives that can be used to modify the generated pipeline and mitigate issues such as stalling (e.g. pipeline rewind, pipeline flush). See https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_optimization_techniques.html#kcq1539734224846 for more information.

Loop unrolling - Practical example

Consider our **vadd** example with no pipeline, now with unroll enabled:

```
void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE off
#pragma HLS UNROLL factor=4
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}
```

Loop unrolling - Practical example

The circuit is capable of solving 4 iterations at once (slight reduction in latency)

	Baseline	Pipeline	Unroll			
Latency	307270	8334	304421			
LUTs	2186	1862	4291			
FFs	1361	1391	2356			
DSPs	0	0	0			
BRAMs	2	2	2			

Loop unrolling - Practical example

What happens if the loop bound is not a multiple of the factor?

- Without any condition check, The generated circuit performs out-of-bounds calculations!
- With static loop bounds, Vitis analyses and inserts break conditions where needed
- When the loop bound is variable, a check must be performed after every replicated segment:

```
for(unsigned int i = 0; i < size; i += 4) {
    out[i] = in1[i] + in2[i];
    if((i + 1) >= size) break;
    out[i + 1] = in1[i + 1] + in2[i + 1];
    if((i + 2) >= size) break;
    out[i + 2] = in1[i + 2] + in2[i + 2];
    if((i + 3) >= size) break;
    out[i + 3] = in1[i + 3] + in2[i + 3];
}
```

Parallelism exploration is severely impacted due to the irregular execution flow.

Loop unrolling - Practical example

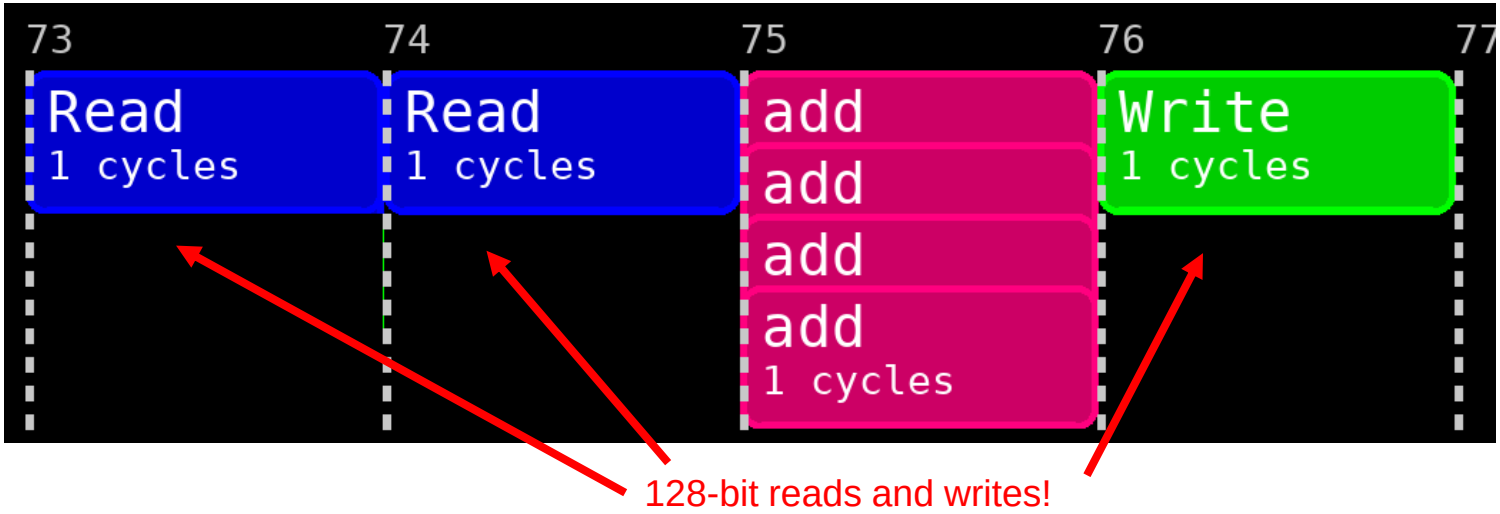
The exit conditions can be removed to reduce the amount of operations and also to open more exploration possibilities with the use of **`skip_exit_check`**.

- In this case, the user must ensure that the runtime loop trip count is always a multiple of the factor.

```
extern "C" {

void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE off
#pragma HLS UNROLL factor=4 skip_exit_check
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}

}
```

Loop unrolling - Practical example

In this case the resulting schedule is significantly simplified:



Vitis detected that the 4 reads of each operand and the four writes could be packed to single 128-bit off-chip requests.

Loop unrolling - Practical example

The latency also has a significant decrease:

	Baseline	Pipeline	Unroll	Unroll (noexit)		
Latency	307270	8334	304421	146433		
LUTs	2186	1862	4291	2554		
FFs	1361	1391	2356	1674		
DSPs	0	0	0	0		
BRAMs	2	2	2	8		

Loop unrolling - Practical example

The latency also has a significant decrease:

	Baseline	Pipeline	Unroll	Unroll (noexit)		
Latency	307270	8334	304421	146433		
LUTs	2186	1862	4291	2554		
FFs	1361	1391	2356	1674		
DSPs	0	0	0	0		

The main reasons are:

- Packed off-chip transactions automatically detected by Vitis
- Less loop condition test logic (trip count reduced from 4096 to 1024)

Loop unrolling - Practical example

Let’s try a larger unroll factor and see how HLS reacts:

```
extern "C" {

void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE off
#pragma HLS UNROLL factor=128 skip_exit_check
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}

}
```

Loop unrolling - Practical example

Vitis packs more loads and stores into the off-chip memory's bandwidth (512 bits):

Loop unrolling - Practical example

HLS takes longer as well. There is performance improvement:

	Baseline	Pipeline	Unroll	Unroll (noexit)	Unroll (large)	
Latency	307270	8334	304421	146433	9505	
LUTs	2186	1862	4291	2554	9329	
FFs	1361	1391	2356	1674	11854	
DSPs	0	0	0	0	0	
BRAMs	2	2	2	8	30	

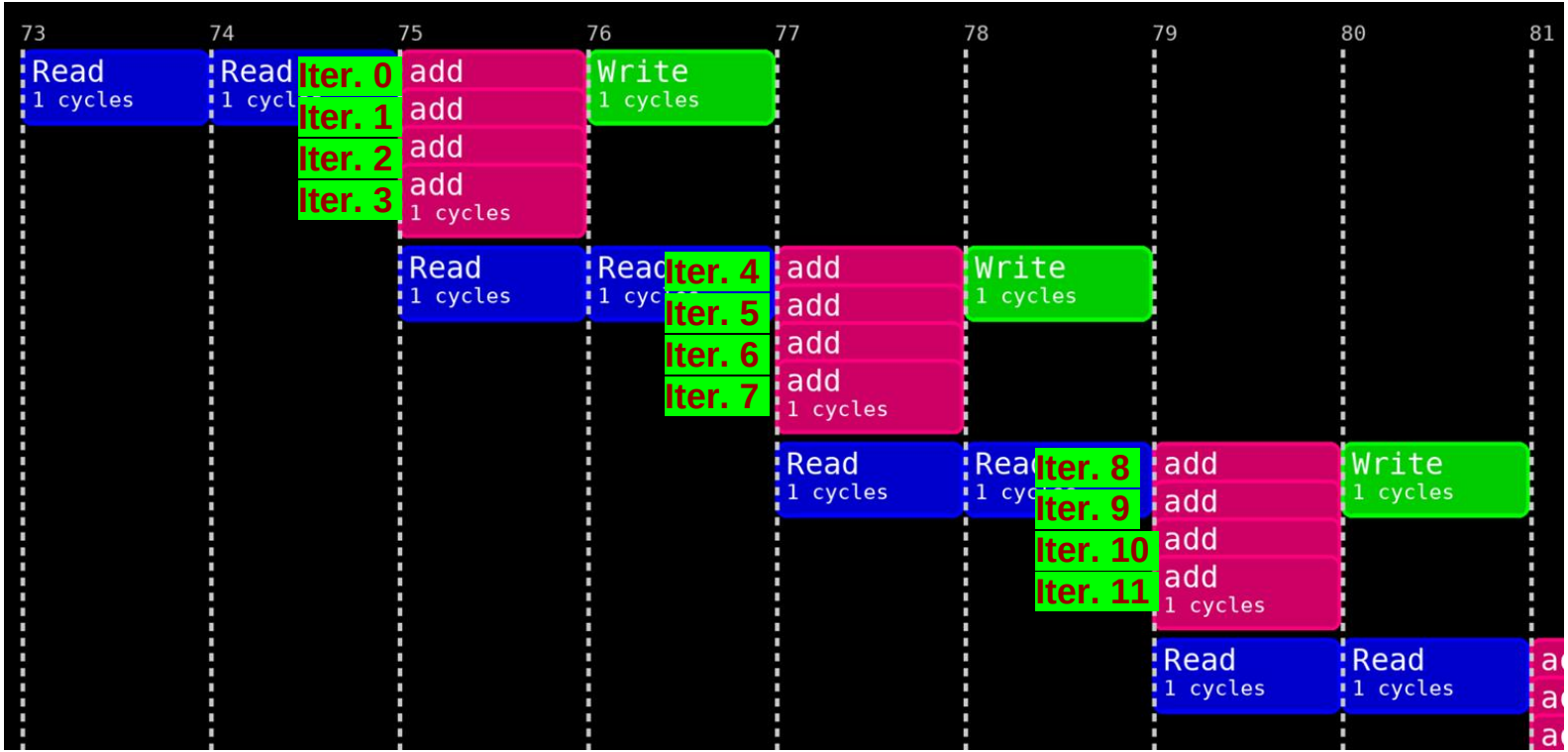
Loop unrolling and pipeline - Best of both worlds

By enabling pipeline, the partially-unrolled loop body will overlap multiple loop iterations. An initiation interval of 2 is reached:

```
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 2,
Depth = 143, loop 'VITIS_LOOP_4_1'
```

```
void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE
#pragma HLS UNROLL factor=4 skip_exit_check
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}
```

Loop unrolling and pipeline - Best of both worlds



Loop optimisations - Summary

Wrapping up the loop explorations on **vadd**:

	Baseline	Pipeline	Unroll	Unroll (noexit)	Unroll (large)	Pipe + unroll
Latency	307270	8334	304421	146433	9505	2190
LUTs	2186	1862	4291	2554	9329	1920
FFs	1361	1391	2356	1674	11854	1767
DSPs	0	0	0	0	0	0
BRAMs	2	2	2	8	30	8

Resource usage are estimates from Vitis.

Loop optimisations - Summary

Pipeline is a very efficient loop optimisation:

- When II is low, good performance is expected
- Does not necessarily incur in resource increase
- Code rewrite might be necessary to reduce the II
- Conservative dependency decisions taken by HLS can kill the design!

In general, unroll should not be used alone:

- Very useful to trigger the load/store data packing from the HLS
- Pipeline can further optimise by overlapping the design
- Resource usage is roughly proportional to unroll factor
- Large factors can incur in great resource usage
- And performance improvement is not guaranteed

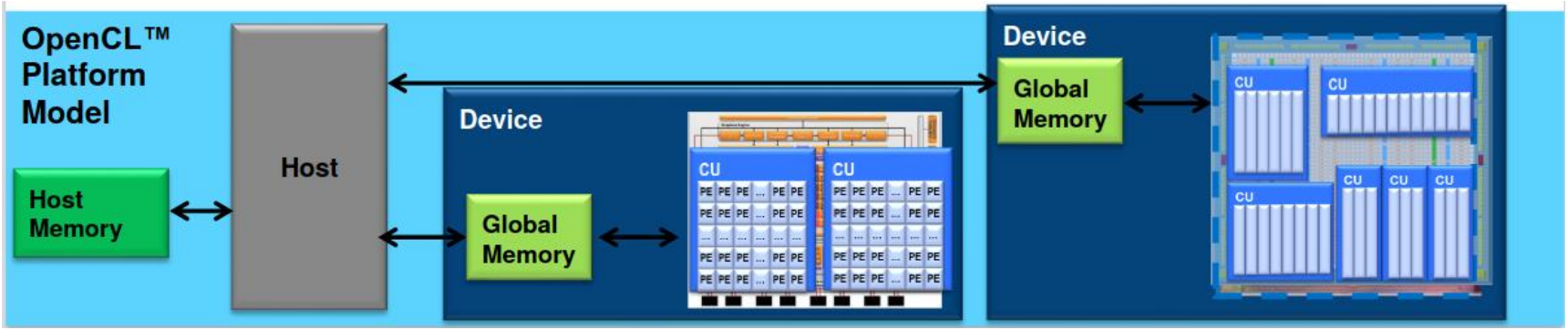
Memory Organisation

Memories and More Memories

- Many memories and levels in a system with FPGAs
  - On-chip
  - Off-chip
- If the FPGA has an on-chip processor
  - Shared memory
  - Processor cached memory
- If the board has an off-chip processor
  - Share off-chip memory (Processor or shared with the FPGA chip)
- If the FPGA board in on a host pc
  - Shared memory
  - Host memory
  - Other devices memories

Memories and More Memories

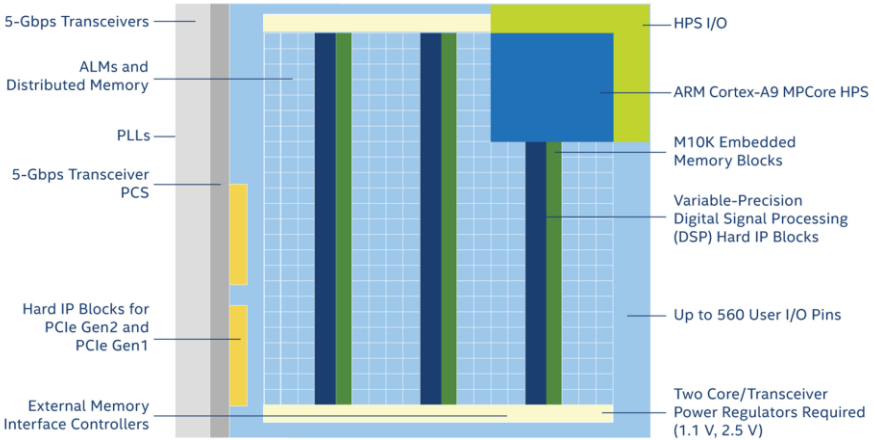
- Transferring data can easily become a bottleneck. Too easily.
- The architecture and available tools depend on the FPGA chip, Board, Vendor, HLS type, Host, …
- The Memories and data management is usually done by the designer.



Example from: “High Level Design Languages for Intel FPGAs”, CNRS DAQ Seminar–Fréjus, November 2018, by francisco.perez@intel.com

On-Chip Memories

- On HLS, they are used to implement local arrays and buffers.
  - Larger memories are created by combining small BRAMS.
  - Each memory has a limited amount of ports.
  - Partitioning memories increase the overall throughput (requires more control hardware and code).



Intel Cyclone V Example. Image from:
https://www.intel.it/content/www/it/it/products/details/fpga/cyclone/v/features.html

Using on-chip memory as buffer

Back to our first recurrence example, an II of 35 was reached due to the recurrence between **out[i - DISTANCE]** and **out[i]**.

```
#define DISTANCE 4

extern "C" {

void example(unsigned int *in, unsigned int *out, int size) {
    for(unsigned int i = DISTANCE; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4092
        out[i] = in[i] + out[i - DISTANCE];
    }
}

}
```

Using on-chip memory as buffer

We can use a shift register to eliminate the **out[i - DISTANCE]** read

- Shift registers acts as FIFOs with the addition of being random-access
- In our case, we will make the shift register store the N = DISTANCE most recent values calculated of **out**

Vitis infers a shift register when a pattern similar to the following Is detected:

```
int sh[N];

/* Computation loop */
for(...) {
    for(int i = 0; i < N - 1; i++)
        sh[i] = sh[i + 1];
    sh[N - 1] = /* new value to the register goes here */;

    /* a random access to the register happens here */
}
```

Using on-chip memory as buffer

In our code:

```
#define DISTANCE 4

extern "C" {

void example(unsigned int *in, unsigned int *out, int size) {
    static unsigned int shiftReg[DISTANCE];

    for(unsigned int i = DISTANCE; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4092
        int _tmp = in[i] + shiftReg[0];
        out[i] = _tmp;

        // These statements below are converted to a shift register
        for(int j = 0; j < DISTANCE - 1; j++)
            shiftReg[j] = shiftReg[j+1];
        shiftReg[DISTANCE - 1] = _tmp;
    }
}

}
```

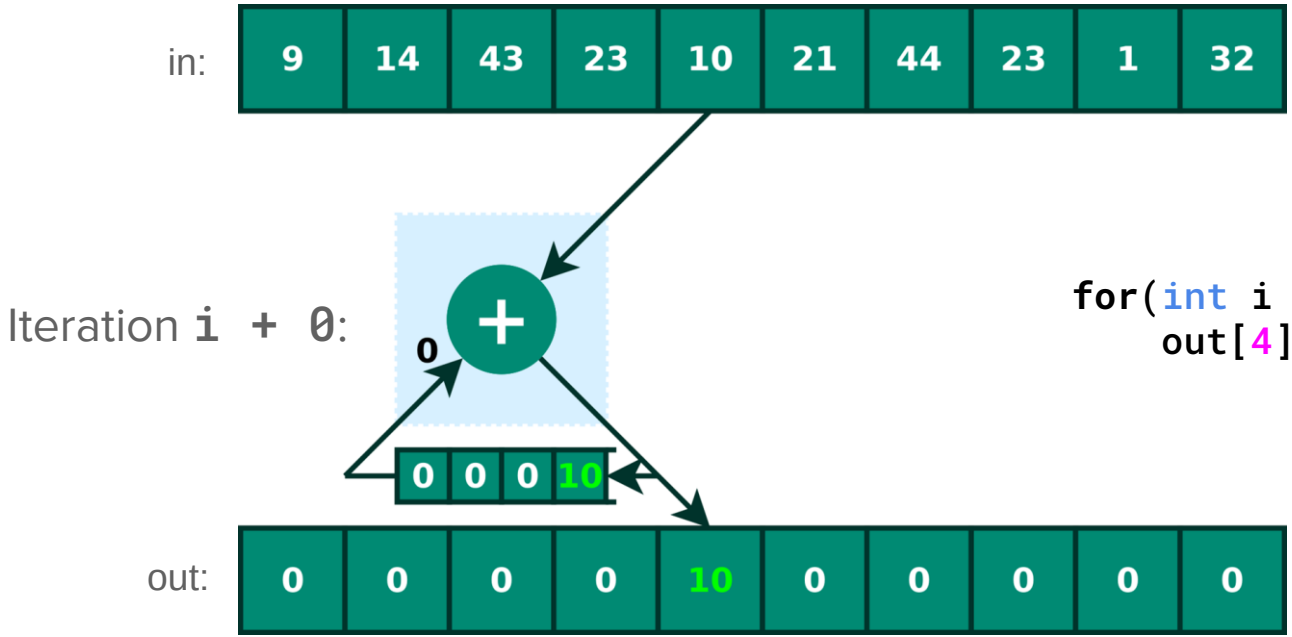
Using on-chip memory as buffer

In our code:

```
#define DISTANCE 4

extern "C" {

void example(unsigned int *in, unsigned int *out, int size) {
    static unsigned int shiftReg[DISTANCE];

    for(unsigned int i = DISTANCE; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4092
        int _tmp = in[i] + shiftReg[0];
        out[i] = _tmp;

        // These statements below are converted to a shift register
        for(int j = 0; j < DISTANCE - 1; j++)
            shiftReg[j] = shiftReg[j+1];
        shiftReg[DISTANCE - 1] = _tmp;
    }
}

}
```

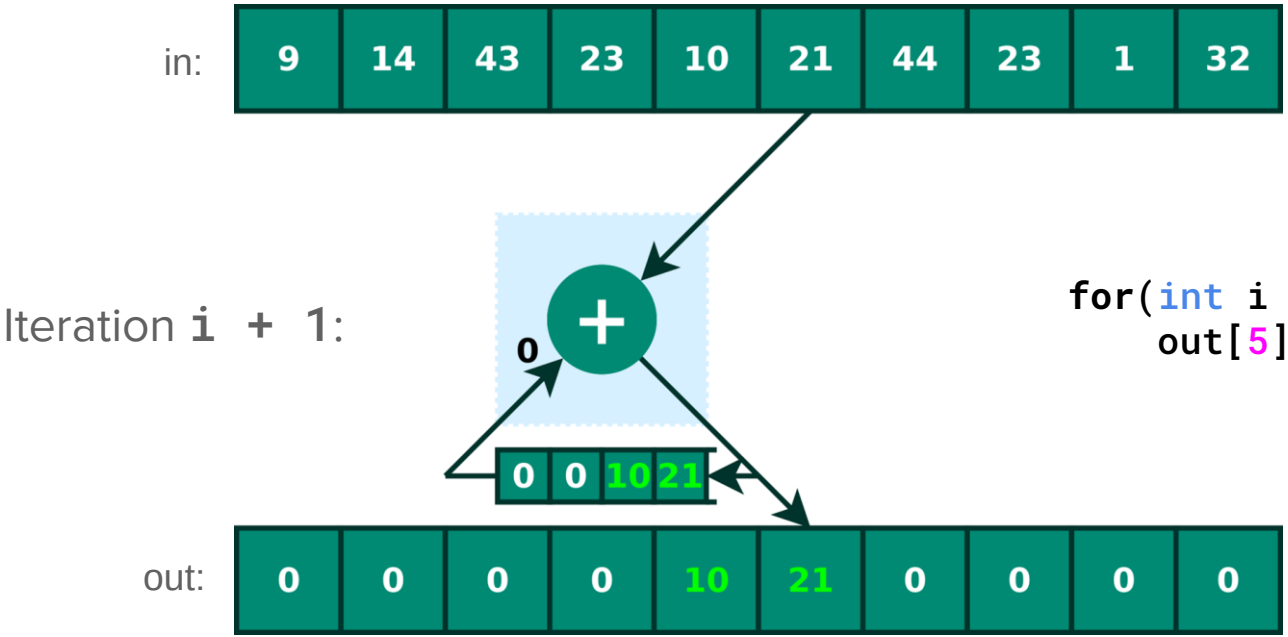
The **out[i - DISTANCE]** read was replaced by the **shiftReg[0]** on-chip read

Using on-chip memory as buffer



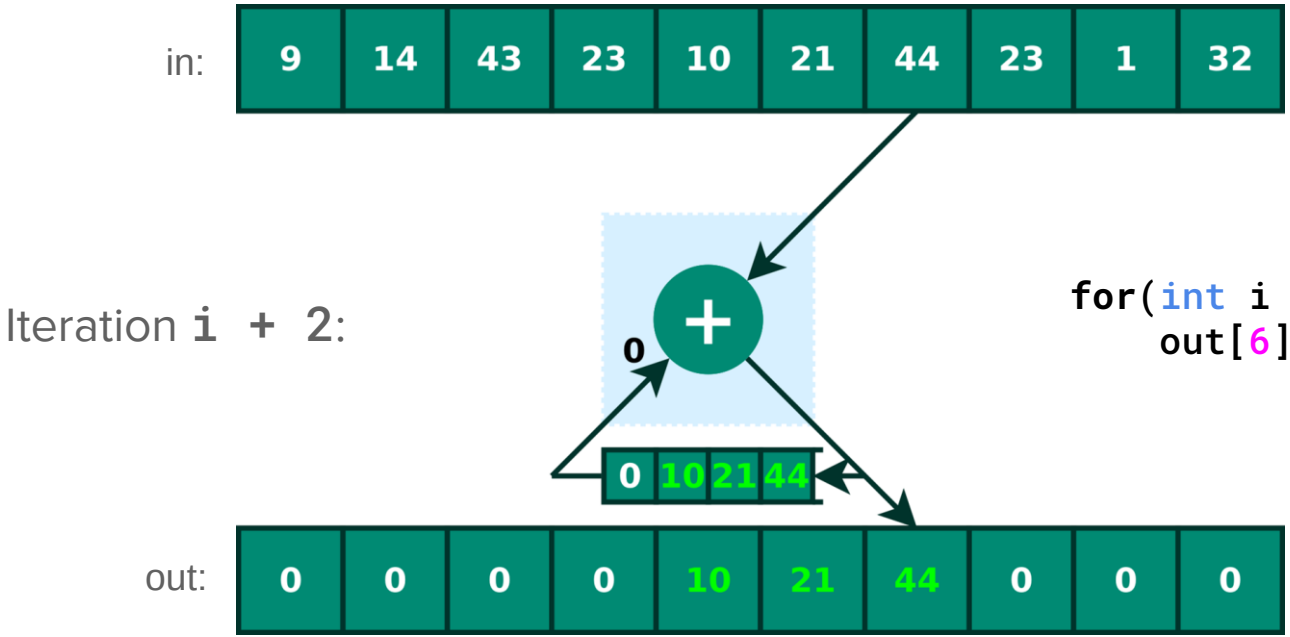

```
for(int i = 4; i < size; i++)
    out[4] = in[4] + shiftReg[0]
```

Using on-chip memory as buffer



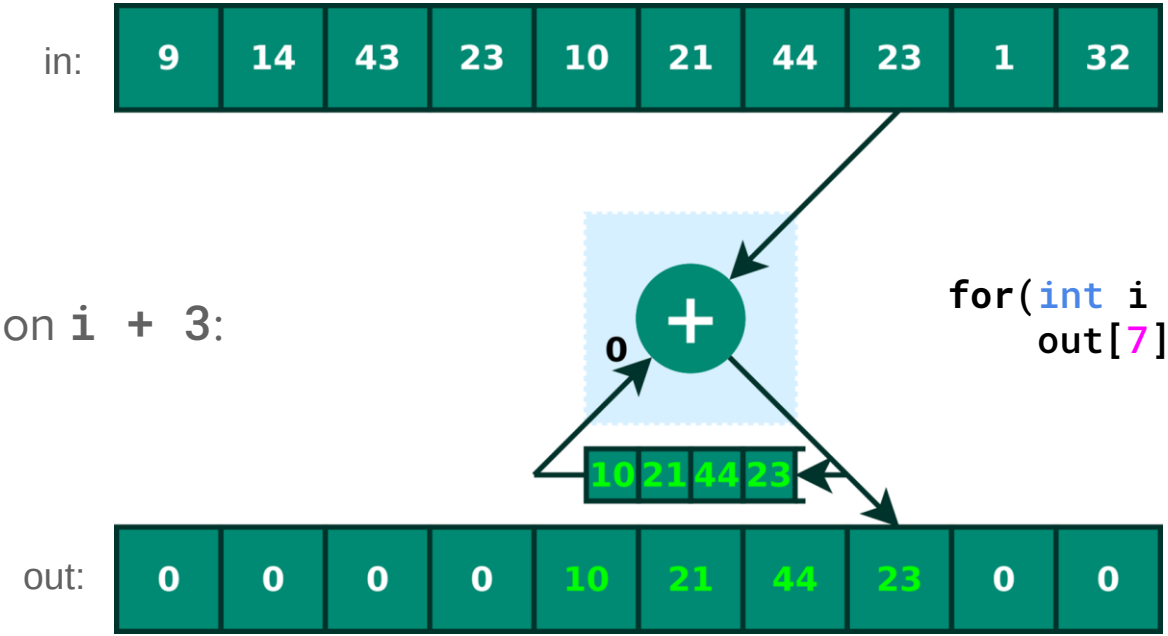

```
for(int i = 4; i < size; i++)
    out[5] = in[5] + shiftReg[0]
```

Using on-chip memory as buffer




```
for(int i = 4; i < size; i++)
    out[6] = in[6] + shiftReg[0]
```

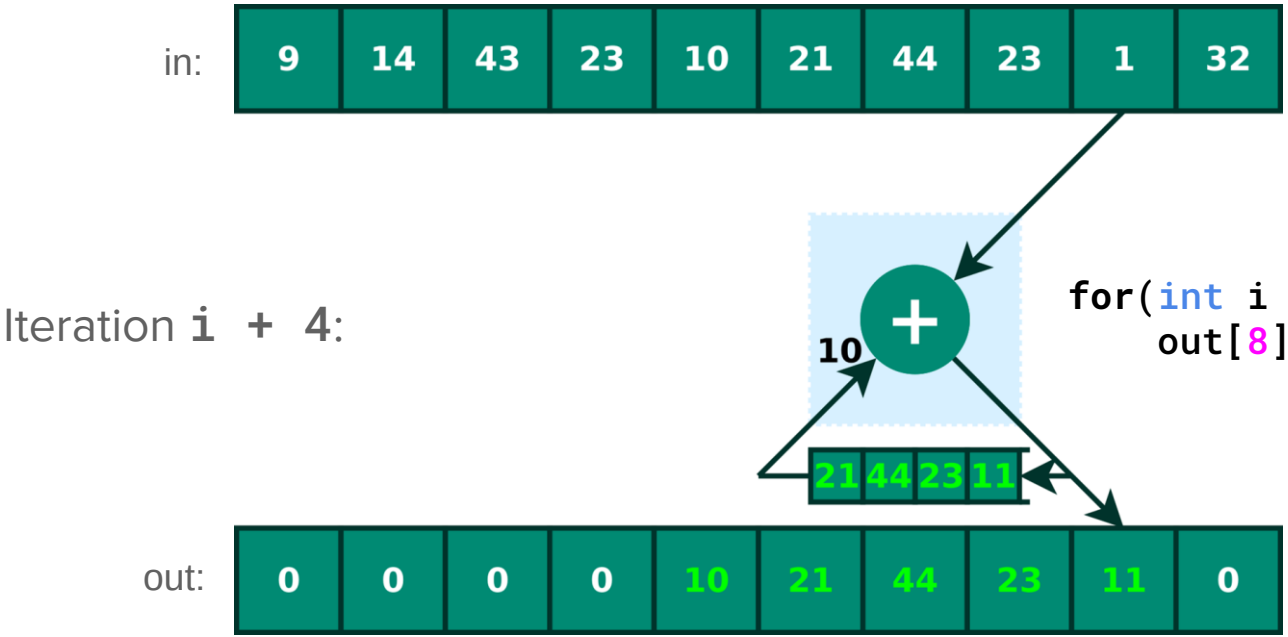
Using on-chip memory as buffer



Iteration `i + 3`:

```
for(int i = 4; i < size; i++)
    out[7] = in[7] + shiftReg[0]
```

Using on-chip memory as buffer




```
for(int i = 4; i < size; i++)
    out[8] = in[8] + shiftReg[0]
```

Using on-chip memory as buffer

Our II dropped from 35 to 1:

```
+---------+---------+----------+-----------+-----+------+---------+
|  Latency (cycles) |  Latency (absolute)  |  Interval  | Pipeline|
|   min   |   max   |    min   |    max    | min |  max |   Type  |
+---------+---------+----------+-----------+-----+------+---------+
|        1|     4233|  6.667 ns|  28.221 us|    2|  4234|     none|
+---------+---------+----------+-----------+-----+------+---------+
+-----------+---------+---------+----------+-----------+-----------+----------
+----------+
|           |  Latency (cycles) | Iteration|  Initiation Interval  |   Trip
|           |
| Loop Name |   min   |   max   |  Latency |  achieved |   target  |   Count
| Pipelined|
+-----------+---------+---------+----------+-----------+-----------+----------
+----------+
|- LOOP_8_1 |        3|     4094|         4|          1|          1|  1 ~
```

Array partition

Now consider a very simple stencil computation

- **out[i]** is calculated by summing the inputs **in1** in the range **i** to **i** + **STENCIL_SIZE - 1**

```
void stencil(int *in1, int *out) {
    for(unsigned int i = 0; i < DATA_SIZE; i++) {
        int acc = 0;

        for(unsigned int j = 0; j < STENCIL_SIZE; j++)
            acc += in1[i + j];

        out[i] = acc;
    }
}
```

in1 is read 16 times here!
(STENCIL_SIZE = 16)

Array partition

The pipeline generated has an II of STENCIL_SIZE = 16, constrained by the 16 reads performed at every top-loop iteration.

- In general stencils can be buffered in a sliding window fashion. We can use a similar approach as the last example and generate a shift register with 16 elements

```
for(unsigned int i = 0; i < DATA_SIZE; i++) {
    for(unsigned int j = 0; j < (STENCIL_SIZE - 1); j++)
        buffer[j] = buffer[j + 1];
    buffer[STENCIL_SIZE - 1] = in1[i + STENCIL_SIZE - 1];

    int acc = 0;
    for(unsigned int j = 0; j < STENCIL_SIZE; j++)
        acc += buffer[j];
    out[i] = acc;
}
```

Only one off-chip read per loop iteration

Array partition

The II went from 16 to 145! Vitis reports several warnings about failed attempts to pipeline with lower II:

```
WARNING: [HLS 200-880] The II Violation in module 'stencil' (loop 'LOOP_12_2'):
Unable to enforce a carried dependence constraint (II = 143, distance = 1,
offset = 1) between 'store' operation (stencil.cpp:22) of variable
'trunc_ln22_1', stencil.cpp:22 on array 'buffer', stencil.cpp:7 and 'load'
operation (stencil.cpp:16) on array 'buffer', stencil.cpp:7.
```

Right after, another useful warning is reported:

```
WARNING: [HLS 200-885] Unable to schedule 'load' operation (stencil.cpp:16) on
array 'buffer', stencil.cpp:7 due to limited memory ports. Please consider
using a memory core with more ports or partitioning the array 'buffer'.
```

Array partition

Vitis did not actually infer a shift register here. Instead, it inferred a dual-ported on-chip memory for **buffer**. An II of 1 is not possible, since the shift procedure requires several loads and writes per loop iteration.

As warned by Vitis, partitioning the array **buffer** can increase parallelism and thus potentially reduce II.

Since buffer is small in size, we can perform a complete partitioning, where all elements are mapped to separate registers

- Every element can be accessed and written at the same clock cycle

Array partition

We use **#pragma HLS ARRAY_PARTITION** to indicate that **buffer** should be completely partitioned into separate registers:

```
int buffer[STENCIL_SIZE];
#pragma HLS ARRAY_PARTITION variable=buffer complete
```

With the freedom to read and write to any element in **buffer** at every clock cycle, Vitis is finally able to schedule a pipeline with II of 1:

```
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1,
Depth = 142, loop VITIS_LOOP_13_2'
```

Array partition

Comparing the three approaches:

The buffered + partitioned solution is faster AND also consumes less resources.

Our manual code modifications and pragmas applied prepared a “terrain” where the HLS compiler could easily schedule the loop and generate a module with efficient routing.
Important to note that in our case the array is small, being suitable for complete partitioning!

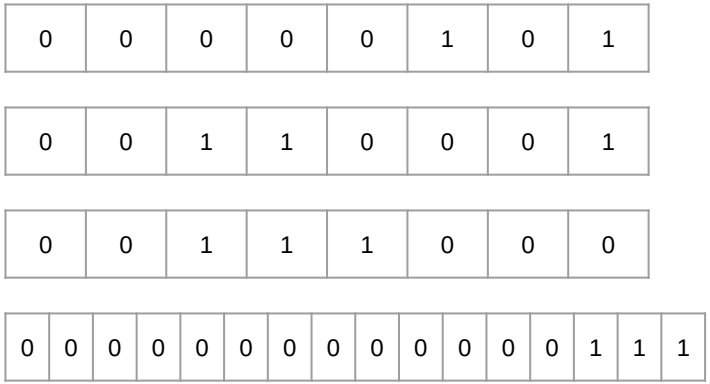
	Baseline	Buffered	Buffered + Partitioned
Latency	16526	15708	1254
LUTs	39766	3099	2394
FFs	12890	1694	2799
DSPs	0	0	0
BRAMs	30	2	2

Resource usage are estimates from Vitis.

Data Types and Representations

Data Types and Representations

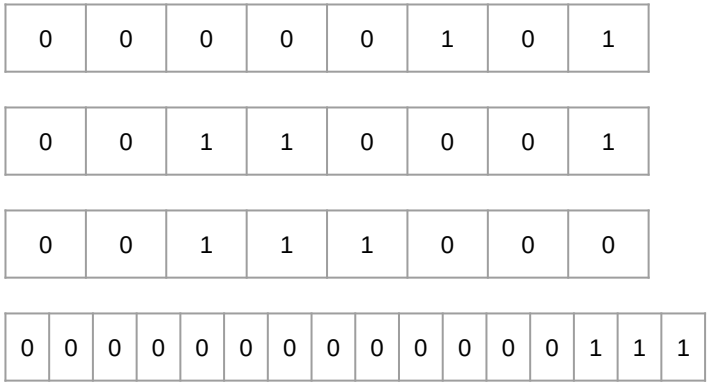
- Data representation on hardware
  - Everything is an array of bits:
    - short short Int a = 5
    - short int b = 7
    - float c = 3.5
    - char d = ‘a’



Data Types and Representations

- Data representation on hardware
  - Everything is an array of bits:
    - short short Int a = 5
    - short int b = 7
    - float c = 3.5
    - char d = ‘a’

Data type is the meaning we give to the data.



Data representation is how the data stored in an array of bits is interpreted.

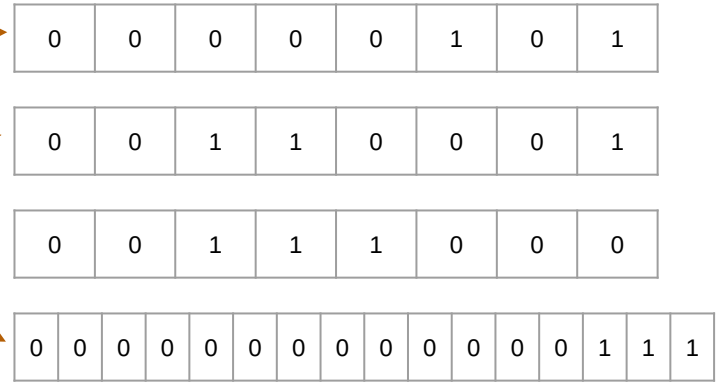
Data Types and Representations

- Data types:
  - Commonly found in high-level code, help the compiler (and the user) to understand how to treat the data.
- For Example:
  - int, float, double, char, etc.
- Data representation:
  - It is how the data is represented in a binary format.
  - Usually only fixed or floating point.

Data Types and Representations

- Data representation on hardware
  - Everything is an array of bits:
    - short short Int a = 5
    - short int b = 7
    - float c = 3.5
    - char d = ‘a’

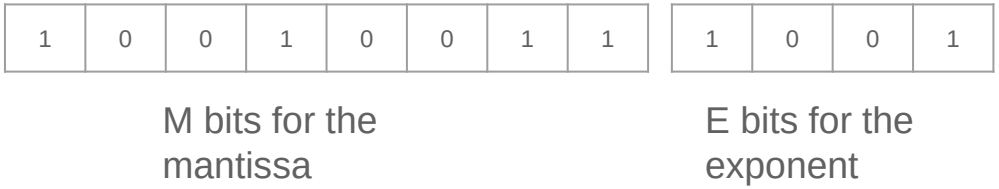
Some information about the representation is usually encoded in the type.



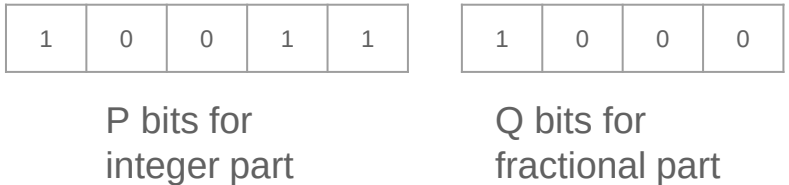
Data representation is how the data is stored in an array of bits.

Fixed vs Floating Point Representations

- Both are used to represent Real values:
- Floating Point:
  - Consider the number in scientific notation: $\pm X.10^{\pm y}$
  - Defined as the format “float M.E”
  - float8.4 a = -3.5



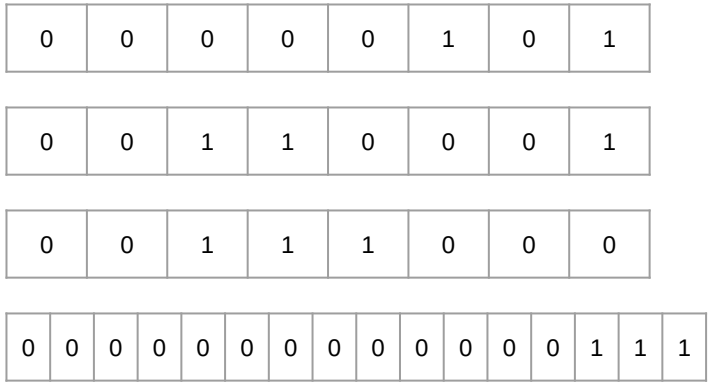
- Fixed Point:
  - A straightforward representation
  - Defined as the format “fixed P.Q”
  - fixed5.3 a = -3.5



Data Types and Representations

- Data representation on hardware
  - Everything is an array of bits:
    - short short Int a = 5
    - short int b = 7
    - float c = 3.5
    - char d = ‘a’

However, we can use different representations for the same data type. For example, here we have the value “3.5” represented as a “4.4 fixed point”.



Data representation is how the data is stored in an array of bits.

Fixed vs Floating Point Representations

- Both are used to represent Real values:

	Fixed-Point	Floating Point
Pros	Arithmetic is the same as for int types. It uses less hardware, and it **usually** takes less clock cycles.	The precision adapts with the magnitude of the number, making it less prone to precision errors. Can represent larger number.
Cons	The precision is fixed and needs to be tuned for the algorithms, which is not a trivial task.	It costs a considerable amount of hardware resources and clock cycles.

A Quick Example

```
1  #include "compute.h"
2
3  void compute(data_t din, data_t *dout){
4      static data_t acc=0;
5      acc = (acc + din)/(3);
6      *dout = acc;
7      return;
8  }
```

```
#include "ap_fixed.h"
typedef ap_fixed<16, 16> data_t;
void compute(data_t din, data_t *dout
```

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	1	1	1	none

Resource Usage

	Verilog
SLICE	0
LUT	77
FF	19
DSP	1
BRAM	0
SRL	0

```
1  typedef float data_t;
2  void compute(data_t din, data_t *dout);
```

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
12	12	12	12	none

Resource Usage

	Verilog
SLICE	0
LUT	955
FF	527
DSP	2
BRAM	0
SRL	32

Conversion from Floating to Fixed Point

- The goal is to estimate how many bits a fixed-point representation need for not compromising the precision.
- If the minimum and maximum values for all inputs are known, and the code is completely static, the number of bits can be calculated and propagated through the code variables

variables	Min value	Max value	q	p
a	-1.5	3.75	3	2
b	0	5	3	0
C = a+b			max(qa, qb)+1	max(pa, pb)+1
D = a*b			qa+qb	pa+pb
for i=1:100 E *= a			…. a lot	.... a lot

Conversion from Floating to Fixed Point

- The goal is to estimate how many bits a fixed-point representation need for not compromising the precision.
- If the minimum and maximum values for all inputs are known, and the code is completely static, the number of bits can be calculated and propagated through the code variables.
    - Loops may make the propagation impractical, since the number of required bits increase too much.

Conversion from Floating to Fixed Point

- The goal is to estimate how many bits a fixed-point representation need for not compromising the precision.
- More general approaches are a topic of research.
  - Saldanha, L., & Lysecky, R. (2009). Float-to-fixed and fixed-to-float hardware converters for rapid hardware/software partitioning of floating point software applications to static and dynamic fixed point coprocessors. *Design Automation for Embedded Systems*, *13*(3), 139-157.
  - Aamodt, T. M., & Chow, P. (2008). Compile-time and instruction-set methods for improving floating-to fixed-point conversion accuracy. *ACM Transactions on Embedded Computing Systems (TECS)*, *7*(3), 1-27.
  - Hopkins, M., Mikaitis, M., Lester, D. R., & Furber, S. (2020). Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. Philosophical Transactions of the Royal Society A, 378(2166), 20190052.

DSPs for Floating Point processing

DSPs are another way to save hardware resources for floating point arithmetic. They are hard builded on modern FPGAs!

```
1  #include "compute.h"
2
3  void compute(data_t din, data_t *dout){
4      static data_t acc=0;
5
6      data_t e;
7      #pragma HLS RESOURCE variable=e core=FRSqrt_nodsp
8      e = sqrt(din);
9
10     #pragma HLS RESOURCE variable=acc core=FAddSub_nodsp
11     acc = acc + e;
12     *dout = acc;
13     return;
14 }
```

Resource Usage

	Verilog
SLICE	0
LUT	761
FF	485
DSP	0
BRAM	0
SRL	11

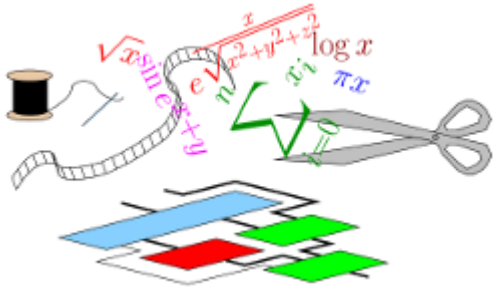
```
1  #include "compute.h"
2
3  void compute(data_t din, data_t *dout){
4      static data_t acc=0;
5
6      data_t e;
7      #pragma HLS RESOURCE variable=e core=FRSqrt_fulldsp
8      e = sqrt(din);
9
10     #pragma HLS RESOURCE variable=acc core=FAddSub_fulldsp
11     acc = acc + e;
12     *dout = acc;
13     return;
14 }
```

Resource Usage

	Verilog
SLICE	0
LUT	640
FF	548
DSP	2
BRAM	0
SRL	11

Creating Customised Floating Point FUs

- FloPoCo allows the creation of customised floating point units which are more efficient then combining basic floating points operations.
  - E.g. $\sqrt{x^2 + y^2}$
- It helps to satisfy the design constraints using floating point, but requires some extra effort on designing.
- http://flopoco.gforge.inria.fr/



De Dinechin, F., Pasca, B., & Normale, E. (2011). Custom arithmetic datapath design for FPGAs using the FloPoCo core generator. *Design & Test of Computers, IEEE*, *28*(4), 18-27.

Arbitrary precision data types - Practical example

Back to our simple pipelined vadd with II = 2. Could we go II = 1?

Let's suppose that our input values do not exceed 16 bits.

- We could pack the operands **in1[i]** and **in2[i]** in a single **int** element of 32 bits and only perform one memory read per iteration.

```
extern "C" {

void vadd(unsigned int *in1, unsigned int *in2, unsigned int *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT max=4096
        out[i] = in1[i] + in2[i];
    }
}

}
```

Arbitrary precision data types - Practical example

Due to the flexible nature of FPGAs, HLS allows us to specify variable types with arbitrary precision.

- Generated hardware becomes more compact, since all related computations are designed to work with that specific bit-width
- Suitable for highly-quantised machine learning applications

Vitis provides two C++ templates -- **ap_int<N>** and **ap_fixed<N>** -- to support integer and fixed-point arbitrary precision types.

Arbitrary precision data types - Practical example

```
#include "ap_int.h"

struct two_apint16_t {
    ap_int<16> in1;
    ap_int<16> in2;
};

extern "C" {

void vadd(two_apint16_t *in1, ap_int<16> *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT max=4096
        two_apint16_t tmp = in1[i];

        out[i] = tmp.in1 + tmp.in2;
    }
}

}
```

Arbitrary precision data types - Practical example

```
#include "ap_int.h"

struct two_apint16_t {
    ap_int<16> in1;
    ap_int<16> in2;
};

extern "C" {

void vadd(two_apint16_t *in1, ap_int<16> *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT max=4096
        two_apint16_t tmp = in1[i];

        out[i] = tmp.in1 + tmp.in2;
    }
}

}
```

Only one load and one store per loop iteration!

Arbitrary precision data types - Practical example

```
#include "ap_int.h"

struct two_apint16_t {
    ap_int<16> in1;
    ap_int<16> in2;
};

extern "C" {

void vadd(two_apint16_t *in1, ap_int<16> *out, int size) {
    for(unsigned int i = 0; i < size; i++) {
#pragma HLS LOOP_TRIPCOUNT max=4096
        two_apint16_t tmp = in1[i];

        out[i] = tmp.in1 + tmp.in2;
    }
}

}
```

The add FU is adjusted to work on 16-bit instead of 32 (could be other “unorthodox” values as well)

Arbitrary precision data types - Practical example

Compared to our other **vadd** results:

II of 1 is reached!

Good performance, low resource usage

We could reduce the latency even more by unrolling the loop

	Baseline	Pipeline	Unroll (large)	Pipe + unroll	Pipe + 16-bit
Latency	307270	8334	9505	2190	4238
LUTs	2186	1862	9329	1920	1789
FFs	1361	1391	11854	1767	1174
DSPs	0	0	0	0	0
BRAMs	2	2	30	8	2

Resource usage are estimates from Vitis.

Dataflow optimisation

Types of Parallelism

- Parallelism can be explored in many level, specially when considering not only the FPGA chip:
  - Instruction/Data Parallelism.
    - Processing instructions concurrently because they are not dependent.
    - Loop pipeline is an way to explore instruction level parallelism.
    - This type of parallelism is within a module/kernel.
  - Task Parallelism
    - Separate modules executing in parallel. Can be multiple instances of the same module.
  - Pipeline Parallelism
    - Separate modules with dependencies running in a pipeline manner.

Task Parallelism

- Dataflow processing is similar to “pipelining instructions in a loop”. Here, the operands can be:
  - Functions
  - Loops (HLS documentation might be deceiving)

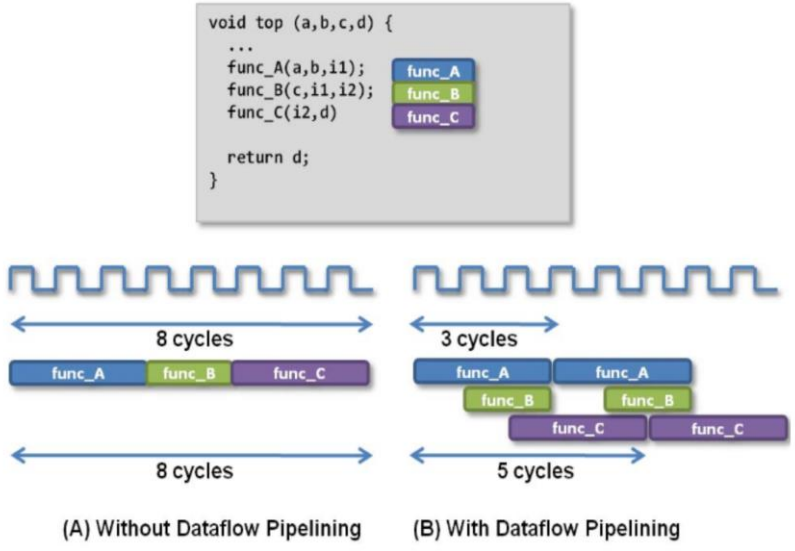


Image from: https://xilinx.github.io/Vitis-Tutorials/2020-1/docs/convolution-tutorial/dataflow.html

Dataflow optimisation

Vitis also provides a dataflow feature, where the control system of a group of subroutines is removed and the computation flow is totally data driven

- Inputs and outputs of the subroutines are connected using FIFOs or similar structures.

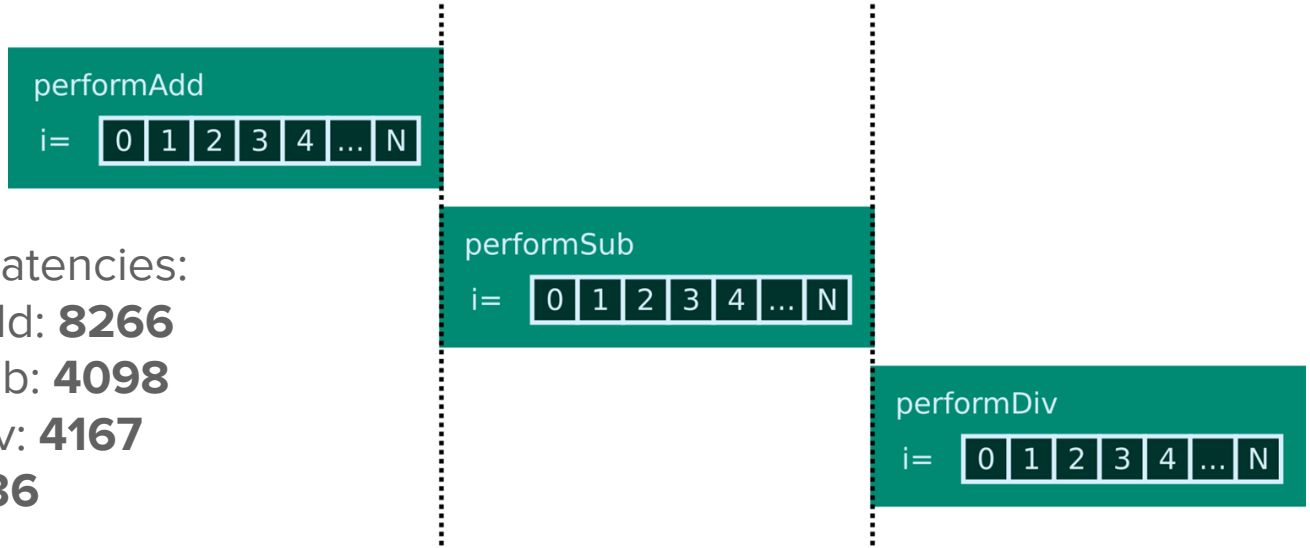
Consider the following code:

performAdd, **performSub** and **performDiv** are non-inlined subroutines

```
void dflow(int *in1, int *in2, int *out) {
    int tmp1[DATA_SIZE];
    int tmp2[DATA_SIZE];

    performAdd(in1, in2, tmp1);
    performSub(tmp1, tmp2);
    performDiv(tmp2, out);
}
```

Dataflow optimisation

Without dataflow, the subroutines are serially executed



Reported latencies:
performAdd: **8266**
performSub: **4098**
performDiv: **4167**
Final: **16536**

Dataflow optimisation

For a successful dataflow scheduling, Vitis recommends to write the dataflow regions in a “canonical” form, for example:

- For dataflow within loops, the index variable should increase monotonically
- Feedback from one subroutine to a previous one should be avoided
- Conditional subroutine execution can also affect performance

In our case, code rewrite was required so that each subroutine communicated using a stream interface instead of temporary buffers like **tmp1** and **tmp2**.

- Vitis provides an **hls::stream<>** C++ construct that works similarly to usual C++ streams (e.g. std::cout)

See https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_optimization_techniques.html#bmx1539734225930 for more details

Dataflow optimisation

```
for(unsigned int i = 0; i < DATA_SIZE; i++) {
    out1 << in1[i];
    out2 << in2[i];
```

```
for(unsigned int i = 0; i < DATA_SIZE; i++)
    out << in3.read() + in4.read();
```

```
for(unsigned int i = 0; i < DATA_SIZE; i++)
    out << in5.read() - 0xcafe;
```

```
for(unsigned int i = 0; i < DATA_SIZE; i++)
    out << in6.read() / 552;
```

```
for(unsigned int i = 0; i < DATA_SIZE; i++)
    in7 >> out[i];
```

```
void dflow(int *in1, int *in2, int *out) {
#pragma HLS DATAFLOW
    hls::stream<int> tmp1("tmp1_stream");
    hls::stream<int> tmp2("tmp2_stream");
    hls::stream<int> tmp3("tmp3_stream");
    hls::stream<int> tmp4("tmp4_stream");
    hls::stream<int> tmp5("tmp5_stream");

    readData(in1, in2, tmp1, tmp2);
    performAdd(tmp1, tmp2, tmp3);
    performSub(tmp3, tmp4);
    performDiv(tmp4, tmp5);
    writeData(tmp5, out);
}
```

Dataflow optimisation

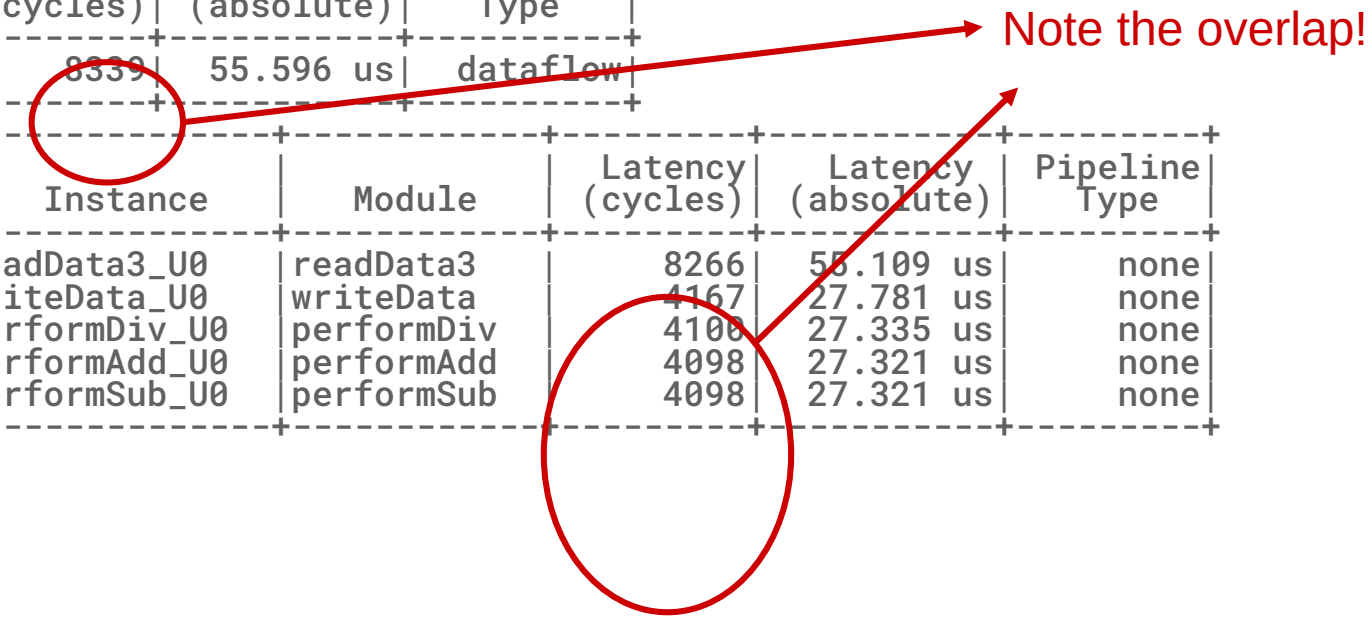
Latency (cycles)	Latency (absolute)	Pipeline Type
8339	55.596 us	dataflow

Instance	Module	Latency (cycles)	Latency (absolute)	Pipeline Type
readData3_U0	readData3	8266	55.109 us	none
writeData_U0	writeData	4167	27.781 us	none
performDiv_U0	performDiv	4100	27.335 us	none
performAdd_U0	performAdd	4098	27.321 us	none
performSub_U0	performSub	4098	27.321 us	none

Dataflow optimisation

Latency (cycles)	Latency (absolute)	Pipeline Type
8339	55.596 us	dataflow

Instance	Module	Latency (cycles)	Latency (absolute)	Pipeline Type
readData3_U0	readData3	8266	55.109 us	none
writeData_U0	writeData	4167	27.781 us	none
performDiv_U0	performDiv	4100	27.335 us	none
performAdd_U0	performAdd	4098	27.321 us	none
performSub_U0	performSub	4098	27.321 us	none



Final remarks

Final Remarks

- There is no golden rule that apply for every input code
- Optimisation usually consists of:
  - Reducing or eliminating recurrence constraints in loops
  - Uncoupling pipelined loops from uncoalesced off-chip load/stores (use on-chip buffers instead)
    - Coalesced off-chip load/stores are still acceptable due to burst
  - Applying pipeline directive to loops
  - Applying unroll directives to increase the parallelism inside pipeline iterations
  - Apply array partitioning to increase on-chip port availability
    - A highly efficient unrolled+pipelined compute datapath usually requires a parallel access to arrays
  - Apply dataflow optimisation for subroutines that can overlap
  - Use arbitrary-precision data types to reduce memory bandwidth and reduce the computation resource footprint

Thank you for your attention!