High Level Synthesis for AMD FPGAs

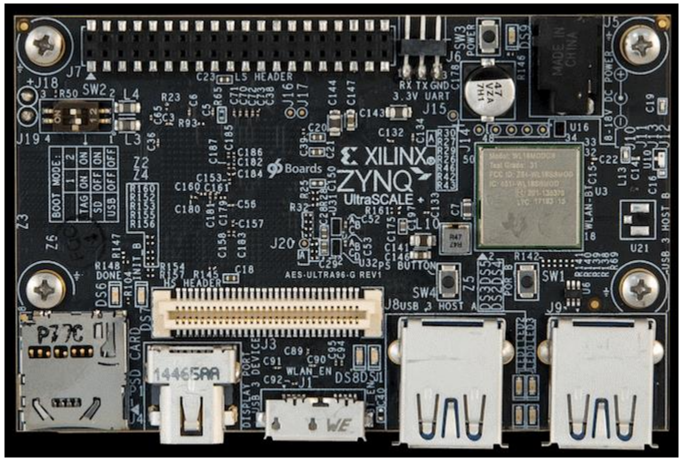
Gianluca Brilli, Paolo Burgio
gianluca.brilli@unimore.it
paolo.burgio@unimore.it

Outline

- Overview of the main AMD FPGA design tools;
  - We will outline what is the standard flow to build an FPGA bitstream;
- How to create an HW accelerator starting from a C/C++ implementation using HLS tools;
- Example of a research project for the acceleration of an Autonomous Driving component.

Reference platform

- AVNET Ultra96v2



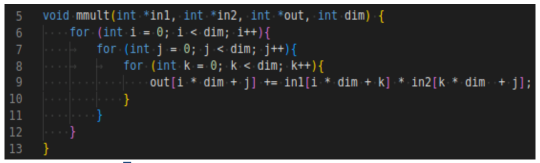
- Zynq UltraScale+
- CPU:
  - Quad-cores ARM Cortex A53;
  - Dual-cores ARM Cortex R5F;
- GPU:
  - ARM Mali 400 (only for graphics);
- 16nm FPGA.

AMD toolchain (1)

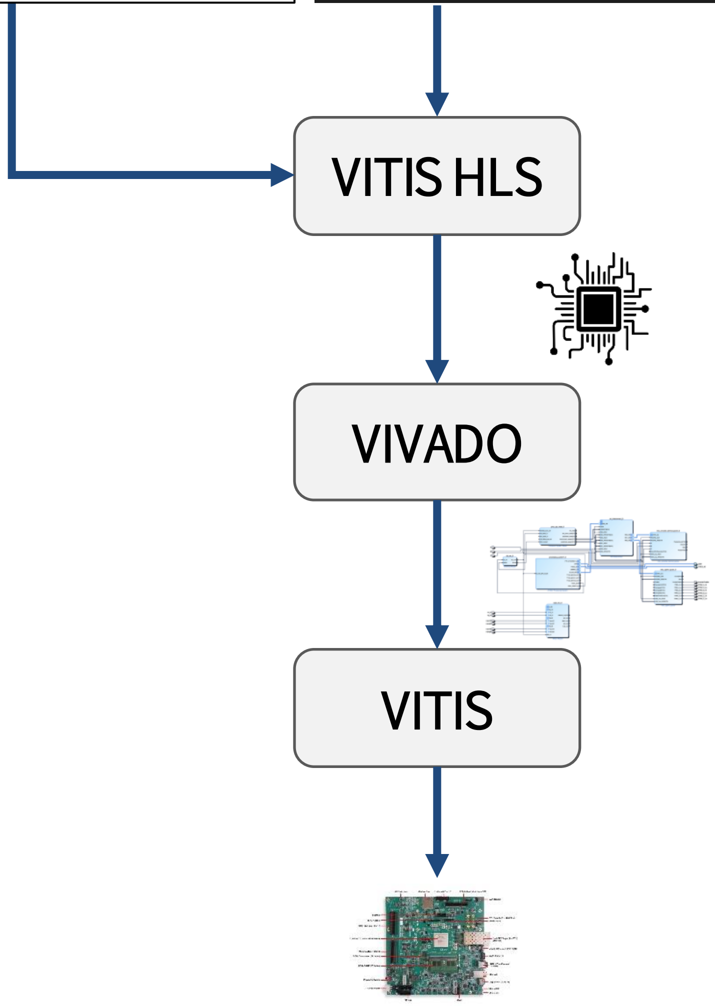
- **Vitis HLS**:
  - To convert a C/C++ based accelerator to an HDL representation (Verilog/VHDL).
- **Vivado**:
  - For FPGA hardware design and CPU hardware configurations.
- **Vitis**:
  - SDK to implement C/C++ software targeting the CPU of the embedded board.

AMD toolchain (2)

- Synthesizer instructions, in the form of **#pragma** directives



```
void mmult(int *in1, int *in2, int *out, int dim) {
    for (int i = 0; i < dim; i++){
        for (int j = 0; j < dim; j++){
            for (int k = 0; k < dim; k++){
                out[i * dim + j] += in1[i * dim + k] * in2[k * dim  + j];
            }
        }
    }
}
```

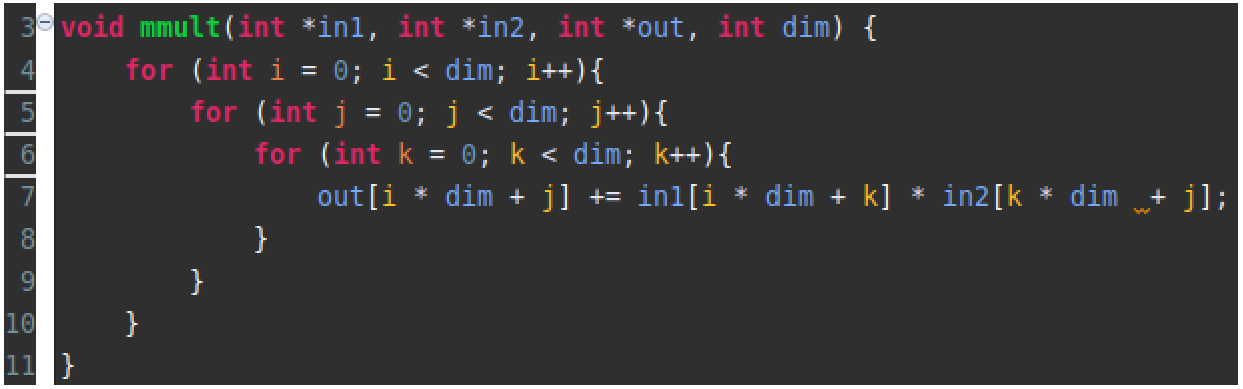


- (**HW** implementation of the matrix multiplication)
- (**implemented block design** containing the matrix multiplication engine)
- (**final software application** with the FPGA-accelerated matrix multiplication)

High-level synthesis using Vitis HLS

Acceleration of a matrix multiplication

› Starting point:



```c
void mmult(int *in1, int *in2, int *out, int dim) {
    for (int i = 0; i < dim; i++){
        for (int j = 0; j < dim; j++){
            for (int k = 0; k < dim; k++){
                out[i * dim + j] += in1[i * dim + k] * in2[k * dim + j];
            }
        }
    }
}
```

› Software implementation of the matrix multiplication, written in C language;

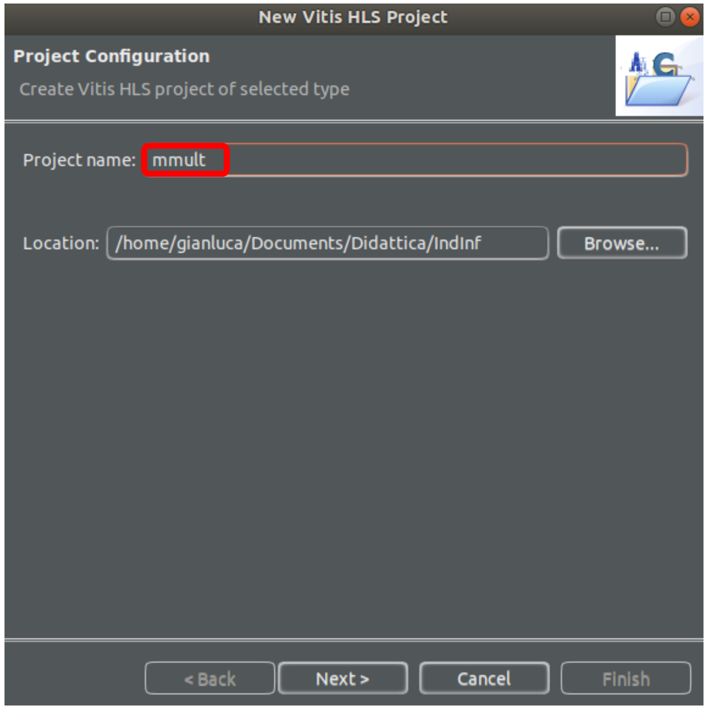
› As first instance we will try to synthesize a matrix multiplication engine using this code as is.

Vitis HLS project (1)

- Let's create a Vitis HLS project targeting the AMD ZCU102 platform;
- Open a terminal, source the environment and launch Vitis HLS:
  - $ source /tools/Xilinx/Vivado/2021.2/settings64.sh
  - $ vitis_hls
- Create a new Vitis HLS project:
  - File > New Project

Vitis HLS project (2)

- Insert project name (e.g. mmult)



New Vitis HLS Project

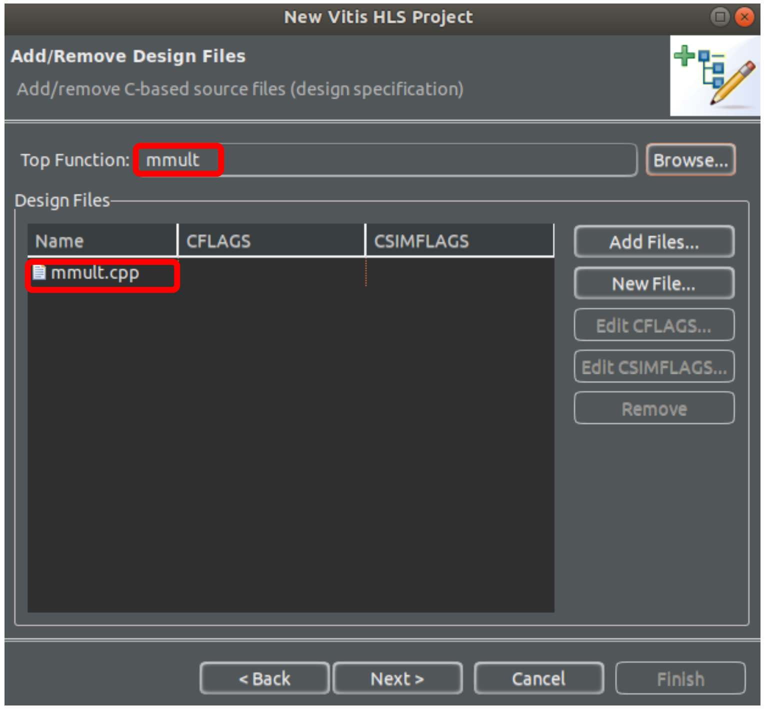
Project Configuration

Create Vitis HLS project of selected type

Project name: mmult

Location: /home/gianluca/Documents/Didattica/IndInf

- Import the .cpp code and set the **top function**.



New Vitis HLS Project

Add/Remove Design Files

Add/remove C-based source files (design specification)

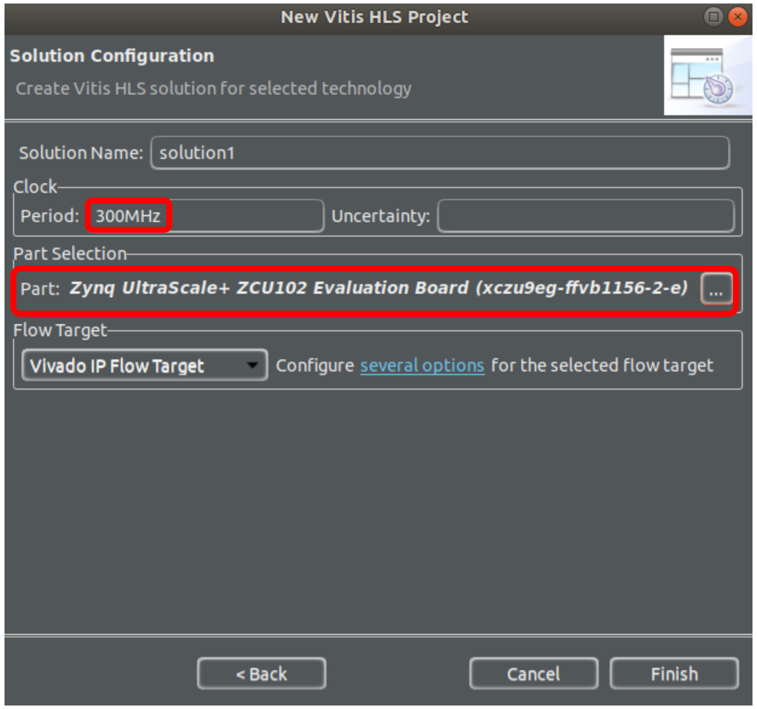
Top Function: mmult

Design Files

Name	CFLAGS	CSIMFLAGS
mmult.cpp		

- The **top function** is the main function that will be converted to a hardware module. All the nested functions will be converted as well.

Vitis HLS project (3)


New Vitis HLS Project

Solution Configuration

Create Vitis HLS solution for selected technology

Solution Name: solution1

Clock

Period: 300MHz Uncertainty:

Part Selection

Part: ***Zynq UltraScale+ ZCU102 Evaluation Board (xczu9eg-ffvb1156-2-e)***

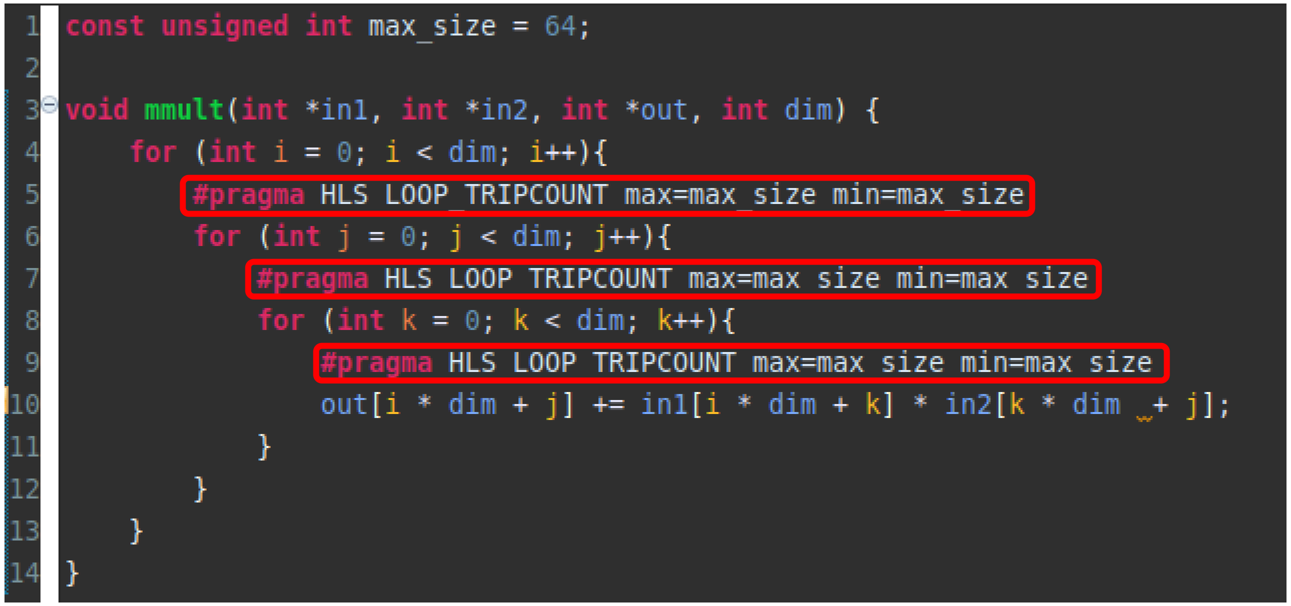
Flow Target

Vivado IP Flow Target — Configure several options for the selected flow target

- Insert the **target frequency** of the mmult engine. We will try to maximize the frequency (300MHz);
- Enter the target platform (e.g. ZCU102 Evaluation Board)

Engine profiling

› Open the matrix multiplication code and insert #pragma directives for profiling.

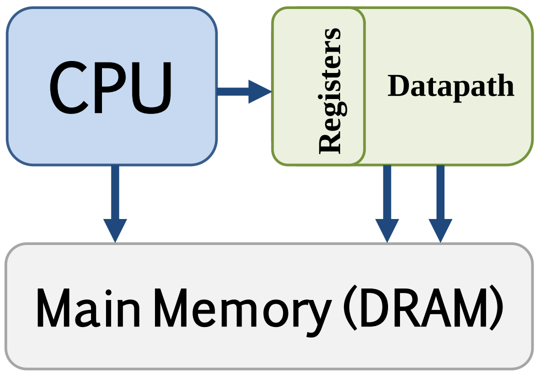


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {
    for (int i = 0; i < dim; i++){
        #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
        for (int j = 0; j < dim; j++){
            #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
            for (int k = 0; k < dim; k++){
                #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
                out[i * dim + j] += in1[i * dim + k] * in2[k * dim + j];
            }
        }
    }
}
```

› **LOOP_TRIPCOUNT** allows to specify a minimum and maximum number of iterations, when the iterations are not determined at design time.

Interfaces (1)

- Some design requirements:
  - The matrix multiplication engine should be configured using the CPU and it should be able to read and write the main memory. As in the figure:



- HW interfaces, typically based on the **AMBA AXI** protocol (Advanced Microcontroller Bus Architecture Advanced EXtensible Interface)

- AXI Interfaces can be specified directly in the C/C++ code as #pragma directives.

Interfaces (2)

› Three different versions of the AMBA AXI protocol:

– **AXI4**: Master/Slave bidirectional link that allows memory mapped accesses. The AXI4 is based on two different phases. The first phase (address request) is the negotiation of the read/write address, while the second one is the effective data transfer. This protocol is characterized by high performance, since it allows to perform up to 256 BUS transfers per address request (burst mode).

– **AXI4-Lite**: simpler version of the previous protocol. The working principle is the same apart that it does not allow to use burst transfers. AXI4-Lite requires an address request for each BUS transfer.

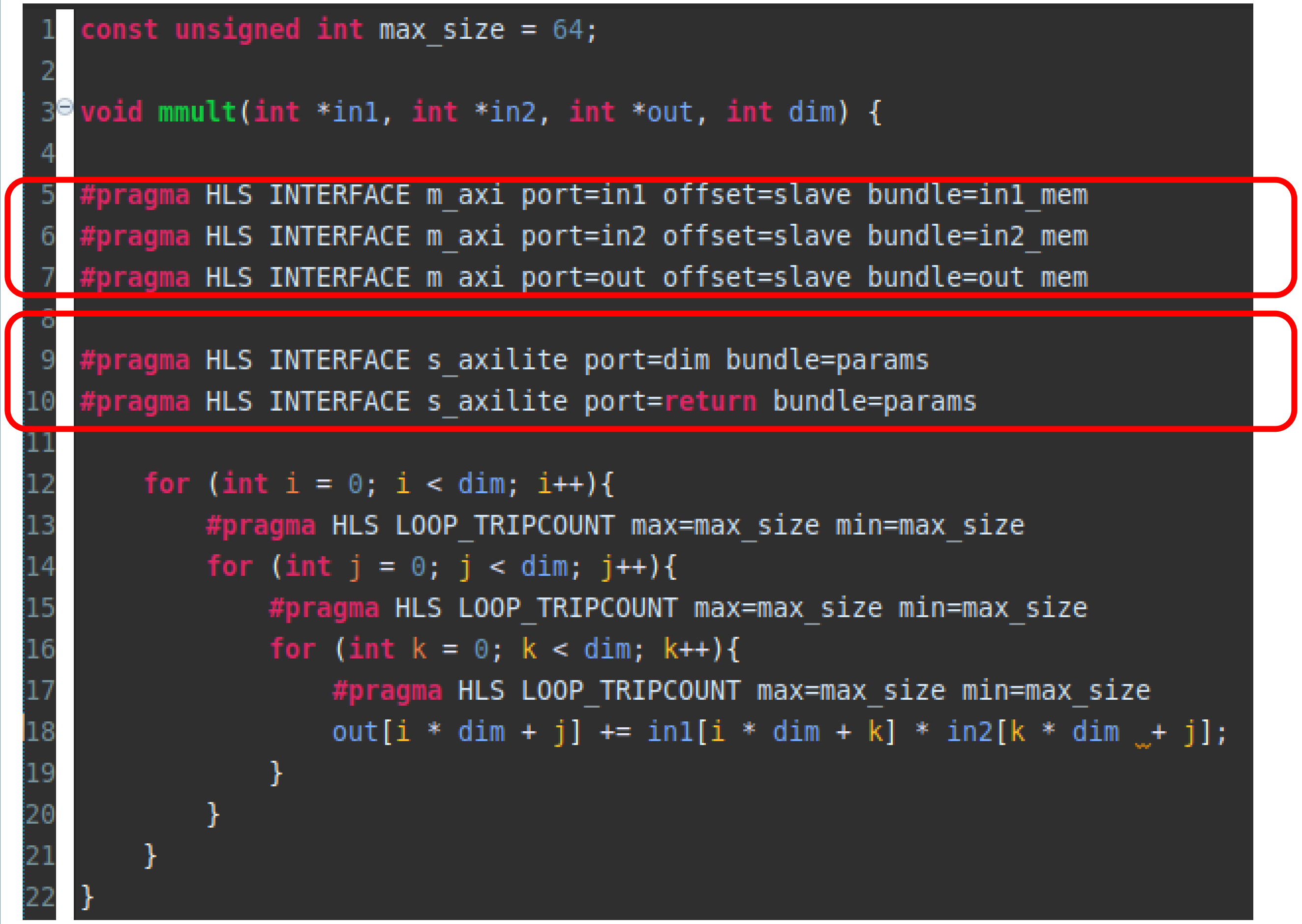
– **AXI4-Stream**: unidirectional point-to-point link without the address request phase. It is characterized by the data transfer phase only. Since it is a point-to-point link does not require the address request phase.

Interfaces (3)

- Typical usage of the three different versions of the AXI4:
  - **AXI4**:
    - Typically adopted to connect the FPGA-based engine to the Main Memory. Since in this case high performance and memory mapped accesses are required.
  - **AXI4-Lite**:
    - Since it is simpler than the full protocol, it is typically used to read/write the registers of an FPGA-based engine.
  - **AXI4-Stream**:
    - To interconnect two different FPGA-based engines.
- We will use AXI4 and AXI4-Lite on our matrix multiplication engine, to access the Main Memory and the engine's registers respectively.

Interfaces (4)

- **AXI4** is used to connect in1, in2 and out ports to the Main Memory.

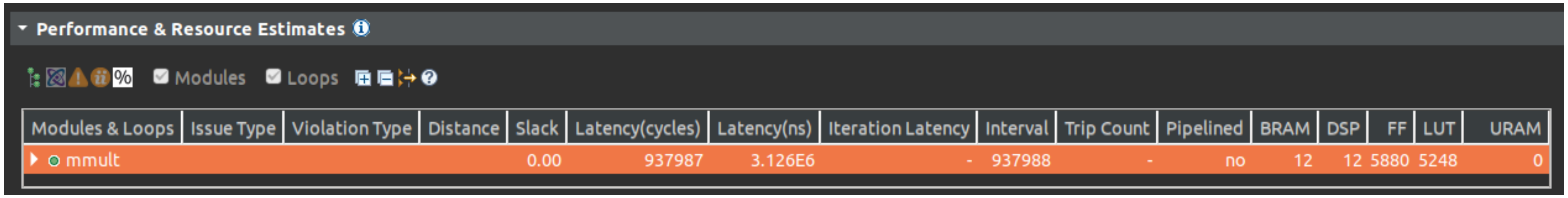


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

#pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
#pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
#pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

#pragma HLS INTERFACE s_axilite port=dim bundle=params
#pragma HLS INTERFACE s_axilite port=return bundle=params

    for (int i = 0; i < dim; i++){
        #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
        for (int j = 0; j < dim; j++){
            #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
            for (int k = 0; k < dim; k++){
                #pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size
                out[i * dim + j] += in1[i * dim + k] * in2[k * dim + j];
            }
        }
    }
}
```

- **AXI4-Lite** interfaces to start the module and read the completion bit;
- The matrix dimension is written to registers using **AXI4-Lite** as well.

Synthesis

- Run the synthesis and see the generated report:



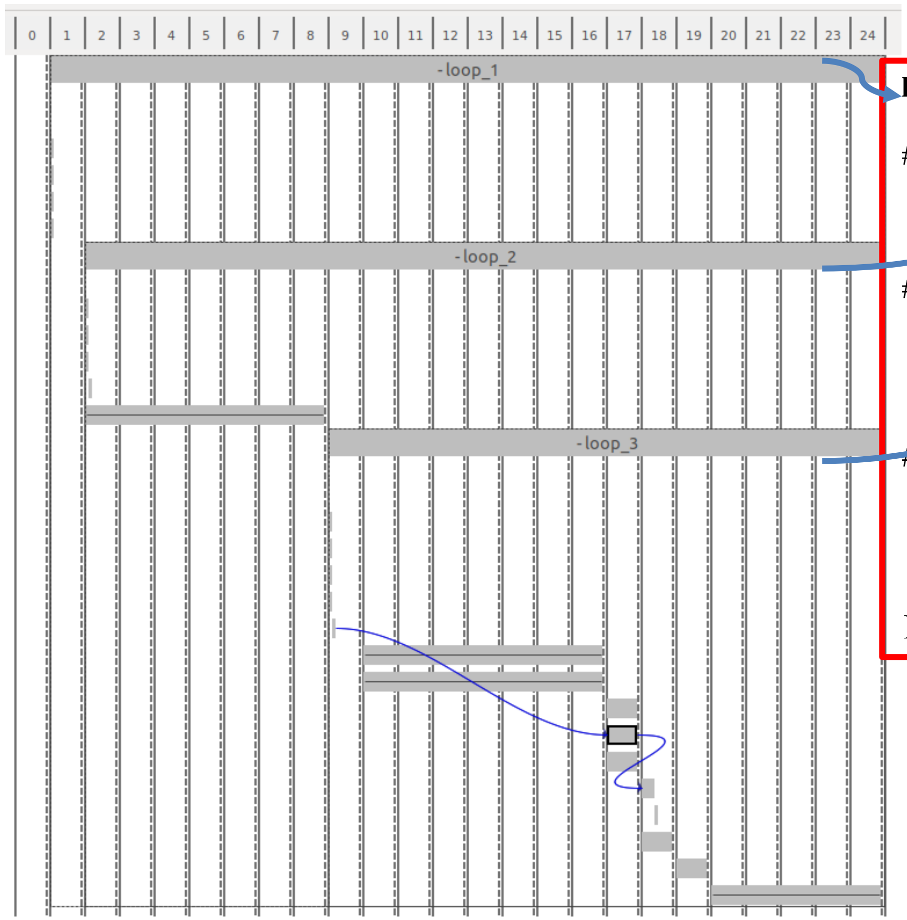
Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
mmult				0.00	937987	3.126E6	-	937988	-	no	12	12	5880	5248	0

- The report shows an estimation of the performance that can be achieved by the engine (in clock cycles and nanoseconds);
- A rough estimation of the programmable logic usage is also reported.

Schedule viewer (1)

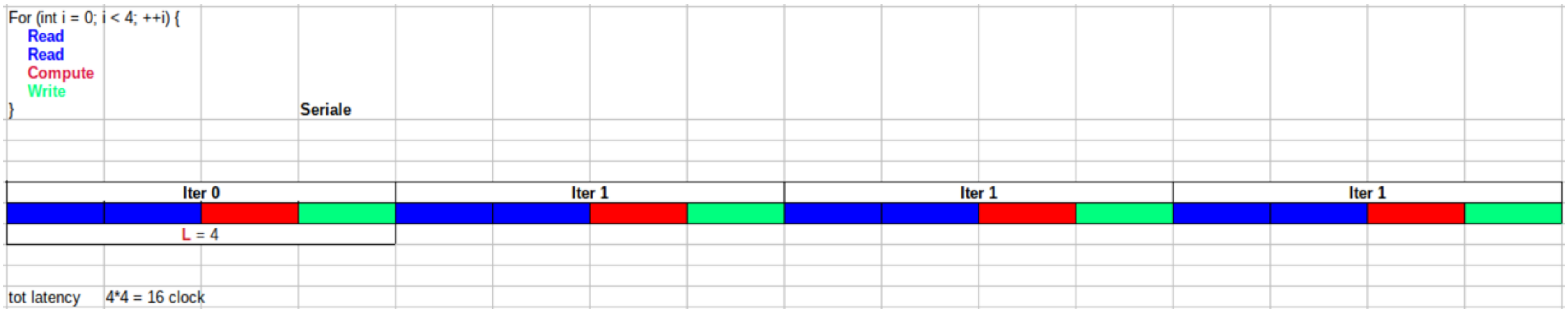


```
loop_1: for (int i = 0; i < dim;  i++){

#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

            loop_2: for (int j = 0; j < dim;  j++){

#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

                    loop_3: for (int k = 0; k < dim;  k++){

                //TODO: insert pipeline directive
#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

                out[i * dim + j] += in1[i * dim + k] * in2[k * dim  + j];
            }
        }
}
```

HLS Optimizations

Loop Pipelining (1)

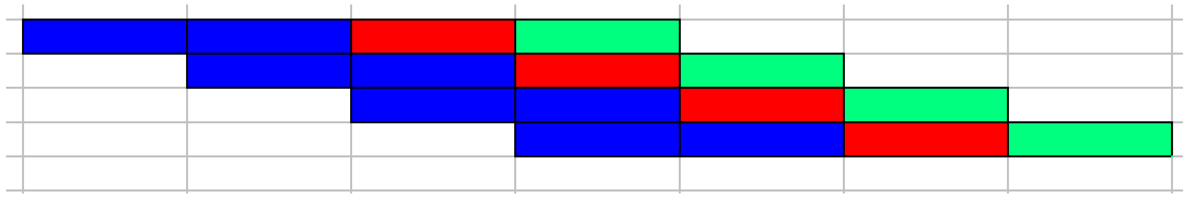
- Let's introduce the first optimization, which involves creating a pipeline on the iterations of a loop;
- Suppose to have the following pseudo-code:



- The latency of the for loop is the number of clock cycles for each iteration, multiplied by the number of iterations;
- The serial execution of an algorithm, running on low-frequency clock hardware (100MHz) and without the use of cache memory, results in poor performance.

Loop Pipelining (2)

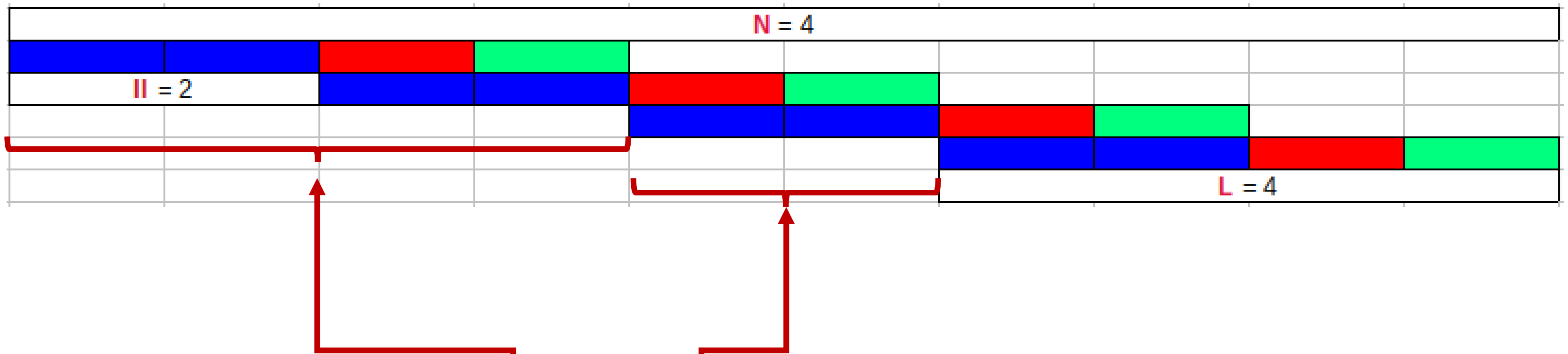
- Pipelining reduces the waiting time for an iteration of a loop to begin its execution.
- The effect is to increase the throughput of the FPGA module.

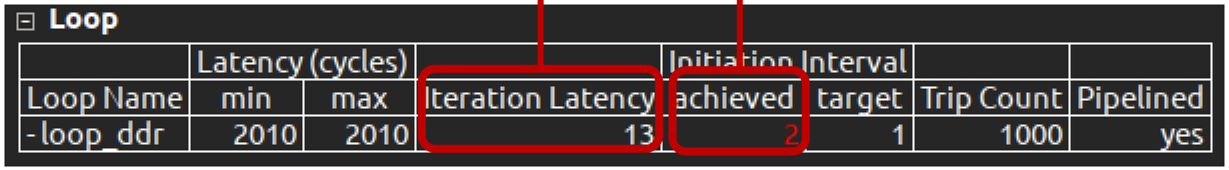


- Using HLS we can use the following #pragma directive:

```
loop: for(int i = 0; i < size; i++) {
          #pragma HLS PIPELINE
          ...
}
```

Loop Pipelining (3)

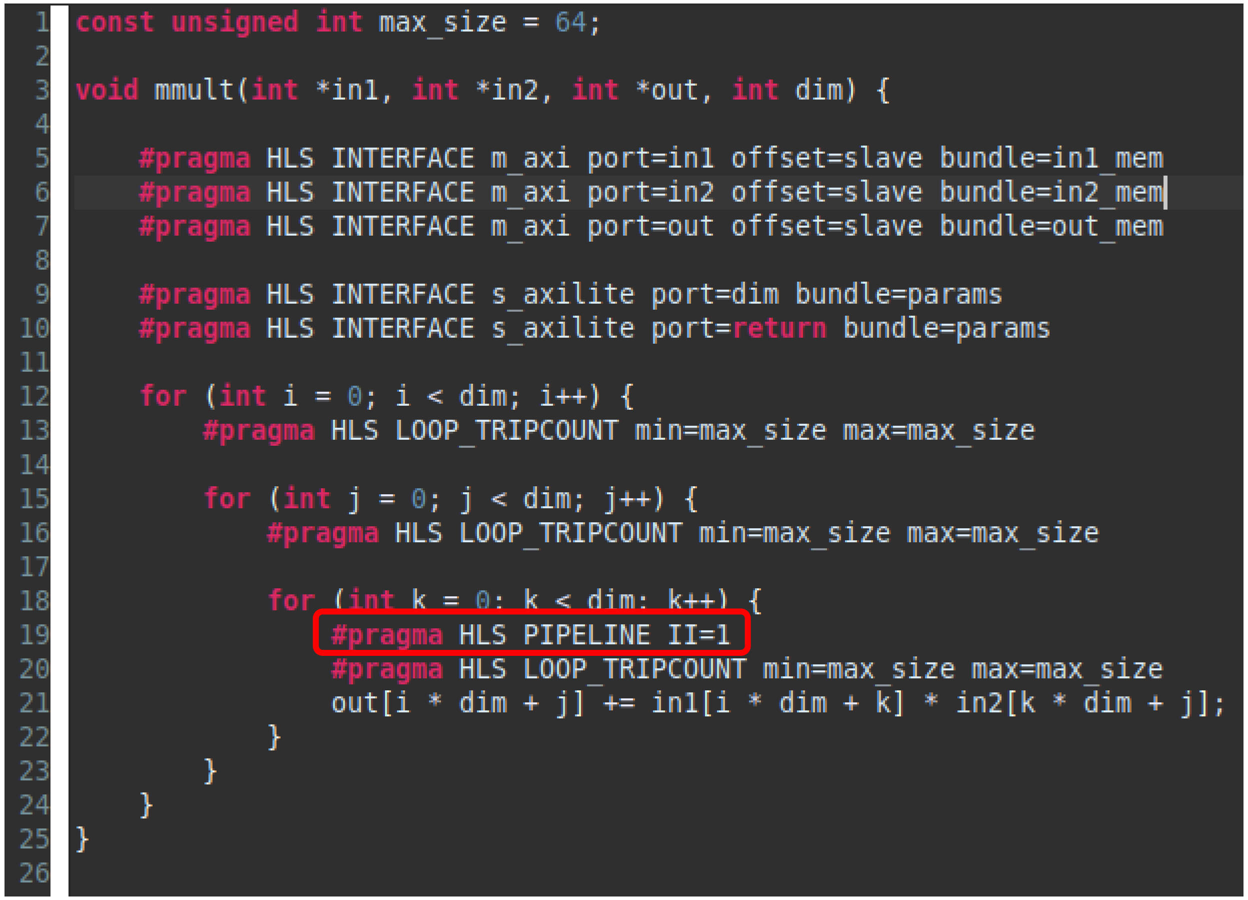




Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop_ddr	2010	2010	13	2	1	1000	yes

MMULT with Pipelining

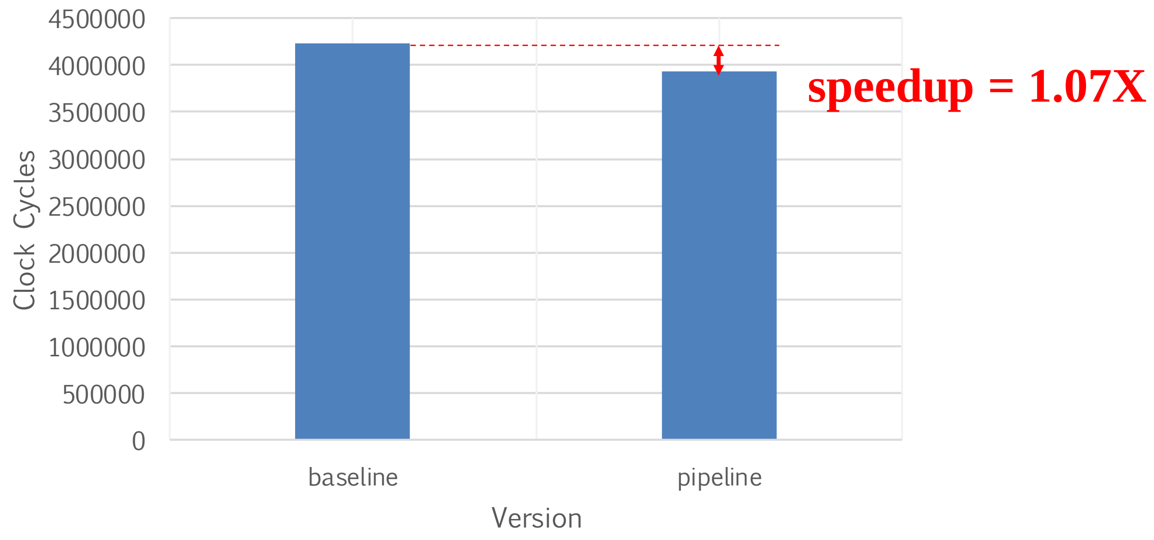


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
    #pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

    #pragma HLS INTERFACE s_axilite port=dim bundle=params
    #pragma HLS INTERFACE s_axilite port=return bundle=params

    for (int i = 0; i < dim; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

        for (int j = 0; j < dim; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

            for (int k = 0; k < dim; k++) {
                #pragma HLS PIPELINE II=1
                #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size
                out[i * dim + j] += in1[i * dim + k] * in2[k * dim + j];
            }
        }
    }
}
```

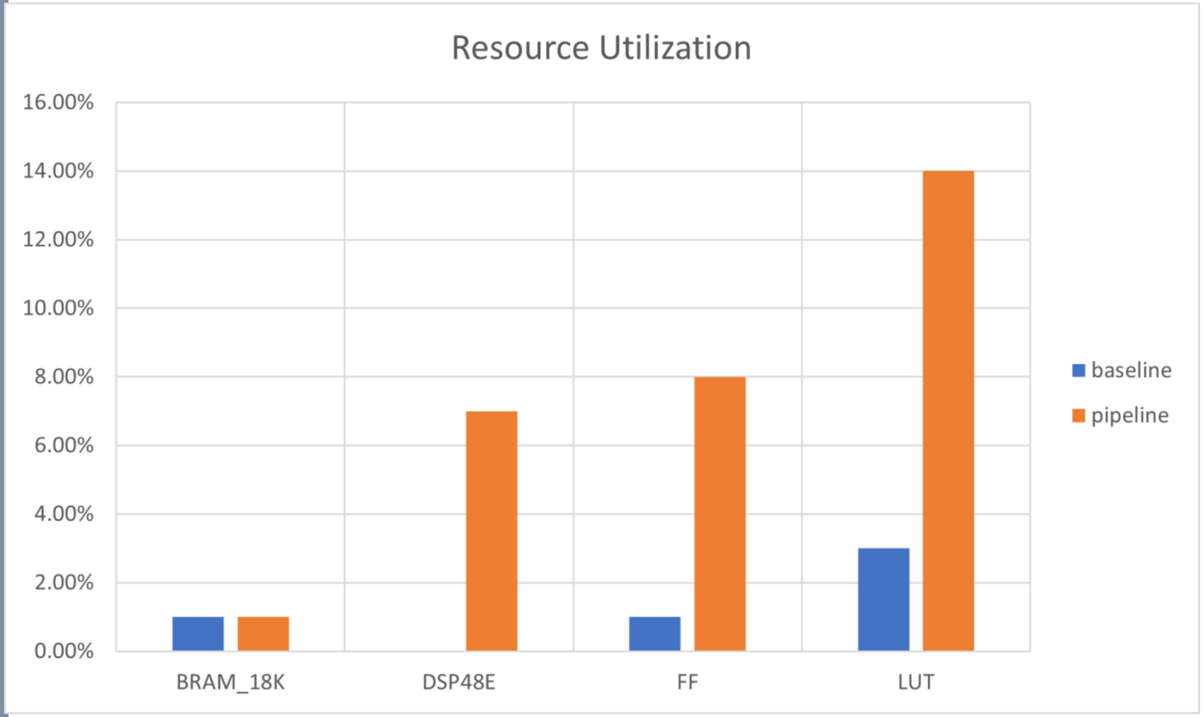
- Let's start to pipeline the inner loop;
- II = 1 tells the synthesizer to try to achieve initiation interval of 1 clock cycle.

MMULT with Pipelining -Latency

- Int32 dataype, 64x64 elements.



MMULT with Pipelining - Resources



- Greater area overhead compared to the baseline case.

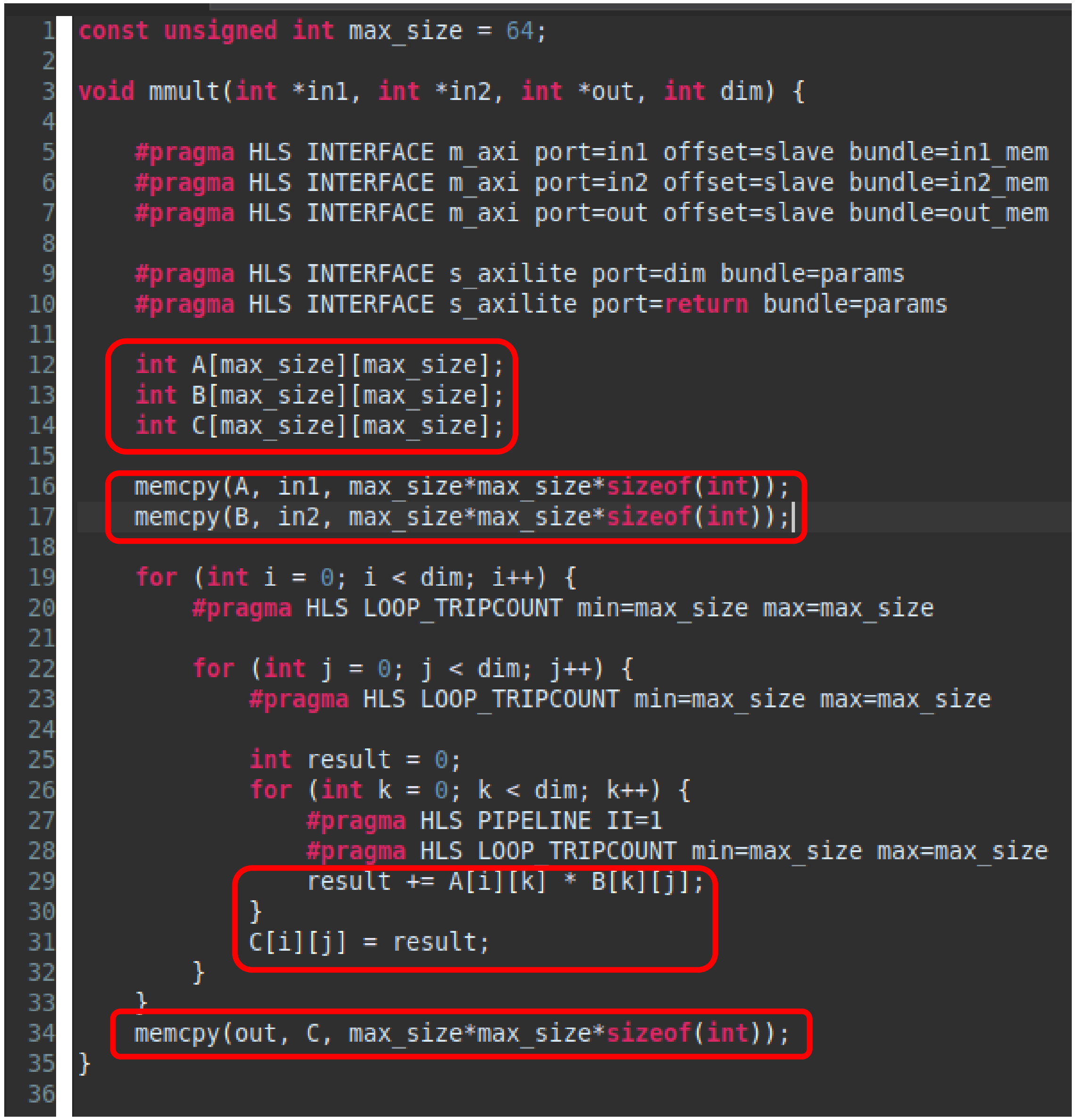
Block RAMs

- Block RAMs (BRAMs) are small memories scattered within the FPGA; they are very fast but represent a limited resource;
- BRAMs can be used to implement small scratchpads accessible by FPGA kernels, optimizing memory accesses.

- In HLS, you can use them by instantiating internal arrays within modules, for example:

```
void fun(int * in1) {

   #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem

   int v1_buffer[MAX_SIZE_LOC];                              // Block RAM
   memcpy(v1_buffer, in1, MAX_SIZE_LOC*sizeof(int));

   ...
   ...
}
```

MMULT using BRAMs

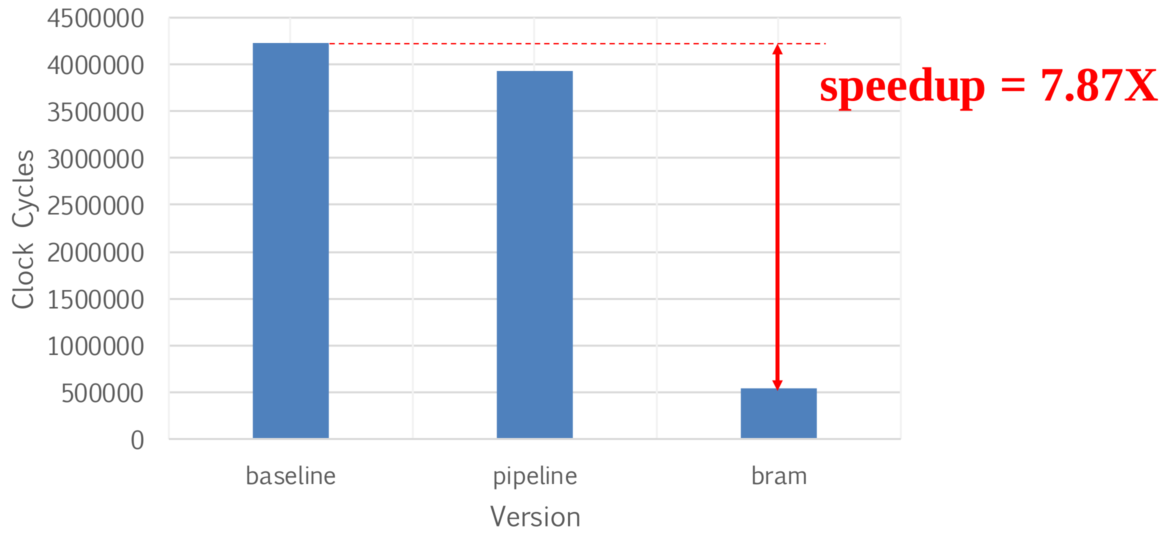


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
    #pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

    #pragma HLS INTERFACE s_axilite port=dim bundle=params
    #pragma HLS INTERFACE s_axilite port=return bundle=params

    int A[max_size][max_size];
    int B[max_size][max_size];
    int C[max_size][max_size];

    memcpy(A, in1, max_size*max_size*sizeof(int));
    memcpy(B, in2, max_size*max_size*sizeof(int));

    for (int i = 0; i < dim; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

        for (int j = 0; j < dim; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

            int result = 0;
            for (int k = 0; k < dim; k++) {
                #pragma HLS PIPELINE II=1
                #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size
                result += A[i][k] * B[k][j];
            }
            C[i][j] = result;
        }
    }
    memcpy(out, C, max_size*max_size*sizeof(int));
}

```

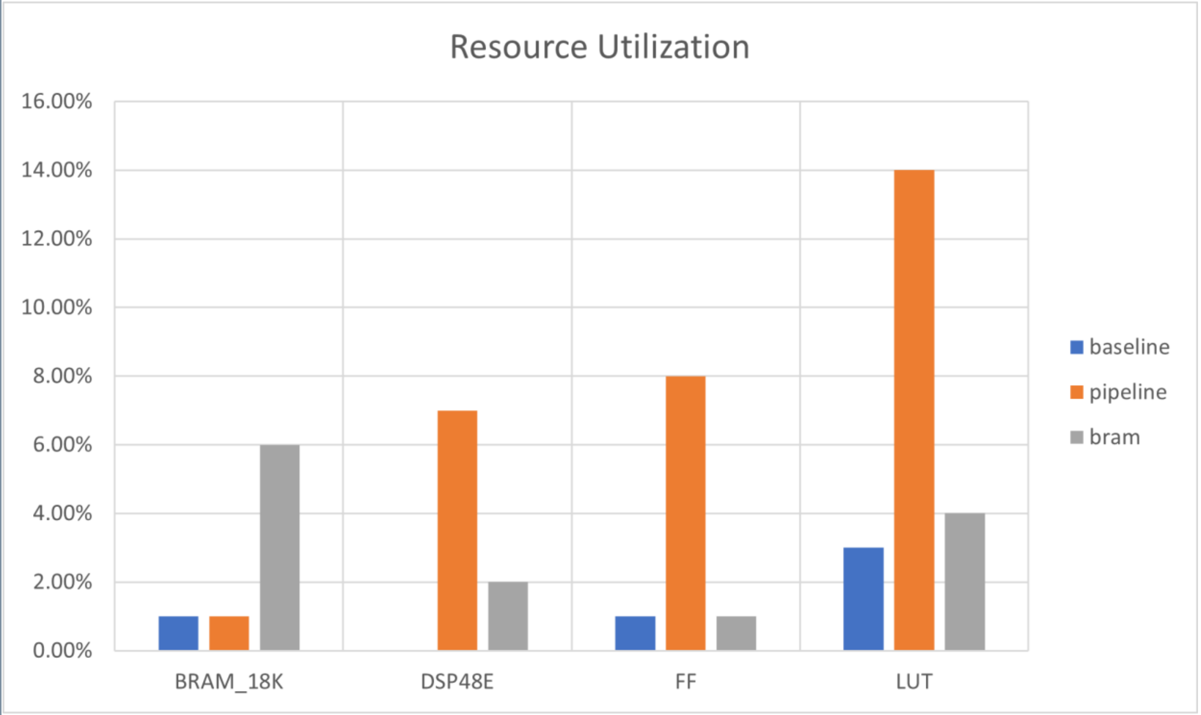
- We can declare arrays that are synthesized into BRAMs;
- Memcpys are synthesized as pipelined loops.

MMULT using BRAMs - Latency

- Int32 dataype, 64x64 elements.



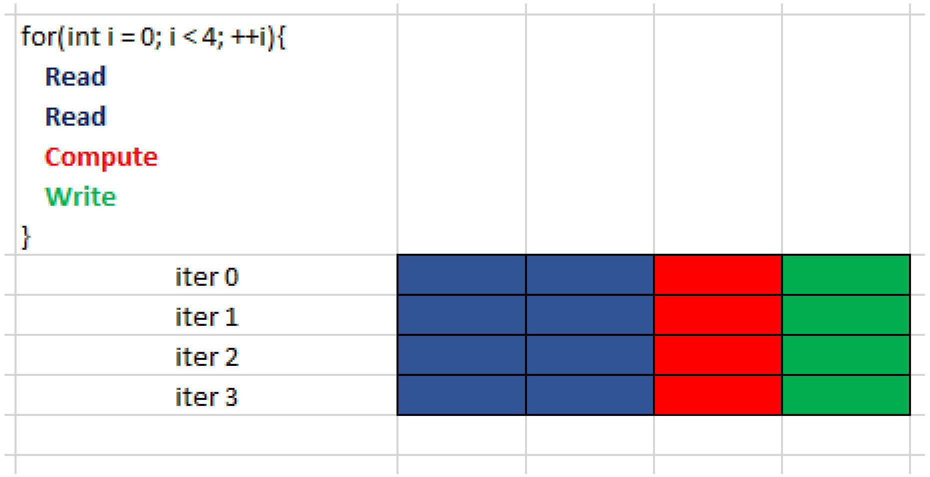
MMULT using BRAMs - Resources



- Small increase in the BRAM usage;
- In this case we have a reduction in all other resources.

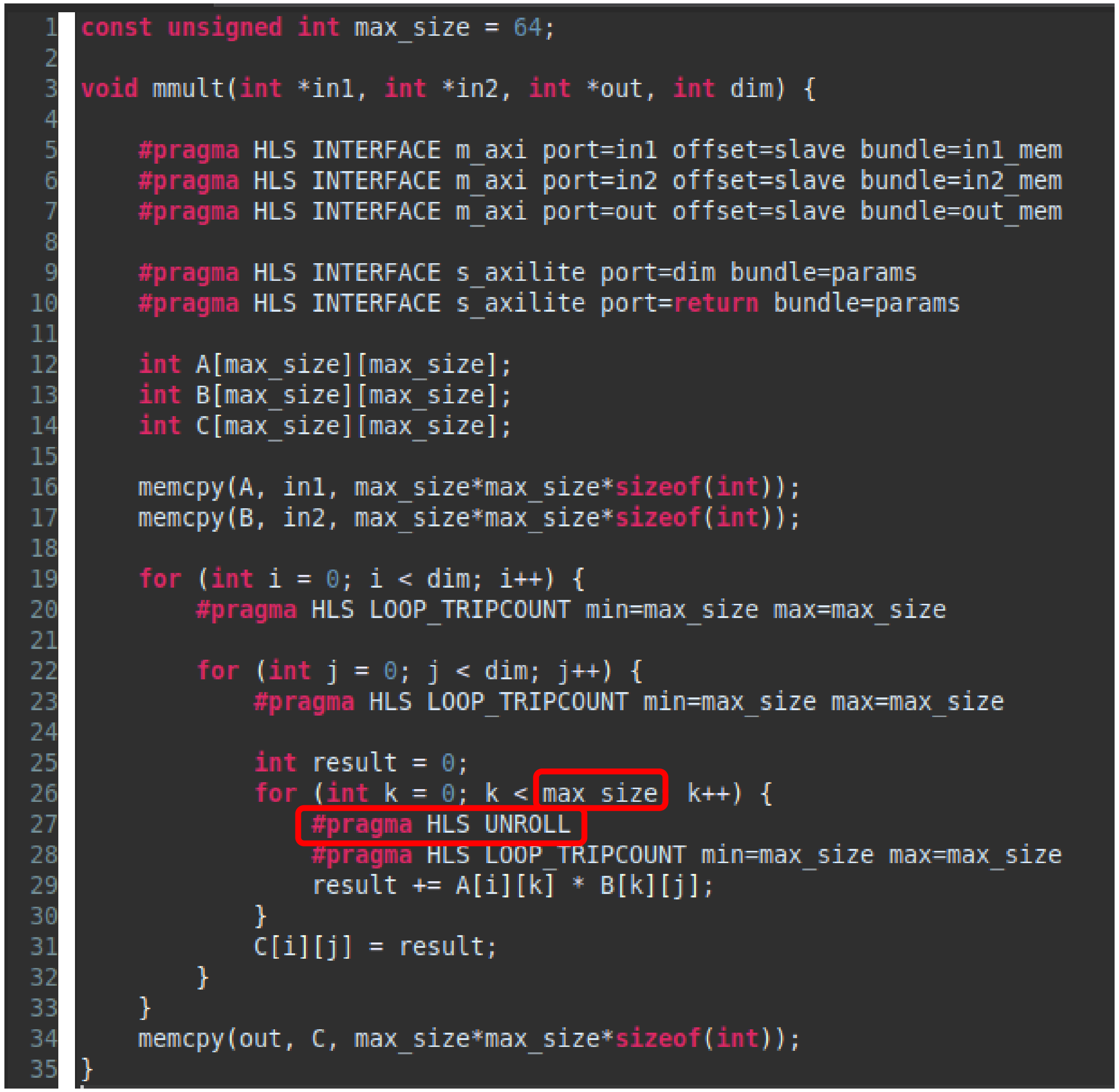
Loop Unrolling

- Through Loop Unrolling, the hardware that performs the loop iteration is replicated for each iteration.



```
loop: for(int i = 0; i < size; i++) {
            #pragma HLS UNROLL
            ...
}
```

Loop Unrolling

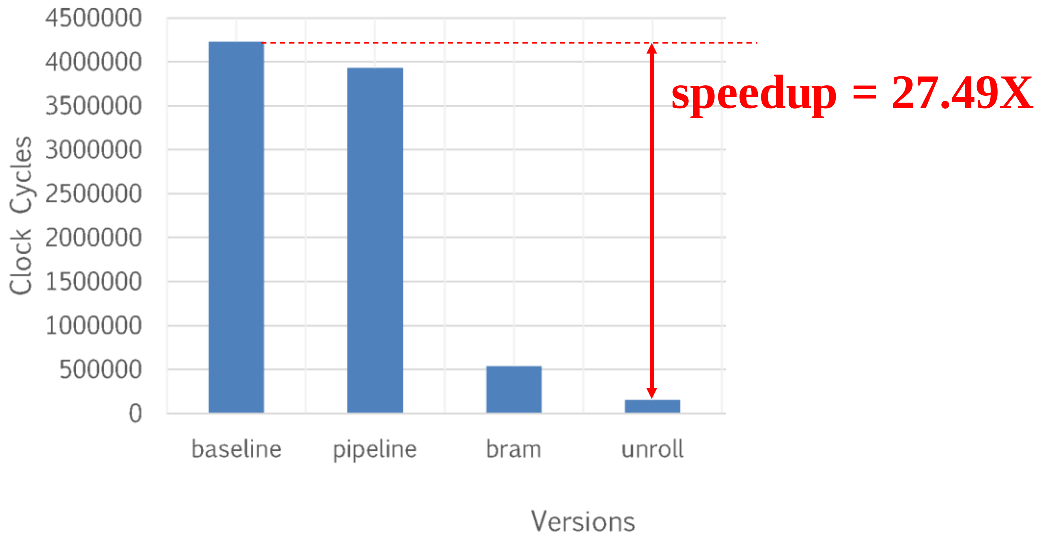


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
    #pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

    #pragma HLS INTERFACE s_axilite port=dim bundle=params
    #pragma HLS INTERFACE s_axilite port=return bundle=params

    int A[max_size][max_size];
    int B[max_size][max_size];
    int C[max_size][max_size];

    memcpy(A, in1, max_size*max_size*sizeof(int));
    memcpy(B, in2, max_size*max_size*sizeof(int));

    for (int i = 0; i < dim; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

        for (int j = 0; j < dim; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

            int result = 0;
            for (int k = 0; k < max_size; k++) {
                #pragma HLS UNROLL
                #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size
                result += A[i][k] * B[k][j];
            }
            C[i][j] = result;
        }
    }
    memcpy(out, C, max_size*max_size*sizeof(int));
}
```

- All the inner loops after a #pragma HLS PIPELINE are **automatically unrolled**;
- Unrolled loops must have **a constant** number of iterations.

MMULT Unrolled - Latency

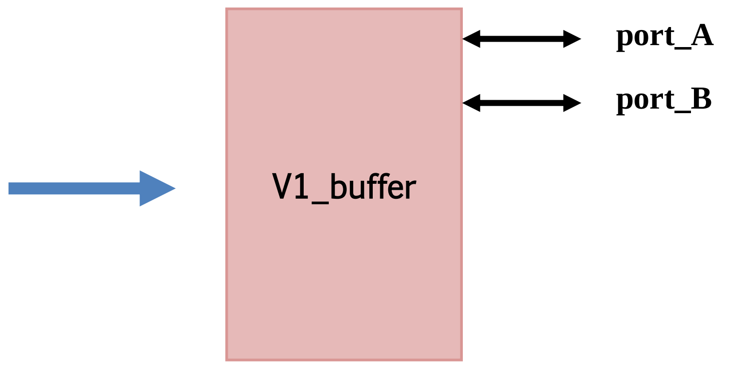
- Int32 dataype, 64x64 elements.



Array Partitioning (1)

- The use of Loop Unrolling and Block RAM is often combined. However, the **parallel access** of unrolled iterations of a loop can cause **slowdowns due to contention on the ports** of BRAM:
- Example:

```
void fun(int * in1) {

   int v1_buffer[MAX_SIZE_LOC];

   …
   …
}
```

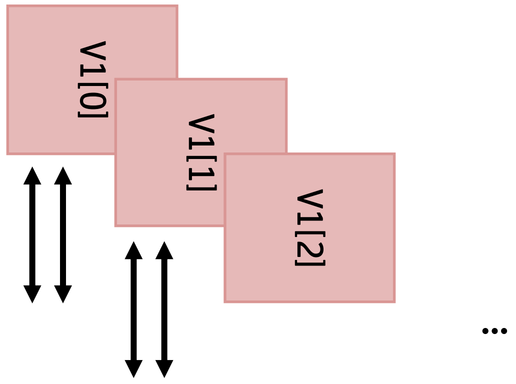


- A BRAM declared in this way has **only two ports.**

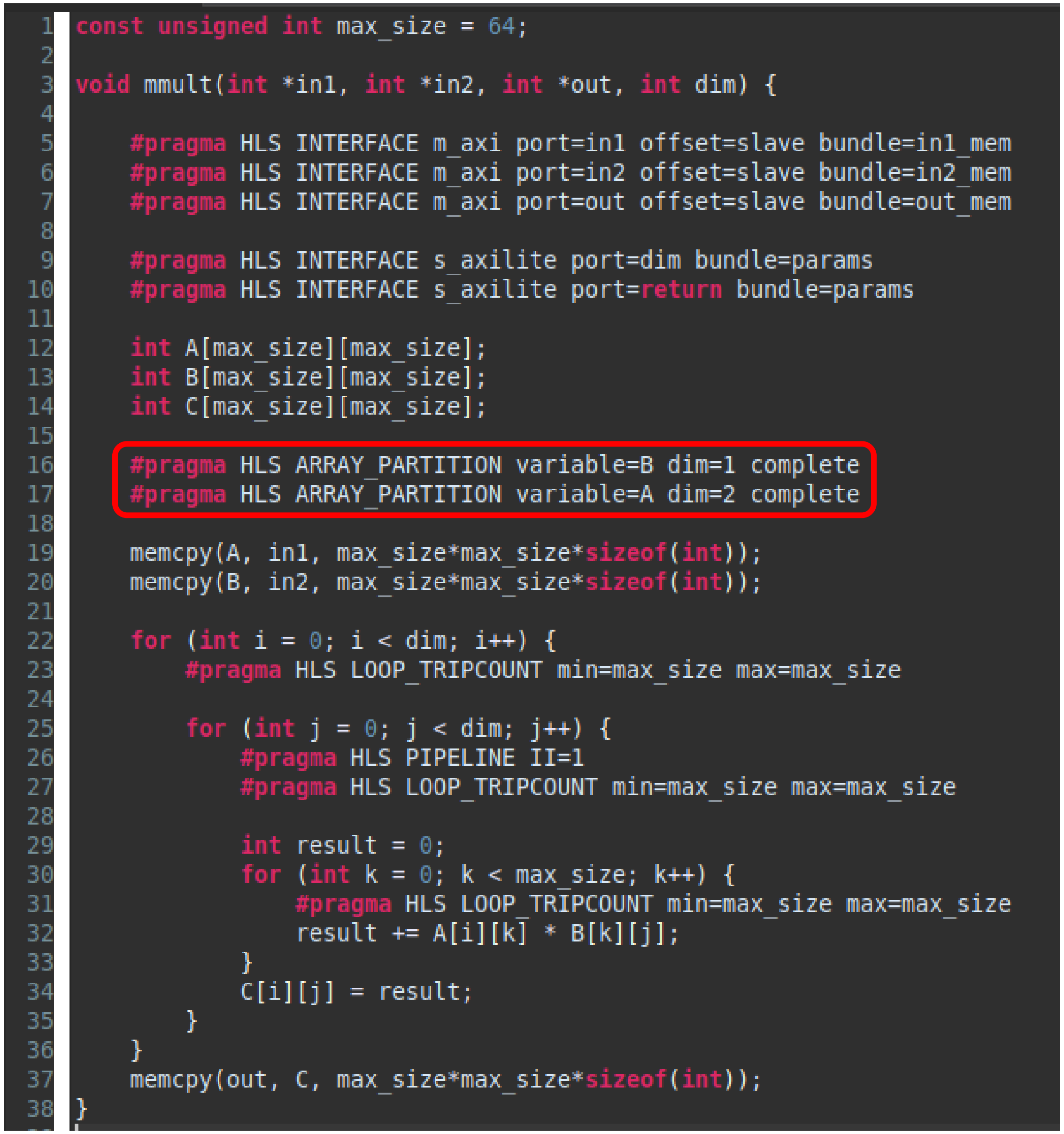
Array Partitioning (2)

- The idea of Array Partitioning is to spread the array across multiple distinct BRAMs.
  - Pros:
    - It can significantly increases the parallelism of memory access;
  - Cons:
    - The Block RAMs are not used at 100%, so the usage of BRAM increases significantly.

```
void fun(int * in1) {

   int v1_buffer[MAX_SIZE_LOC];

   #pragma HLS ARRAY_PARTITION variable=v1_buffer complete
dim=1
   …
}
```



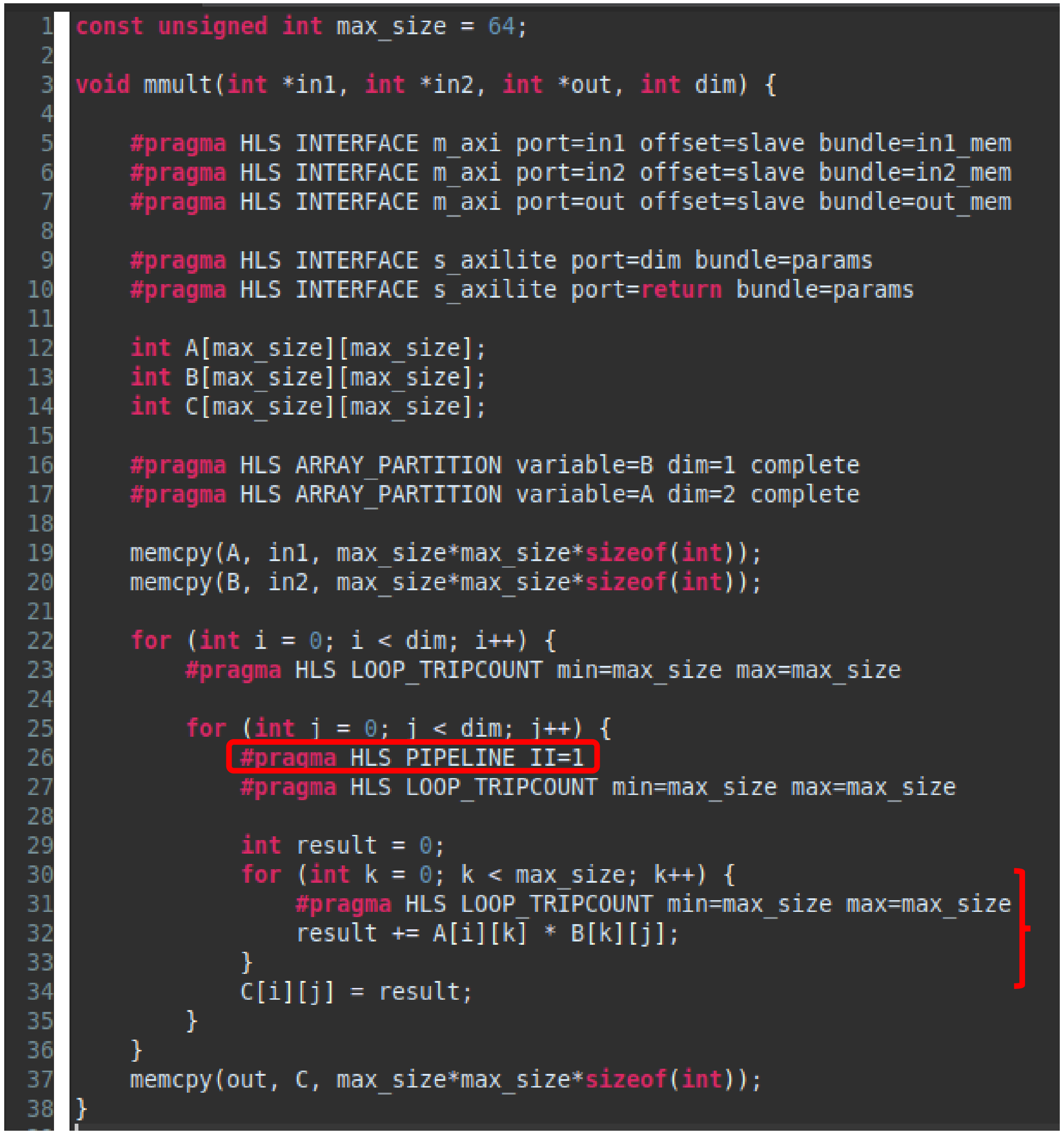
MMULT Unrolled & Partitioned



```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
    #pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

    #pragma HLS INTERFACE s_axilite port=dim bundle=params
    #pragma HLS INTERFACE s_axilite port=return bundle=params

    int A[max_size][max_size];
    int B[max_size][max_size];
    int C[max_size][max_size];

    #pragma HLS ARRAY_PARTITION variable=B dim=1 complete
    #pragma HLS ARRAY_PARTITION variable=A dim=2 complete

    memcpy(A, in1, max_size*max_size*sizeof(int));
    memcpy(B, in2, max_size*max_size*sizeof(int));

    for (int i = 0; i < dim; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

        for (int j = 0; j < dim; j++) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

            int result = 0;
            for (int k = 0; k < max_size; k++) {
                #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size
                result += A[i][k] * B[k][j];
            }
            C[i][j] = result;
        }
    }
    memcpy(out, C, max_size*max_size*sizeof(int));
}
```

- With «complete» each element on the selected dimension, are scattered on different Block RAMs.

MMULT outer loop pipelined

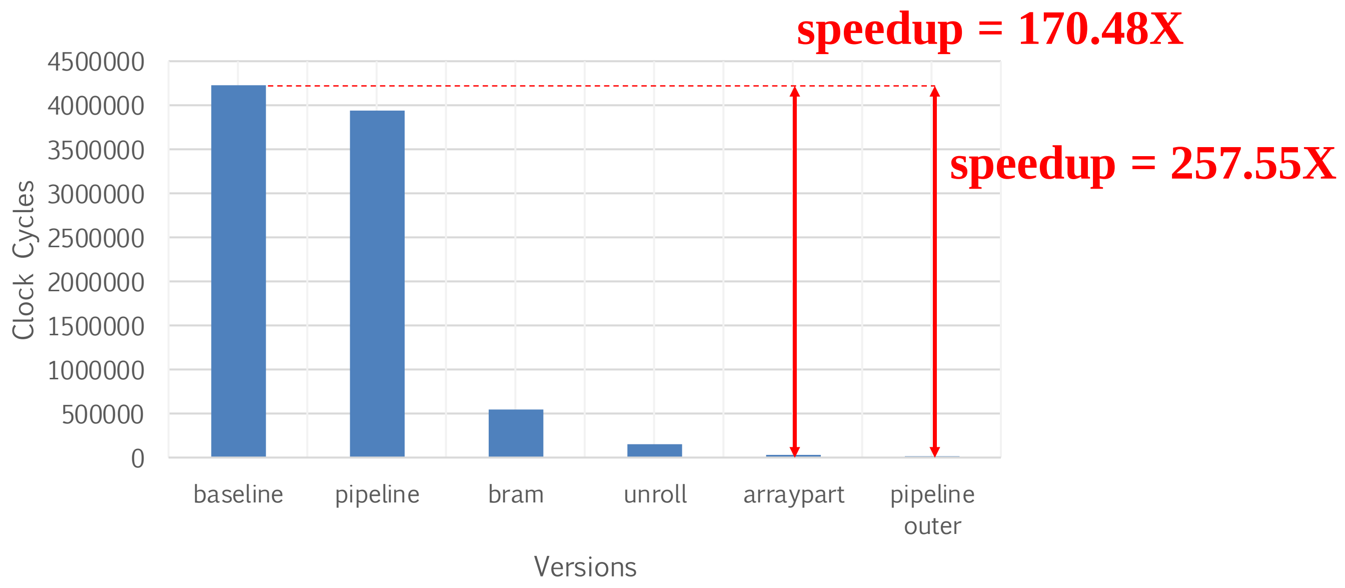


```
const unsigned int max_size = 64;

void mmult(int *in1, int *in2, int *out, int dim) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem
    #pragma HLS INTERFACE m_axi port=in2 offset=slave bundle=in2_mem
    #pragma HLS INTERFACE m_axi port=out offset=slave bundle=out_mem

    #pragma HLS INTERFACE s_axilite port=dim bundle=params
    #pragma HLS INTERFACE s_axilite port=return bundle=params

    int A[max_size][max_size];
    int B[max_size][max_size];
    int C[max_size][max_size];

    #pragma HLS ARRAY_PARTITION variable=B dim=1 complete
    #pragma HLS ARRAY_PARTITION variable=A dim=2 complete

    memcpy(A, in1, max_size*max_size*sizeof(int));
    memcpy(B, in2, max_size*max_size*sizeof(int));

    for (int i = 0; i < dim; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

        for (int j = 0; j < dim; j++) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size

            int result = 0;
            for (int k = 0; k < max_size; k++) {
                #pragma HLS LOOP_TRIPCOUNT min=max_size max=max_size
                result += A[i][k] * B[k][j];
            }
            C[i][j] = result;
        }
    }
    memcpy(out, C, max_size*max_size*sizeof(int));
}
```

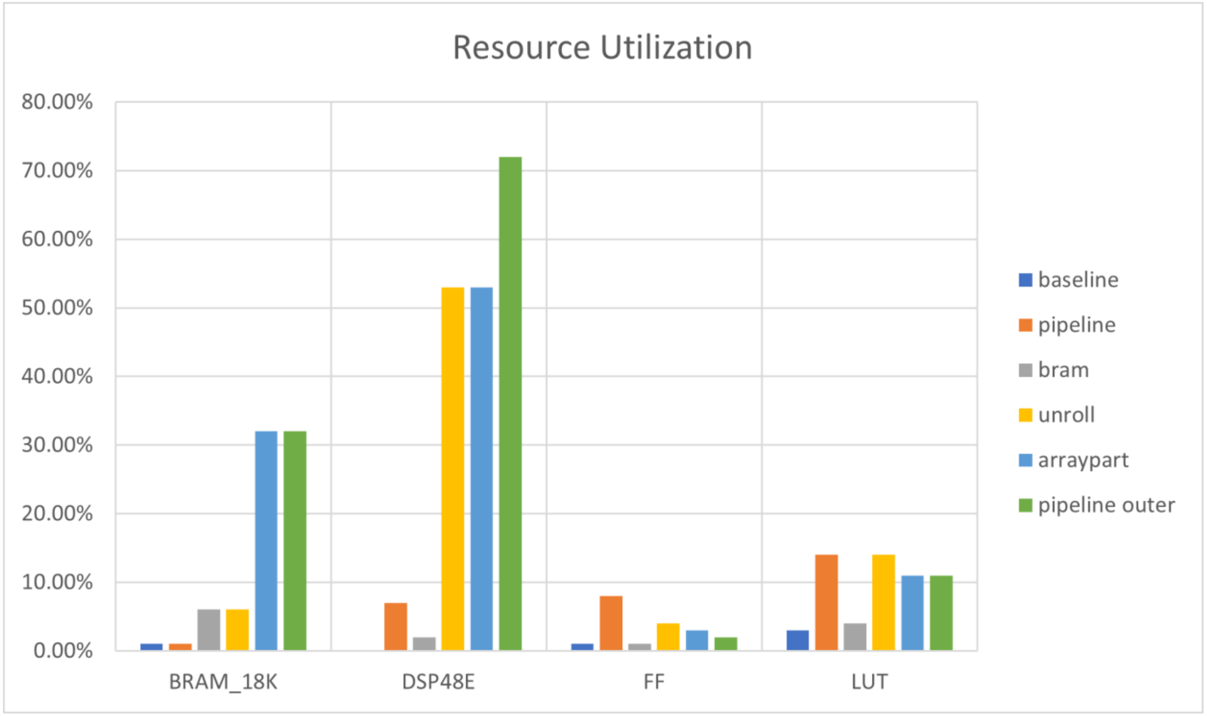
- Inserting the pipelining on the intermediate loop, the inner loop is completely unrolled.

MMULT Unrolled & Partitioned - Latency

- Int32 dataype, 64x64 elements.

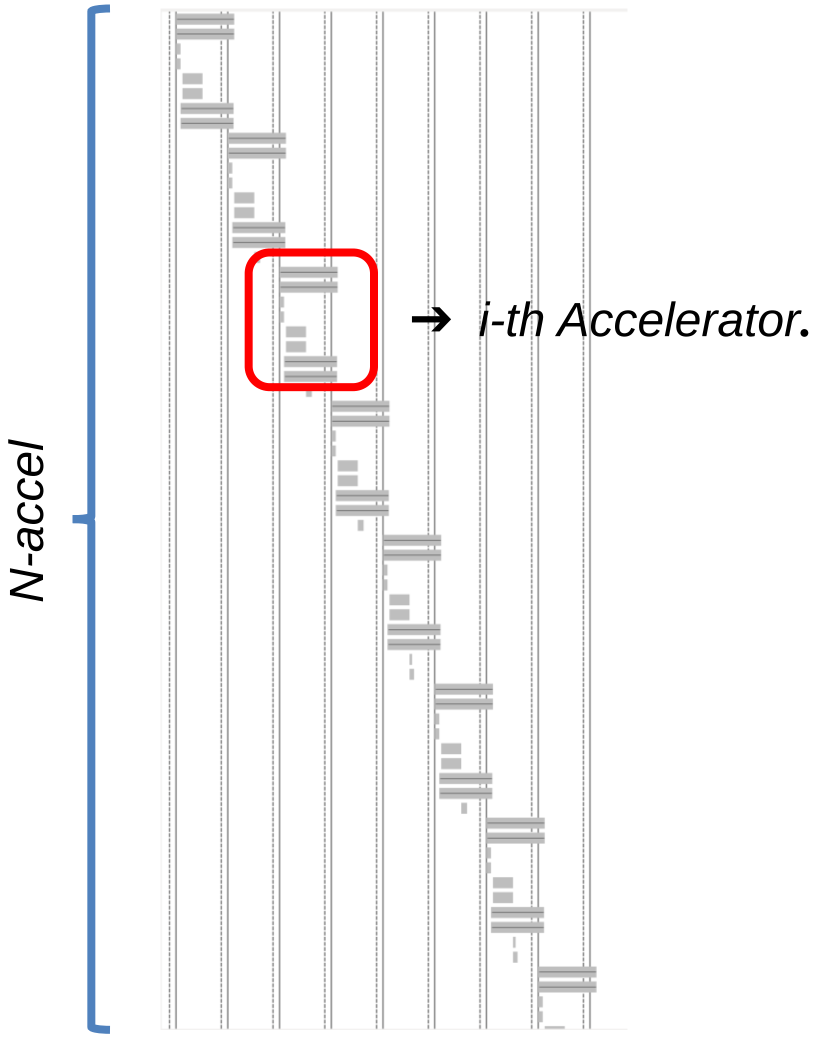


MMULT Unrolled & Partitioned - Resources



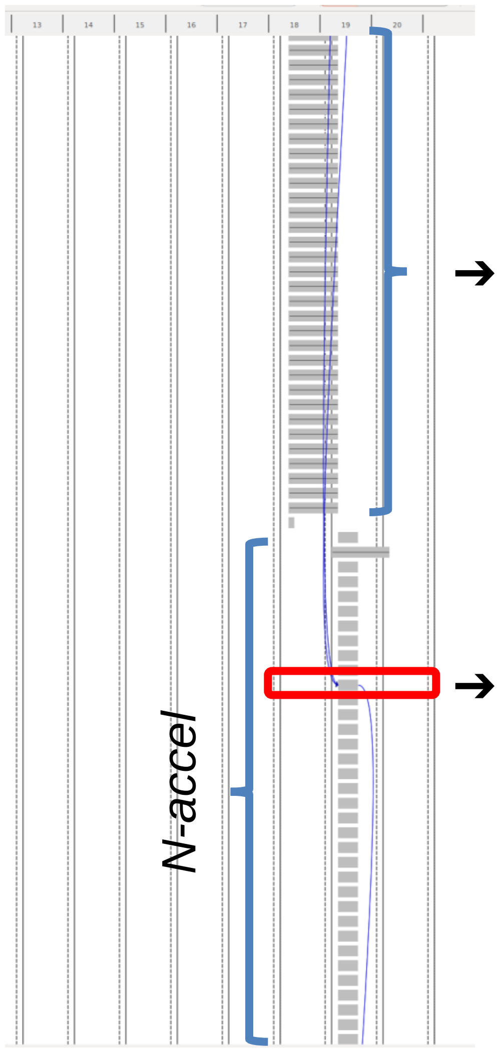
- Unrolling results in a significant increase in DSP (~50%);
- Array partitioning leads to an increment of ~25% in BRAM usage, due to underutilization;
- Pipelining results in a further increase in DSP.

MMULT Unrolled - scheduler



- **Unrolled** Version;
- **N parallel accelerators**;
- In this case, the reduced number of ports on the BRAMs causes contention on accesses;
- Despite having N accelerators, the execution does not effectively happen in parallel.

MMULT Unrolled & Partitioned- scheduler



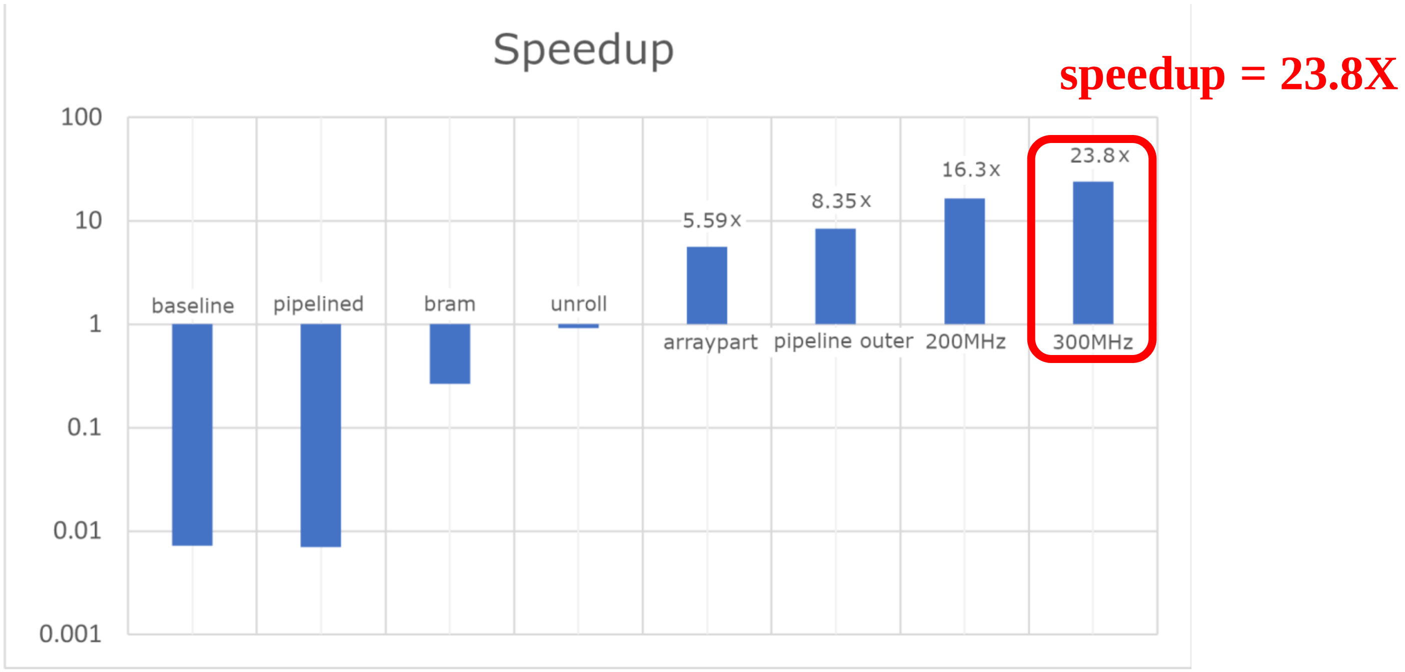
➔ ***The reads/writes effectively occur in parallel.***

➔ *i-th Accelerator.*

- **Unrolled** Version with **array partitioning**;
- **N parallel accelerators**;
- In this case, the accelerators are scheduled in parallel, utilizing all ports on the BRAMs

Speedups on real platform

Speedups on real platform



- The speedup are referred to a software implementation (baremetal) on a single-core ARM Cortex A53 @ 1.3 GHz;
- The final FPGA version is an accelerator with all the optimizations that we saw before, clocked @ 300 MHz.

How to use an HLS design on a real platform

(we'll see it in practice)

Real-Time localization using FPGA

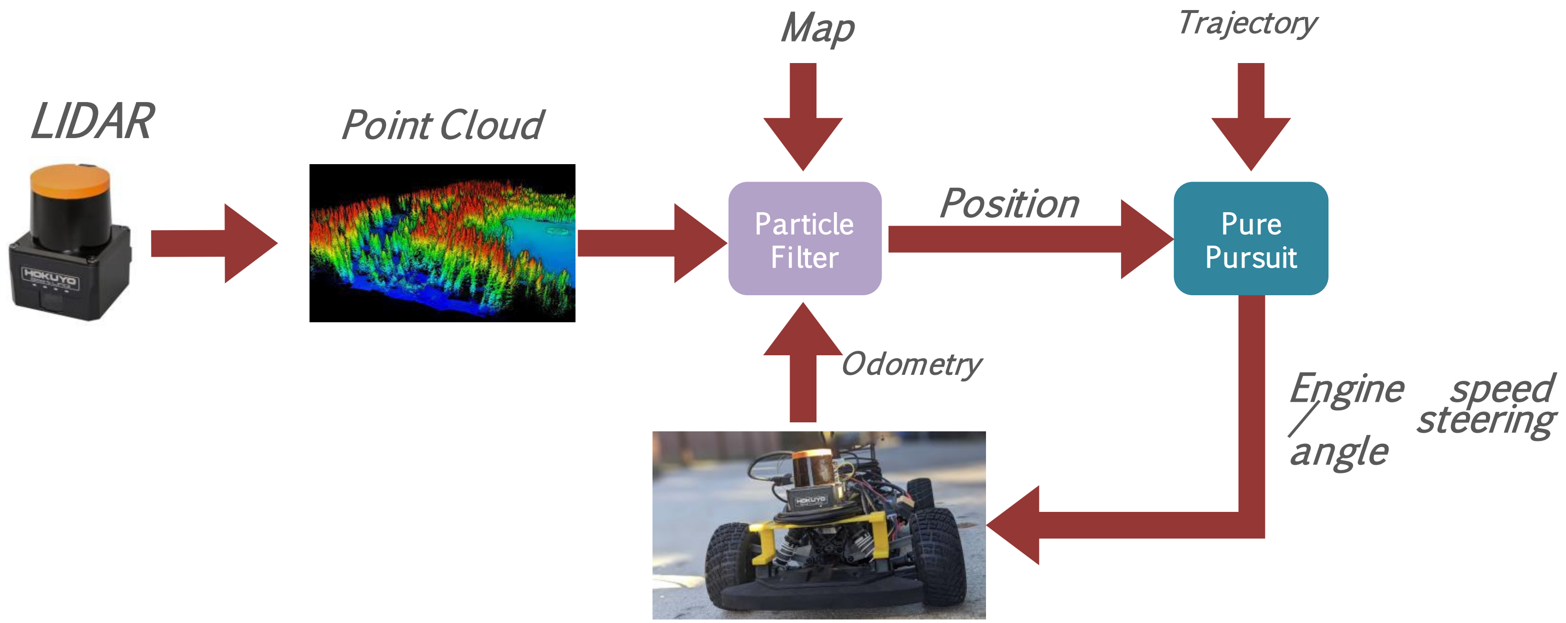
Autonomous Driving using FPGA (1)

- Autonomous Driving (AD) on FPGAs is an interesting research topic;
- 1/10 scale cars (F1/10) are a perfect environment to test and learn different AD solutions;
- The natural follow-up to F1/10 is the transition to full-size vehicles.

Andrea Bernardi, Gianluca Brilli, Alessandro Capotondi, Andrea Marongiu, Paolo Burgio, An FPGA Overlay for Efficient Real-Time Localization in 1/10th Scale Autonomous Vehicles, ***2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*****.**

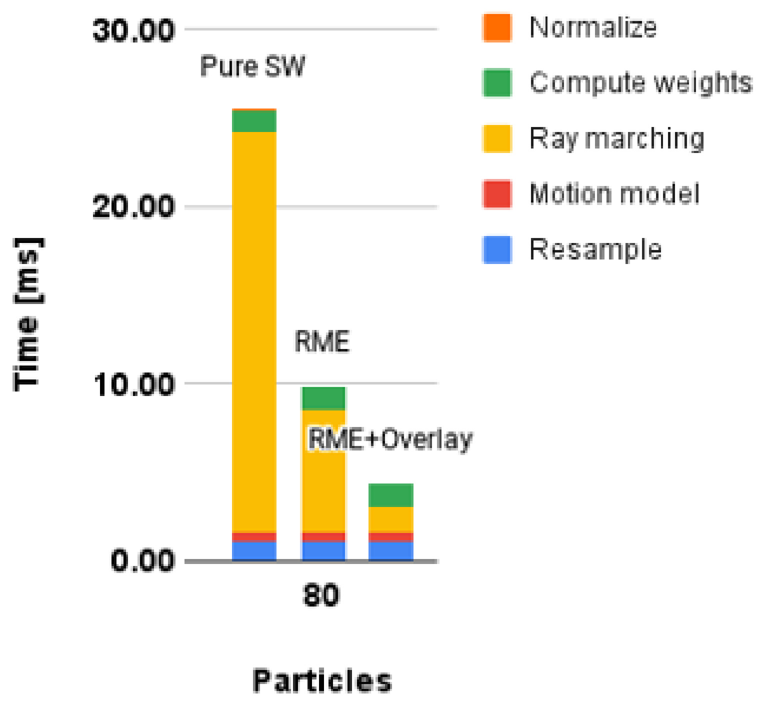
Autonomous Driving using FPGA (2)

- Our F1/10th AD stack:



- Our focus is to implement the Particle Filter algorithm using our FPGA-based template.

Autonomous Driving using FPGA (3)



› 4.6x improvement using our template (HW accel / proxy core).

References

Course website

› http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

› paolo.burgio@unimore.it

› http://hipert.mat.unimore.it/people/paolob/

Resources

› Xilinx Zynq-7000 All Programmable SoC => https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

› Pynq => http://www.pynq.io/

› Xilinx Ultrascale => https://www.xilinx.com/products/technology/ultrascale.html

› A "small blog« => http://www.google.com