Rapporti Tecnici INAF
INAF Technical Reports

Number	96
Publication Year	2021
Acceptance in OA@INAF	2021-09-27T08:36:37Z
Title	Software acceleration on Xilinx FPGAs using OmpSs@FPGA ecosystem
Authors	Goz, David; BERTOCCO, SARA; TAFFONI, Giuliano; CORETTI, Igor
Affiliation of first author	O.A. Trieste
Handle	http://hdl.handle.net/20.500.12386/31054; http://dx.doi.org/10.20371/INAF/TechRep/96

INAF Technical Report: Software acceleration on Xilinx FPGAs using OmpSs@FPGA ecosystem

D. Goz[a], S. Bertocco[a], G. Taffoni[a], I. Coretti[a]

[a]*INAF-Osservatorio Astronomico di Trieste, Via G. Tiepolo 11, 34131 Trieste - Italy*

Abstract

The `OmpSs@FPGA` programming model allows offloading application functionality to Xilinx Field Programmable Gate Arrays (FPGAs). The `OmpSs` compiler splits the code (written in C/C++ high level language) in two parts, targeting the host and the fpga. The first is usually compiled by the `GNU Compiler Collection` (`GCC`), while the latter is given to the `Xilinx Vivado HLS` tool (hereafter `HLS`) for high level synthesis to VHDL and bitstream used to program the FPGA. `OmpSs@FPGA` is based on compiler directives, which allow the programmer to annotate the part of the code to automatically exploit all Symmetric MultiProcessor system (`SMP`) and FPGA resource available in the execution platform.

This technical report provides both descriptive and hands-on introductions to build application-specific FPGA systems using the high-level `OmpSs@FPGA` tool. The goal is to give the reader a baseline view of the process of creating an optimized hardware design annotating `C`-based code with `HLS` directives. We assume the reader has a working knowledge of C/C++, and familiarity with basic computer architecture concepts (e.g. speedup, parallelism, pipelining).

Keywords: `Xilinx FPGA`, `OmpSs@FPGA`, `Zynq-SoC`, `EuroExa`

Email addresses: `david.goz@inaf.it, ORCID:0000-0001-9808-2283` (D. Goz), `sara.bertocco@inaf.it, ORCID:0000-0003-2386-623X` (S. Bertocco), `giuliano.taffoni@inaf.it, ORCID:0000-0002-4211-6816` (G. Taffoni), `igor.coretti@inaf.it, ORCID:0000-0001-9374-3249` (I. Coretti)

Contents

1 **Introduction and motivation** **7**

2 **FPGA architecture explained to software developer** **8**

2.1 FPGA architecture . . . 8

2.2 Computation on an FPGA . . . 9

3 **FPGA design with Xilinx Vivado High-Level Synthesis** **10**

3.1 `Vivado HLS` design flow . . . 10

3.1.1 Input to the `HLS` stage . . . 10

3.1.2 Functional verification . . . 10

3.1.3 `HLS` . . . 10

3.1.4 C/RTL cosimulation (optional) . . . 12

3.1.5 Evaluation of the implementation and design iterations . . . 12

3.1.6 RTL export . . . 12

3.2 Coding style . . . 12

3.3 Latency and pipelining . . . 13

3.3.1 Unrolling . . . 14

3.3.2 Dataflow . . . 15

3.4 Throughput/Area tradeoff . . . 15

4 `OmpSs@FPGA` **ecosystem** **16**

4.1 `OmpSs@FPGA` source code example . . . 16

4.2 `OmpSs@FPGA` toolchain flow . . . 17

4.3 Execution model . . . 18

4.4 Support for heterogeneous platform . . . 19

4.5 Memory management . . . 19

4.6 Tracing the program execution . . . 20

4.7 Supported board . . . 20

5 **Experimental setup** **22**

5.1 System setup and requirements . . . 24

6 **The Zynq device** **28**

6.1 SoC with `Zynq` . . . 28

6.2 Processing System . . . 29

6.2.1 Processing System External Interfaces . . . 30

6.3 Programmable Logic . . . 30

6.3.1 Additional resources: `Block RAMs` and `DSP48s` . . . 32

6.4 PS – PL interfaces . . . 32
6.4.1 The standard AXI . . . 33
6.4.2 AXI interface and interconnection . . . 33
6.5 The Zynq-7000 series . . . 34

7 Basic algorithm case studies **35**
7.1 Loops . . . 35
7.1.1 Default loop synthesis . . . 35
7.1.2 Pipelining loop . . . 39
7.1.3 Unrolling loop . . . 42
7.2 Array-optimizations . . . 44
7.2.1 #pragma HLS array_map . . . 44
7.2.2 #pragma HLS array_partition . . . 44
7.2.3 #pragma HLS array_reshape . . . 46
7.2.4 #pragma HLS data_pack . . . 46
7.2.5 Vector addition using arrays partition . . . 48
7.3 Vivado HLS coding examples . . . 49

8 Algorithm case studies **49**
8.1 Running total . . . 49
8.2 Matrix-Vector multiplication . . . 54
8.2.1 Pipelining . . . 57
8.2.2 Unrolling . . . 58
8.2.3 Matrix-vector multiplication optimizations . . . 61
8.3 Matrix-matrix multiplication . . . 66
8.3.1 Background . . . 66
8.3.2 Block matrix multiplication . . . 67
8.4 Histogram . . . 70

9 Roofline model **79**
9.1 FPGA Empirical Roofline . . . 79
9.2 Results . . . 81
9.3 Energy . . . 83

10 Conclusions **84**

11 Acknowledgments **85**

12 Appendix: Compile `OmpSs@FPGA` programs **86**

12.1 Bitstreams 86

12.2 Hardware instrumentation 86

12.3 Binaries 86

Acronyms

The following abbreviations are used in this manuscript:

AA Astronomy and Astrophysics

ACP Accelerator Coherency Port

5 **AIT** Accelerator Integration Tool

ALU Arithmetic Logic Unit

APU Application Processing Unit

ARM Advanced RISC Machine

AXI Advanced eXtensible Interface

10 **BRAMs** Block RAMs

CLB Configurable Logic Block

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DRAM Dynamic Random Access Memory

15 **ECC** Error Correction Coding

EuroEXa co-designed innovation and system for resilient EXAscale computing in Europe: from application to silicon

FER FPGA Empirical Roofline

FF Flip-Flop

20 **FIFO** First In First Out

FLOPs FLoating point Operations Per Second

FMA Fused Multiply–Add

FPGA Field Programmable Gate Array

GCC Gnu Compiler Collection

25 **GPU** Graphic Processing Unit

HDL Hardware Description Language

HLS High Level Synthesis

HPC High Performance Computing

IOBs Input/Output Blocks

30 **LUT** Look-up table

MMU Memory Management Unit

MPI Message Passing Interface

MPSoC Multi Processing System-on-Chip

OCM On Chip Memory

35 **OpenCL** Open Computing Language

OpenMP Open Multi-Processing

PL Programmable Logic

PS Processing System

RAM Random Access Memory

40 **RISC** Reduced Instruction Set Computer

ROM read only Memory

RTL Register Transfer Language

SCU Snoop Controller Unit

SIMD Single Instruction Multiple Data

45 **SMP** Symmetric MultiProcessor System

SoC System On Chip

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

1. Introduction and motivation

FPGAs were already used in the past as accelerators in the context of physics simulations and scientific computing in general. The relevant programming effort to program FPGAs using Hardware Description Languages (HDLs) and the weak code portability had been the main stoppers towards a wide adoption of FPGAs in the High Performance Computing (HPC) context.

Nowadays, thanks to the High Level Synthesis `HLS` approach, FPGAs can be programmed using high level languages such as C/C++ and OpenCL, highly reducing the programming effort required, allowing a faster design space exploration in terms of FPGA resource-usage and performance, and a much higher software portability.

The enhanced compute capability of FPGAs, in conjunction with high energy-efficient hardware – thanks to their low clock frequency and intrinsic parallelism – are increasing the interest towards their usage as accelerator in HPC platforms [1].

The technical report has been done in the context of the EuroEXA project[1], aiming at developing a prototype of an exascale level computing architecture, delivering world-leading energy-efficiency. The compute units are based on multi-cores ARM processors combined with Xilinx FPGAs. `OmpSs@FPGA` is one of the high-level programming framework available on the EuroEXA platform that enables FPGA-accelerated computing.

The technical report is organized as follows. In Section 2 we describe the FPGA architecture and in Section 3 the `HLS` tool to create the detailed `RTL` micro-architecture on such a device. The Section 4 is devoted to present the directive based language `OmpSs`, developed by the Barcelona Supercomputing Center. We describe the `ZedBoard` development kit used in this Technical Report in Section 5. The Section 6 focuses on the Zynq SoC architecture hosted by the `ZedBoard`. Algorithm case studies are presented in Sections 7 and 8. In Section 9 we describe the *Roofline* model used to measure the computing performance and the energy-efficiency to be expected of an FPGA device. The Section 10 is devoted to the conclusions.

[1] https://euroexa.eu/

2. FPGA architecture explained to software developer

The aim of this section is to provide to the reader enough information about the FPGA architecture to understand the `HLS` compiler reports and effectively use the `HLS` directives, many of which are meant for target modern FPGA architecture features.

2.1. FPGA architecture

An FPGA is a type of integrated circuit made of logic blocks and memory elements that can be connected together using programmable interconnect. The FPGA architecture is based on the following elements:

- **Look-up table (LUT)**: it performs logic operations. An n-bit LUT can compute any n-input Boolean function. Larger FPGAs can have million of these elements;
- **Flip-Flop (FF)**: it acts as a register element storing the result of the LUT. LUTs can be replicated and combined with FFs to create more complex logic elements called a configurable logic block (Xilinx `Vivado HLS` tool uses the term slice);
- **Programmable interconnects (wires)**: they allow the connection between elements (or slices);
- **Input/Output (I/O) ports**: they get data in and out of the FPGA. The external device can be a microprocessor (e.g., an on-chip ARM processor using an AXI interface), or memory (e.g., off-chip DRAM memory controller).

Current FPGAs incorporate custom resources designed to perform specific task that increase the compute capability of the device.

An example of this is the DSP48 block available in Xilinx FPGAs. The DSP efficiently implements a series of arithmetic operations including addition, multiplication, and multiply-accumulate (MAC). So, the DSP is able to implement functions of the form:

$$p = a \times (b + c) + d \tag{1}$$

or

$$p \mathrel{+}= a \times (b + c) \tag{2}$$

Modern FPGAs have hundreds to thousands of these DSP48 distributed through their logic fabric.

Another example of additional resources are the embedded block RAMs (BRAMs). They are configurable random access memory banks, which typically are used to transfer data between the FPGA fabric and the microprocessor. Different locations of BRAMs can be accessed in parallel in the same clock-cycle due to the dual-port nature of these memories. A typical BRAM holds either 18k or 36k bits and can be cascaded with others to create larger memories. The BRAMs are co-located next to the DSP48[2]. Compared to the FFs, the BRAMs provide more storage capacity at the cost of limited total bandwidth, because only one of the two entries of the BRAMs can be accessed during every

[2] The UltraRAM blocks are BRAMs with a fixed configuration of 4,096 bits deep and 72 bits wide, available on Xilinx UltraScale+ device.

	External memory	BRAMs	FFs
count	1-4	thousands	millions
size	GBytes	KBytes	Bits
total size	GBytes	MBytes	100s of KBytes
total bandwidth	GBytes/sec	TBytes/sec	100s of TBytes/sec

Table 1: A comparison of the different off- and on-chip memories storage. Moving from left to right the storage capability decreases while the total bandwidth is increased.

clock-cycle. On the contrary, FFs on-chip can be read to and written to on every clock-cycle and thus providing a huge amount of total bandwidth. The decision about where to place the application's data is fundamental in terms of achievable performance and is one of the content covered by this technical report. The `HLS` tool provides many options that allows the programmer to specify exactly where and how to store data on the FPGA. Table 1 provides a comparison between different type of memories resources.

2.2. Computation on an FPGA

The FPGA device is able to implement any arithmetic and logical function that can run on a x86 processor. The main difference is that, for the processor, the source code is compiled to a sequence of instructions that execute on a fixed architecture, while for the FPGA, it is compiled to custom processing pipelines built up from the programmable resources. Usually, software developers put a lot of effort restructuring the algorithms in order to increase the temporal and spatial locality of data in memory to decrease the processor time spent per instruction and to increase the cache hit rate. This level of effort is not required when the same operation is implemented in an FPGA. Unlike a processor, where the same ALU performs different computations, in an FPGA independent sets of LUTs are instantiated for each operation in the software algorithm. For example, the addition of two 32-bit integers gets implemented by Xilinx `Vivado HLS` as 32 LUTs[3]. Moreover, the compiler arranges memories into multiple storage banks as close as possible to the point where the operation is performed (this is the reason for the high memory bandwidth shown in Table 1).

[3] As a rule of thumb, 1 LUT is equivalent to 1 bit of computation

3. FPGA design with Xilinx Vivado High-Level Synthesis

The job of the `Vivado HLS` compiler is to create an architecture from the resources available on the FPGA that
115 best fits the software. In contrast, with a processor, the job of the compiler is to best fit the software in the available processor architecture.

The software developer, knowing the FPGA architecture (Section 2), should guide the `Vivado HLS` compiler to create the best processing architecture.

Historically, the programming of an FPGA was based on Register-Transfer Level (`RTL`). It is analogous to Assembly
120 language, a low-level programming language designed for processors. Nowadays, the `HLS` approach enables the software developer to focus on algorithms rather than individual registers and cycle-to-cycle operations. Many commercial `HLS` tools now use C/C++, OpenCL, or SystemC as the input language.

Basically, `HLS` performs several steps automatically that `RTL` does manually, e.g., it analyzes and exploits the concurrency in an algorithm, implements interfaces to connect the rest of the system, and maps data onto storage
125 elements. The programmer should provide hints to the `HLS` tool in order to create the most efficient design (using `#pragma` directives).

3.1. `Vivado HLS` design flow

The full design `HLS` flow comprises many stages, that are summarized in the following subsections, and shown in Figure 1.

130 #### *3.1.1. Input to the `HLS` stage*

The input to the `HLS` process is a C/C++, OpenCL, or SystemC function.

3.1.2. Functional verification

First of all, it is necessary to verify the functionality of the target function, before beginning the process of synthesizing it into `RTL` code. The outputs of such a function can be checked using a testbench written in the same
135 high-level language.

3.1.3. `HLS`

The next step is performed by the `Vivado HLS` tool. It analysis and processes the target function, together with the directives supplied by the programmer, to create an `RTL` description of the circuit.

The first stage for `HLS` is to extract the `datapath` and the `control` from the nature of the algorithm. The first
140 refers to operations performed on the data samples, whereas the second to the circuity requested to coordinate the dataflow.

Then, the processes of `scheduling` and `binding` are undertaken iteratively:

- `Scheduling`: it is the conversion of `RTL` statements into a set of operations. Basically, it determines which operations occur during each clock cycle based on:

145
    - length of the clock cycle (or clock frequency);

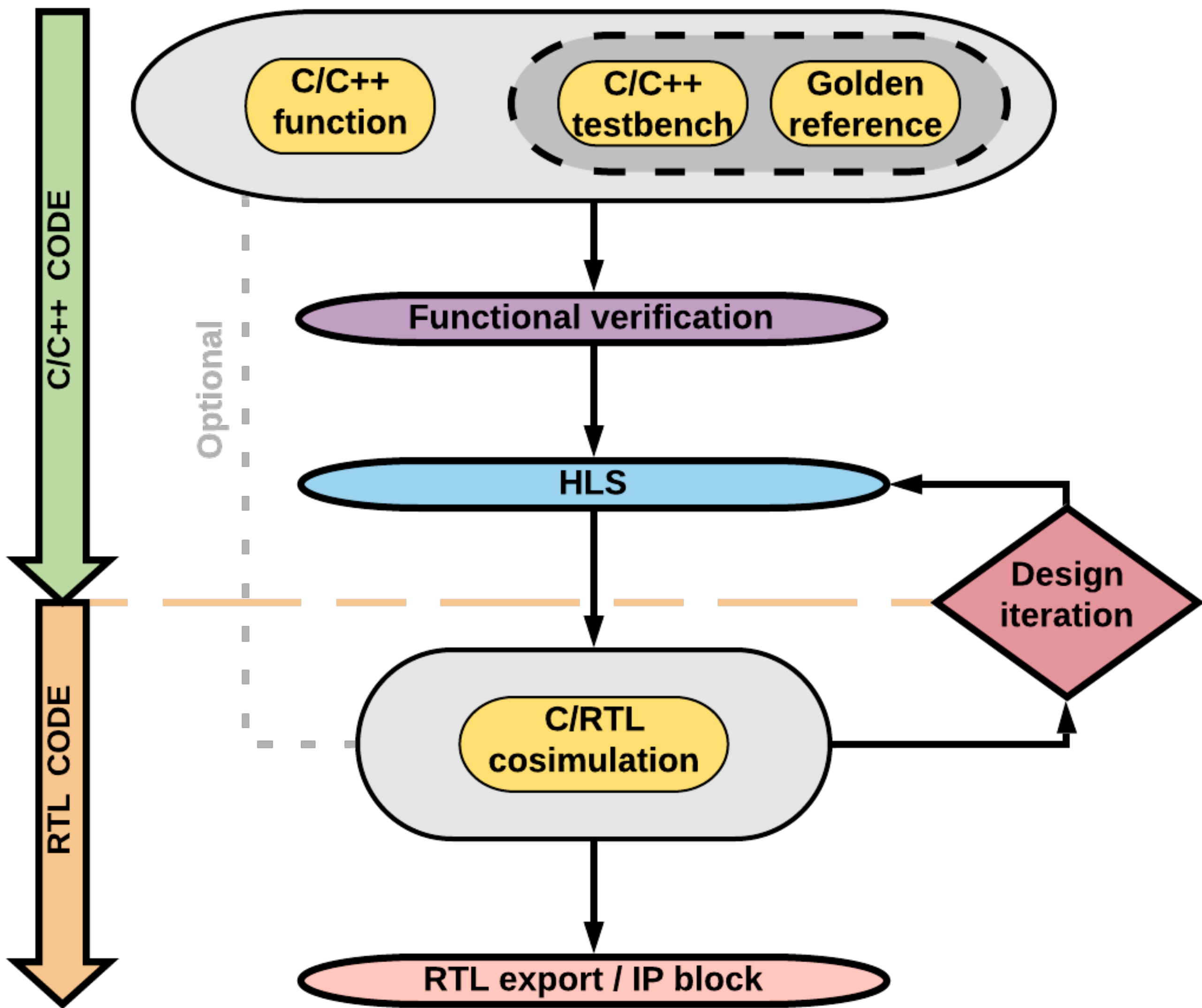

Figure 1: Sketch of the `Vivado HLS` design flow.

  - time it takes the operation to be performed (function of the target device);
  - user-specified optimizations directives (`#pragma HLS`).

  In general, the number of operations performed per clock cycle depends on the clock frequency and the target FPGA.

150 • `Binding`: it is the process of associating the scheduled operations with the resources of the target device. Hence, the resulting implementation has a set of characteristics, namely (i) latency, (ii) throughput, and (iii) physical resources utilized[4].

By default, the `Vivado HLS` will optimise area, i.e. it will adopt the strategy that consumes the fewest resources. The

[4]These concepts are explained later.

disadvantages of this method are its long latency and low throughput, which may not meet the requirements/constraints of the application. However, the software developer can drive the `HLS` processes of scheduling and binding using directives in the source code.

3.1.4. C/RTL cosimulation (optional)

The equivalent `RTL` design of the target function can be checked against the original code through the C/RTL cosimulation embedded in `Vivado HLS`. This stage re-uses the testbench in order to supply input to the `RTL` version generated by `HLS`.

3.1.5. Evaluation of the implementation and design iterations

After the (optional) check of the integrity of the design has been performed, it is also necessary to evaluate the performance of the implementation. For example, the amount of the resources it requires in the FPGA, the latency, and the maximum supported clock frequency.

If some of those (or all) do not accomplish the goals of the designer, multiple `HLS` iterations could be necessary, either re-engineering or tuning (e.g. using different `HLS` directives) the target function in order to find the "best" solution. Any changes to the code require functional re-verification, before the subsequent `HLS`, C/RTL verification, and implementation evaluation processes are again performed and iterated as needed.

3.1.6. RTL export

Once the desired design has been achieved, the `RTL` files are "packed" to create the so called Intellectual Property (IP) block, which can be integrated in other `Vivado` tools for inclusion in a FPGA-based system.

3.2. Coding style

`HLS` tools cannot handle any arbitrary software code. For example, in general they are not able to handle dynamic memory allocation, system calls, and there is often a limited support for standard libraries.

Following the guidelines of the `Vivado HLS` tool, we make the following assumption about functions on the source code:

- no dynamic allocation (no C/C++ function like `malloc(), calloc(), free(), new, and delete()`). The source code provided to the `HLS` tool must use compile time analyzable memory allocation;
- `HLS` compiler supports pointers that can be completely analyzed at compile time. Pointers-to-pointers cannot appear at the interface (pointers to external memory as any memory outside the scope of the function synthesized in hardware);
- system calls are not supported (e.g., `printf(), exit()`). They can only be used outside the scope of the synthesized code;
- limited support of standard libraries. Common functions declared in `math.h` are supported;
- recursive function calls are not supported.

The rest of the Section covers fundamental concepts that apply to FPGA-design.

3.3. Latency and pipelining

Latency is the number of clock cycles to complete an instruction or, in case of a function invocation, the time between when the function call is issued and the generation of its result value. In the technical report we use the term `task latency` meaning the number of clock cycles required to perform a given operation, and `task interval` meaning the time between when one task starts and the next starts. As a consequence, the computation is bounded by the task latency, but the start of a new task may not coincide with the writing of output by the old task. The task interval is related to the instruction pipelining technique. Pipelining means that the next instruction can be lunched in the execution queue before the current instruction is complete. In an FPGA, the overhead cycles associated with instruction fetch and instruction decoding are not present. Indeed, one of the key difference between a processor and an FPGA is the execution model. Regardless of the type of processor, the instruction stages are always the same, namely:

- **IF**: fetch the instruction from program memory;
- **ID**: instruction decoding;
- **EXE**: execute the instruction on hardware;
- **MEM**: memory operations for the next instruction;
- **WB**: write back the result of the instruction either to global/external memory (e.g. DRAM) or local registers.

The majority of modern processors are capable of running instructions with some degree of overlap, but the EXE stage, where resides the real application computation, executes sequentially. On the contrary, an FPGA executes the program on custom circuit for that program, therefore the **IF**, **ID**, and **MEM** stages are not present. The conclusion is that in an FPGA the task latency is measured by how many clock cycles it takes to run the EXE stage.

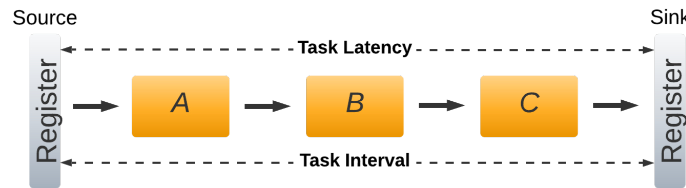

Figure 2: FPGA implementation without pipelining.

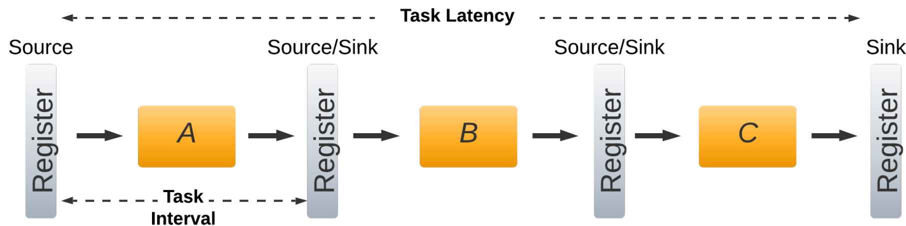

Figure 3: FPGA implementation with pipelining.

Figure 2 shows an hypothetical implementation of the EXE stage using three building blocks, namely A, B, and C. The task latency is the number of clock cycles it takes a signal to travel from the source to the sink register. In

this case, the HLS tool connects the functional blocks, so a second signal cannot start before the first has reached the sink, resulting in a task interval equal to the task latency.

Pipelining in an FPGA is the process of inserting more registers among building blocks. This partitioning could increase the number of the total clock cycle (i.e. increase the task latency), but allows to significantly reduce the task interval. Figure 3 shows a complete pipelining, meaning that a register is inserted between each functional block. Using this approach, each building block is allowed to always be busy, which enhances the application throughput at the cost of more FPGA resource usage.

Pipelining is a common technique to reduce the total latency of loops. For example, we assume that in our source code there is the following loop:

Listing 1: Example of loop C code

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

Without pipelining the task latency, $T_{L,\text{no pip}}$, (the number of clock cycles to perform the entire loop) is:

$$T_{L,\text{no pip}} = I_L \times N \tag{3}$$

where I_L is the iteration latency (the number of clock cycles to perform one iteration of the loop) and N is the loop count ("trip count"). The effect of pipelining is to reduce the task latency:

$$T_{L,\text{pip}} = I_L + (II \times (N-1)) \tag{4}$$

where II is the loop initialization interval that specifies the number of clock cycles between the start times of consecutive loop iterations. The minimum value for II is 1. As stated above, the I_L in the two cases could be different, because the insertion of registers can increase the iteration latency of the pipelining design. In the loop code 1, to achieve the desired II, the HLS tool analyzes the data dependencies and resource contention between consecutive loop iterations and acts as follows:

- to address data dependencies, HLS re-arranges the operations or queries the software developer for algorithm changes;
- to address resource contentions, HLS instantiates more copies of the resources (if available on the target FPGA) or queries the software developer for algorithm changes.

3.3.1. Unrolling

Another technique to decrease the loop latency is the loop unrolling. The purpose is to reduce the number of loop count and to execute different loop iterations at the same time. In this case FPGA resources cannot be shared among concurrent loop iterations and must be replicated.

Regarding Vivado HLS, it is worth to be noticed it is not possible to have nested pipelines and loops inside a pipeline. In the case there is a loop inside a pipeline it needs to be completely unrolled otherwise the pipeline cannot

be created. The complete unroll is done in automatic by `Vivado HLS`, but this implies that the inner loops need to have a loop count known at compile time, and that there are enough resources in the FPGA.

3.3.2. Dataflow

240 `HLS` is able to exploit parallelism of both loops and functions. With loops, task latency is typically reduced using pipelining and unrolling. With functions, all the stated operations are in the same hierarchical context, which might be confusing in terms of pipelining or unrolling. Dataflow optimization enables task-level pipelining allowing instructions in a block of code to overlap in their operations, increasing the overall throughput of the design. The purpose of the dataflow is to express parallelism at a coarse-grain level. The simplest case of parallelism is when functions of 245 a program work on a different data sets and do not communicate with each other, meaning that each function can be run independently. `Vivado HLS` analyzes the dataflow between sequential instructions and creates channels that allow consumer block of instructions to start operations before the producer block of instructions has completed. The parallelism is achieved by instantiating memory banks (BRAM memory) arranged in a ping and pong manner.

3.4. Throughput/Area tradeoff

250 The throughput is the rate at which the FPGA can process data. It is just a measurement of the processed volume of data per unit of time. In the example described in Figure 2 and 3, the task interval plays a key role in the obtained throughput. It is clear that the second implementation has higher performance, because it can accept a higher input data rate.

By default, `Vivado HLS` tool generates a largely sequential architecture, which tends to limit the number of func- 255 tional units used in the design (minimizing the area used in the FPGA). `HLS` can be directed to generate parallel architecture using techniques stated above (i.e. pipelining, unrolling, dataflow), increasing the resource usage (FPGA area) but also the overall performances. Writing highly optimized synthesizable `HLS` code could not be a straightforward process. First of all it requires a deep understanding of the algorithm of the application, and second the ability of change the code such that the `HLS` tool utilizes the re-engineered code in an effective way creating an optimized 260 hardware. Experience reveals that code designed for x86 processor yields very poor quality of results when converted into FPGA design, and restructured code typically differs substantially from the software implementation. However, restructuring the code using most of FPGA resources might cause issues during the stages of place and route, resulting in bitstream generation failure.

4. OmpSs@FPGA ecosystem

In this technical report we show how to use the directive based language, named OmpSs [2], developed by the Barcelona Supercomputing Center (BSC). OmpSs is based on task parallelism, a forerunner of OpenMP, with new directives extending other accelerator-based APIs like CUDA or OpenCL.

The application programming interface OpenMP[5] (Open Multi-Processing) supports multi-platform shared-memory multiprocessing programming in C/C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior, and it is designed to target multi-CPUs (OpenMP >= 5.0 supports CPUs/GPUs). CUDA[6] (Compute Unified Device Architecture) and OpenCL[7] (Open Computing Language) are the main tools used to program the General-Purpose Graphics Processing Units (GPGPUs). While the OpenCL framework allows running the same code on all GPU manifacturers (e.g. Nvidia, ADM, Intel) and multi-core CPUs too, the CUDA tool targets specifically Nvidia-GPUs.

With OmpSs it is possible to offload tasks among the host SMP cores, or integrated/discrete GPUs and/or FPGAs. OmpSs@FPGA[8] uses the vendor Xilinx Vivado and Vivado HLS tools[9], to generate the hardware configuration from high-level source code. OmpSs environment is built on top of Mercurium compiler[10] and Nanos++ runtime system[11]. The first is a source-to-source compilation infrastructure aimed at fast prototyping, supporting C/C++ and Fortran languages. The latest is a scheduler designed to serve as runtime support in parallel environments.

4.1. OmpSs@FPGA source code example

The following code, customized from the OmpSs@FPGA user guide[12], shows an implementation of the dot product using an FPGA task.

Listing 2: OmpSs@FPGA source code example of the implementation of the dot product using an FPGA task.

```
typedef float MyData;
const int BSIZE = 64;
#pragma omp target device(fpga) num_instances(2)
#pragma omp task in([BSIZE]v1, [BSIZE]v2) inout([1]result)
void dotProduct(const MyData *const restrict v1,
                const MyData *const restrict v2,
                      MyData *const restrict result)
{
  MyData resultLocal = *result;

  for (unsigned short int i=0 ; i<BSIZE ; i++)
```

[5] https://www.openmp.org/
[6] https://developer.nvidia.com/cuda-zone
[7] https://www.khronos.org/opencl/
[8] Online documentation at: https://pm.bsc.es/ompss-at-fpga.
[9] Online documentation available at: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.
[10] Online documentation available at: https://pm.bsc.es/mcxx.
[11] Online documentation available at: https://pm.bsc.es/nanox.
[12] Online documentation available at: https://pm.bsc.es/ftp/ompss-at-fpga/doc/user-guide/

```
295     {
          resultLocal += (v1[i] * v2[i]);
        }

      *result = resultLocal;
    }
300
    int main()
    {
      const int vecSize = 256;
      MyData v1[vecSize];
305   MyData v2[vecSize];

      /* Initialize the vectors */
      /* ...                    */

310   MyData result = 0;
      for (int i=0 ; i<vecSize ; i+=BSIZE)
        {
          dotProduct(v1+i, v2+i, &result);
        }
315    #pragma omp taskwait

      /* Somehow use the result value */
      /* ...                          */

320   return 0;
    }
```

In the source code 2, the function *dotProduct* is defined as a `OmpSs@FPGA` task (also called FPGA *kernel*) with input dependencies $v1$ and $v2$ and input/output dependency *result*. Each call to this function in the *main* function will generate a task that will be run as soon as its dependencies are ready. The task can be executed concurrently by two instances in the FPGA (if there are enough resources available). The programmer can also annotate the source code by using some additional directives (`#pragma HLS` not related to `OmpSs` programming model, but understandable by the `Vivado HLS` tool during the creation of the bitstream for the FPGA.

Through the technical report, we will show the most widely applied `#pragma HLS` directives, discussing in details how they can be used to optimized the accelerated kernels.

4.2. OmpSs@FPGA toolchain flow

The source-to-source `OmpSs` compiler (`Mercurium`) splits the host code and the fpga kernels into separate files. The part to be run in the host SMP is annotated with all the runtime calls and then compiled with `GCC`. The Accelerator Integration Tool (`AIT`) takes care of the part related to FPGA. `AIT` interfaces with `Vivado` tools (`Xilinx Vivado Design Suite`) to generate the FPGA bitstream. Moreover, the `Mercurium` compiler produces the wrapper function for each accelerated task, used to invoke the FPGA from `Nanos++` runtime system. More in detail, `AIT` first invokes

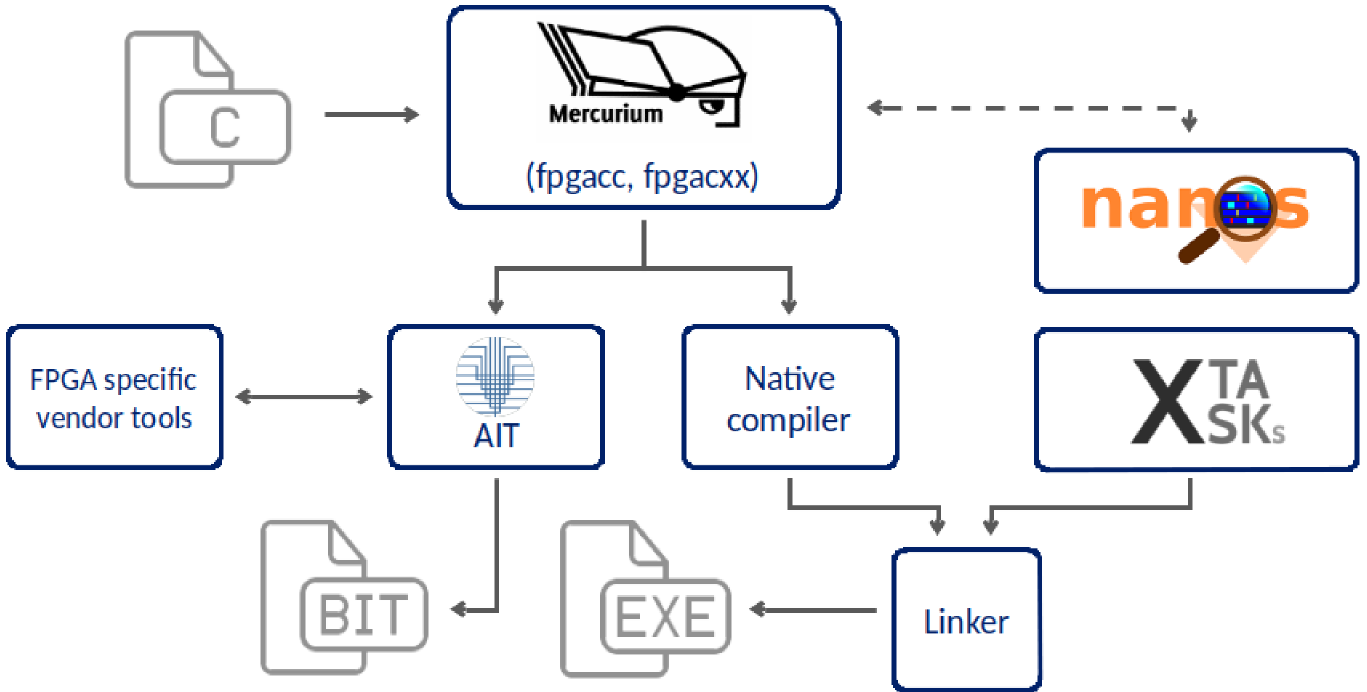

Figure 4: OmpSs@FPGA toolchain flow. Credit https://pm.bsc.es/ompss-at-fpga.

Vivado HLS to synthesize the kernel creating the Intellectual Property (IP) core, and then Vivado Design to connect the IP core with local memory (BRAM) and AXI ports (they allow the communications between the host and the accelerator), generating the FPGA bitstream. All the compilation process is done automatically by AIT following the directives added by the programmer. Figure 4 shows the toolchain flow.

340 *4.3. Execution model*

Nanos++ runtime system takes care of executing tasks as annotated by the programmer in the source code mapping them to the available resources in the compute platform. Nanos++ and OpenMP environment differ in the execution model. The former employs the thread-pool execution model, while the latter implements the fork-join parallelism. When programming using OpenMP, the software developer has to define which section of the code will be executed in
345 parallel, then explicitly express how the inner code has to be executed by threads in the parallel region (adding specific directives if necessary). OmpSs simplifies part of this job by providing an environment where parallelism is implicitly assumed and created from the beginning of the execution of the program, thus the programmer is allowed to omit the declaration of parallel regions. The definition of a parallel region is performed using the concept of task, which is a block of code that can be executed asynchronously in parallel, following an explicit dependency mechanism added by
350 the programmer to guarantee a correct execution.

This means that, the Nanos++ environment has:

- the thread-team created by default when the program execution starts;
- the dependence graph used to organize tasks and resolve data dependencies among them;
- the task-pool that will be executed by threads.

355 The duty of `Nanos++` is also to take care of the possible heterogeneity expressed by the programmer, that could target GPUs/FPGAs accelerators, managing the necessary memory transfers.

4.4. Support for heterogeneous platform

Listing 3: `OmpSs@FPGA` source code example of heterogeneous execution.

```
#define N 128
const unsigned int SIZE = N;

float x[N];
float y[N];
float z[N];

int main()
{
  #pragma omp target device(smp)
  #pragma omp task inout([SIZE]x)
  do_computation_CPU(x);

  #pragma omp target device(fpga)
  #pragma omp task inout([SIZE]y)
  do_computation_fpga(y);

  #pragma omp target device(smp,fpga)
  #pragma omp task inout([SIZE]z)
  do_computation_CPU_fpga(z);

  #pragma omp taskwait

  /* Somehow use the results */
  /* ... */

  return 0;
}
```

Figure 3 shows an example code includes `target device` directives that indicate two target devices for the defined tasks. One that must be run on CPU, which is marked as `target device(smp)`, a second that must be run on a FPGA, marked with `target device(fpga)`, and a third that can be run either in the CPU or the FPGA. The latest task creates instances of the same type, which can potentially run in parallel at runtime in both devices. In general 390 the clause `device(target-device)` allows to specify on which device should be targeted the construct. If no device is specified then the SMP device is assumed.

4.5. Memory management

`OmpSs` is able to handle architectures with disjoint memory address spaces like heterogeneous systems built around accelerators with private memory, freeing the programmer to explicitly expose the underlying memory hierarchy.

395 OmpSs hides the existence of other memory spaces present on the system, but, the programmer has to specify the data each task is accessing in order to correctly support the platform.

There is a set of directives aimed at specifying data task dependencies. The clauses are the following:

- `copy_in(list-of-variables)`: data should be copied to the device before the associated kernel is executed;
- `copy_out(list-of-variables)`: data should be copied from the device to the host after the kernel execution;
- 400 `copy_inout(list-of-variables)`: this clause is the combination of the previous ones;
- `copy_deps`: if the attached construct has any dependence clauses then they require, at runtime, device space memory for copies between host memory and device local memory (i.e. in will also be considered `copy_in`, output will also be considered `copy_out`). `copy_deps` is automatically deactivate if `copy_in/out/inout` are specified.

In all cases the runtime, managed by Nanos++, takes care of allocating device memory and perform data movements 405 between host and device memories for the list of variables labeled as copies. It is also possible to explicitly allocate device memory using Nanos++ runtime APIs so that the device can access them. In any of the task variables is a scalar, then it is directly passed to the accelerator without the need of being listed.

4.6. Tracing the program execution

OmpSs@FPGA ecosystem allows to trace Nanos++ threads running in cores and/or FPGA. The programmer can 410 analyze both the application and runtime internals such as the creation of tasks, task executions, synchronizations, etc. Regarding FPGA, the current support provides the user with the information of execution time and data movements. The FPGA execution trace is generated using an internal tracing library (the programmer has only to activate the corresponding compilation flag), the information is merged with the host information using the Extrae tool[13] and can be visualized with the Paraver tool[14].

415 *4.7. Supported board*

At the time of writing, OmpSs@FPGA has been successfully tested in the following boards:

- Zynq-7000 Family
  - ZedBoard
  - Zynq 702
  - 420 Zynq 706
  - Diligent Zybo Zynq 7000
- Virtex-7 Family
  - Alpha Data ADM-PCIE-7V3

[13]Extrae is available at: https://tools.bsc.es/extrae.
[14]Paraver is available at: https://tools.bsc.es/paraver.

- Zynq-Ultrascale+ Family
  - SECO Axiom Board
  - Trenz Electronics Zynq U+
  - Zynq U+ ZCU102

Programmers interested in `OmpSs@FPGA` and want more information, have to mail to the `OmpSs` support team: `ompss-fpga-support@bsc.es`. They provide:

- `Docker image`: it contains a pre-configured environment to build `OmpSs@FPGA` applications targeting the chosen board. The prerequisites required are the following:
  - Docker;
  - Xilinx Vivado Design Suite (recommended 2017.X or higher)[15];
  - Vivado licence for the target device. Academic licenses for Xilinx software and IP and low cost Xilinx FPGA and Zynq SoC development kits can be obtained through the Xilinx University Program[16];
  - [Optional for BOOT.bin generation] Xilinx Petalinux.

  The image contains different `Mercurium` installations to cross-compile applications for each architecture. In addition, it contains the required libraries in the */opt/install-XXX* folder. The image does not contain an installation of Xilinx tools (see above);

- `SD image`: it contains the OS of the target board (usually Ubuntu 16.04 LTS). The user should use both Docker and SD image from the same release.

[15] Xilinx tool can be downloaded from `https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html`

[16] Xilinx University Program: `https://www.xilinx.com/support/university.html`.

5. Experimental setup

Figure 5: `ZedBoard` development kit. It includes the Avnet ZedBoard 7020 baseboard, 12 V power supply, 4 GB SD Card, a micro-usb cable, USB adapter (male micro-B to female standard-A).

In this Section we detail the software and hardware setup that we used to exploit the Xilinx FPGA using `OmpSs@FPGA`:

- *Software stack on the development x86 computer*:

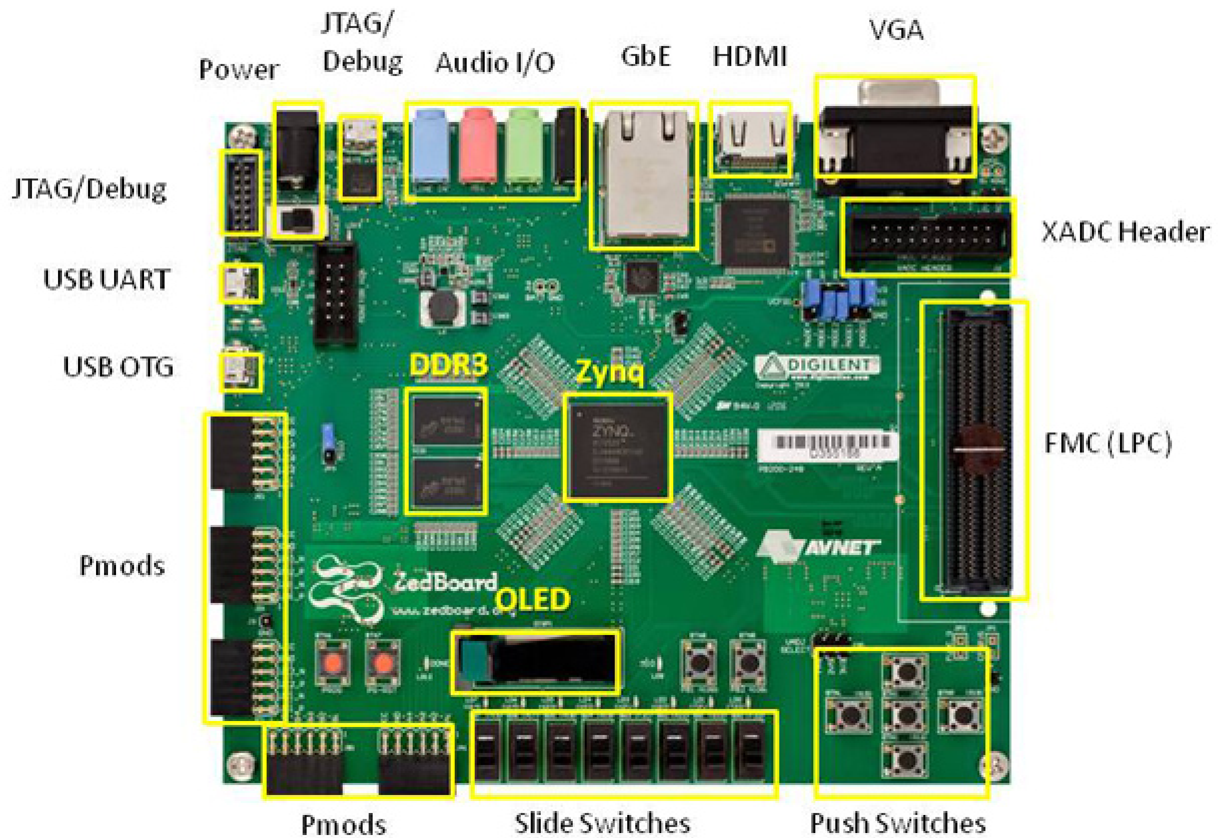

Figure 6: Functional overlay of the `ZedBoard`.

- `Ubuntu 18.04 LTS` (64 bit). Xilinx tools support also Linux CentOS 6.x/7.x, SUSE Linux Enterprise 11.4 and 12.3, Red Hat Enterprise Workstation 6.x/7.x;
- `Vivado Design Suite 2017.3` (Linux version), downloadable from the software webpage `https://www.xilinx.com/products/design-tools/vivado.html`. From the webpage download the Linux Self Extracting Web Installer. Release notes, installation and licensing can be found in the *Vivado Design Suite User Guide* (`https://www.xilinx.com/support.html#documentation`). It is possible to get an evaluation licence with 30 days expiration, or an academic license through the Xilinx University Program.
- `Docker version 19.03.6`;
- `OmpSs@FPGA` version 2.1.0. The `OmpSs` support team provides the docker image;
- `Minicom tool`, a text-based serial port communications program. It is used to talk to external devices and serial console ports. We use it to communicate with the board hosting the FPGA.

• *`ZedBoard` FPGA board*:

for testing OmpSs@FPGA we use the ZedBoard[17], a complete development kit for designers interested in exploring designs using the Xilinx Zynq-7000 All Programmable System on Chip (SoC). The board contains all the necessary interfaces and supporting functions to enable a wide range of applications. The expandability features of the board make it ideal for rapid prototyping and proof-of-concept development The kit is shown in Figure 5.

- *Features* (functional overlay of the board is shown in Figure 6):
  - Zynq-7000 SoC family, which integrates the software programmability of an ARM-based dual core processor with the hardware programmability of an FPGA;
  - memory:
    - 512 MB DDR3;
    - 256 Mb Quad-SPI Flash;
    - 4 GB SD card.
  - onboard USB-JTAG programming;
  - 10/100/1000 Ethernet;
  - USB OTG 2.0 and USB-UART;
  - PS & PL I/O expansion;
  - multiple displays (1080p HDMI, 8-bit VGA, 128 x 32 OLED);
  - I2S Audio CODEC.
- *Target applications*:
  - software accelerations (the content of the technical report);
  - video processing;
  - embedded ARM processing;
  - motor control;
  - general Zynq-7000 All Programmable SoC prototyping.

The board runs an Ubuntu Linux 16.04 LTS, provided by the OmpSs support (SD image version 2.1.0.), where we perform the test of our software. We also use performance tools Extrae and Paraver to analyze the application behaviour.

5.1. System setup and requirements

The hardware specification of the development x86 computer is significant, as this influences execution times when running the Xilinx design tools.

Even if Xilinx allows to run its software in any x86 computer, our experience suggests that at least 16 GB of RAM in the development x86 computer is recommended for the (small) ZedBoard device. Generally speaking, at least quad-core processor is recommended, since some of the design tools have multi-core processing support.

To effectively exploit the ZedBoard the following steps are required:

[17] ZedBoard webpage: http://zedboard.org/product/zedboard.

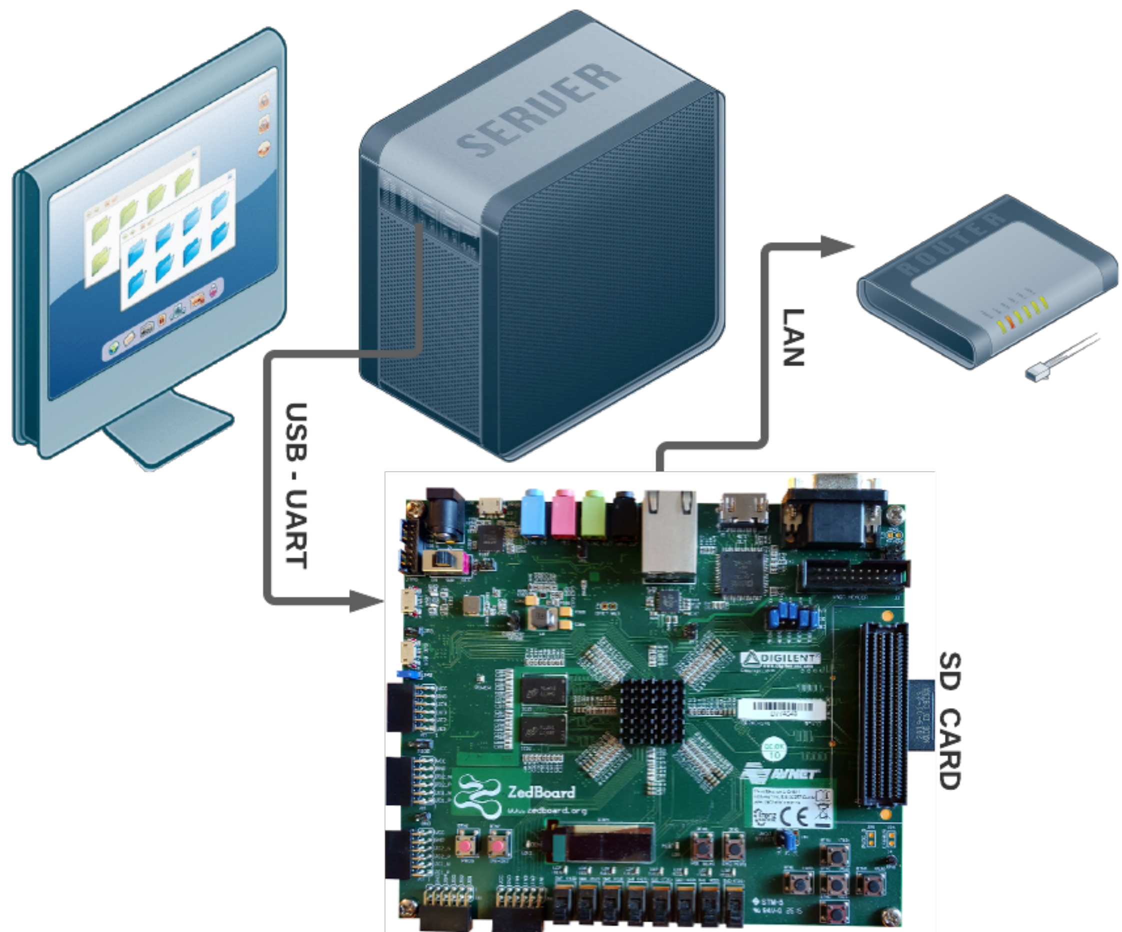

Figure 7: Hardware development setup.

- be sure that the board is turned off;
- insert the SD card in the board (it contains the OS provided by the `OmpSs@FPGA` support team). Set jumpers for boot from SD card as in Figure 8;
- connect the board to the power supply;
- connect the microUSB side of the microUSB–USB cable to UART connection (see Figure 7);
- connect the USB side of the microUSB-USB cable to the development x86 computer;
- turn on the board;
- run on the development x86 computer the " `dmesg` " command and look for something like: *... ttyACM0: USB ACM device ...* (See Figure 9 for an example). This is required to setup the `minicom` port;

- assuming that the `dmesg` log reports *ttyACM0*, run the command `" sudo minicom -D /dev/ttyACM0 -b 115200 "`. The `ZedBoard` boots up the OS (see Figure 10);
- `ssh` and `scp` are available through the ethernet connection. This will be useful to transfer files (in particular the bitstream for the FPGA) between the development x86 computer and the `ZedBoard`.
- when the `ZedBoard` is no longer needed, shut down the board with either the command `" sudo halt "`, or `" sudo shutdown now "`.

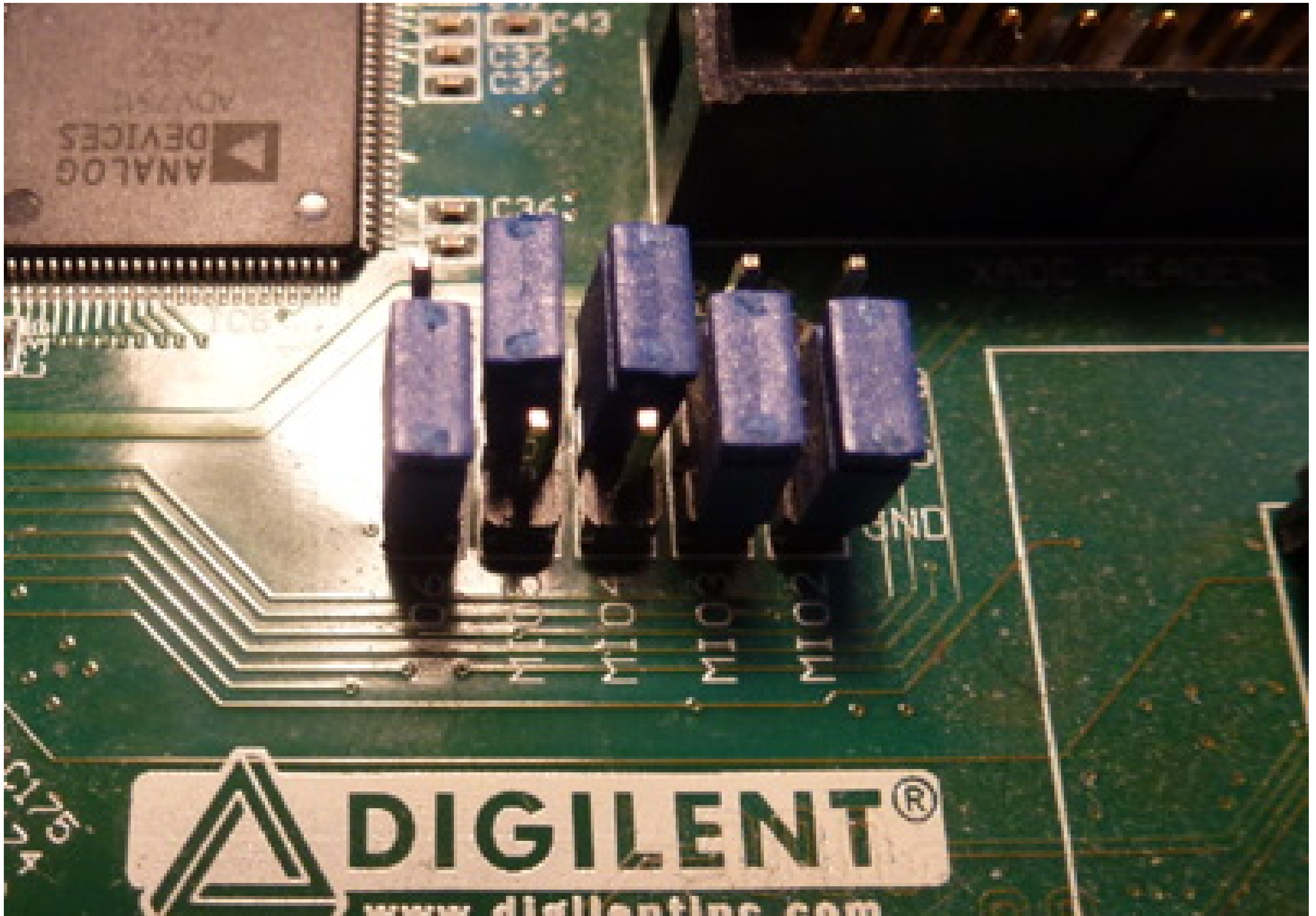

Figure 8: Jumper setting for boot from SD card.

```
[14881.186951] usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=4
[14881.186953] usb 1-2: Product: Cypress-USB2UART-Ver1.0G
[14881.186954] usb 1-2: Manufacturer: 2012 Cypress Semiconductor
[14881.186955] usb 1-2: SerialNumber: 046405034720
[14881.217479] cdc_acm 1-2:1.0: ttyACM0: USB ACM device
[14881.219041] usbcore: registered new interface driver cdc_acm
[14881.219042] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
[15909.053334] usb 1-2: USB disconnect, device number 4
[15909.053452] cdc_acm 1-2:1.0: failed to set dtr/rts
[16073.083009] usb 1-2: new full-speed USB device number 5 using xhci_hcd
[16073.256481] usb 1-2: New USB device found, idVendor=04b4, idProduct=0008, bcdDevice= 0.00
[16073.256483] usb 1-2: New USB device strings: Mfr=1, Product=2, SerialNumber=4
[16073.256484] usb 1-2: Product: Cypress-USB2UART-Ver1.0G
[16073.256484] usb 1-2: Manufacturer: 2012 Cypress Semiconductor
[16073.256485] usb 1-2: SerialNumber: 046405034720
[16073.276529] cdc_acm 1-2:1.0: ttyACM0: USB ACM device
darkenergy@EMPOKNOR:~$
```

Figure 9: Example for output of the `dmesg` command issued on the development x86 computer connect via the UART port to the `ZedBoard`.

```
         Mounting /tmp...
[  OK  ] Mounted /tmp.
[  OK  ] Mounted /var/log.
[  OK  ] Started udev Coldplug all Devices.
[  OK  ] Started udev Kernel Device Manager.
         Starting Flush Journal to Persistent Storage...
[  OK  ] Reached target Local File Systems.
         Starting Raise network interfaces...
[  OK  ] Found device /dev/ttyPS0.
[  OK  ] Started Flush Journal to Persistent Storage.
         Starting Create Volatile Files and Directories...
[  OK  ] Started Create Volatile Files and Directories.
[  OK  ] Started ifup for eth0.
         Starting Network Time Synchronization...
         Starting Update UTMP about System Boot/Shutdown...
[  OK  ] Started Network Time Synchronization.
[  OK  ] Started Update UTMP about System Boot/Shutdown.
[  OK  ] Reached target System Initialization.
[  OK  ] Reached target Basic System.
         Starting LSB: Set the CPU Frequency Scaling governor to "ondemand"...
         Starting getty on tty2-tty6 if dbus and logind are not available...
[  OK  ] Started Daily Cleanup of Temporary Directories.
         Starting Permit User Sessions...
[  OK  ] Reached target System Time Synchronized.
[  OK  ] Started Daily apt activities.
[  OK  ] Reached target Timers.
[  OK  ] Started Permit User Sessions.
[  OK  ] Started LSB: Set the CPU Frequency Scaling governor to "ondemand".
[  OK  ] Started getty on tty2-tty6 if dbus and logind are not available.
[***   ] A start job is running for Raise network interfaces (9s / 5min 1s)EXT4-fs error (device mmcblk0p2): ext4_mb_generate_buddy:758: group 1,������v��
[  OK  ] Started Raise network interfaces.
[  OK  ] Reached target Network.
         Starting OpenBSD Secure Shell server...
         Starting /etc/rc.local Compatibility...
[  OK  ] Started /etc/rc.local Compatibility.
[  OK  ] Started Getty on tty6.
[  OK  ] Started Getty on tty3.
[  OK  ] Started Serial Getty on ttyPS0.
[  OK  ] Started Getty on tty4.
[  OK  ] Started Getty on tty5.
[  OK  ] Started Getty on tty1.
[  OK  ] Started Getty on tty2.
[  OK  ] Reached target Login Prompts.
[  OK  ] Started OpenBSD Secure Shell server.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Update UTMP about System Runlevel Changes.

Ubuntu 16.04 LTS zynq-bsc ttyPS0

zynq-bsc login:
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyACM0
```

Figure 10: `ZedBoard` connected through `minicom` connection to the development x86 computer.

6. The Zynq device

The Zynq combines a dual-core ARM Cortex-A9 (32 bit) processor with an FPGA logic fabric. The ARM Cortex-A9 is an application grade processor, capable to run a full OS such as Linux, while the programmable logic is based on Xilinx 7-series FPGA architecture. The two part of the device are connected through Advanced Extensible Interface
510 (AXI) interfaces, which provide high bandwidth and low latency connections.

6.1. SoC with Zynq

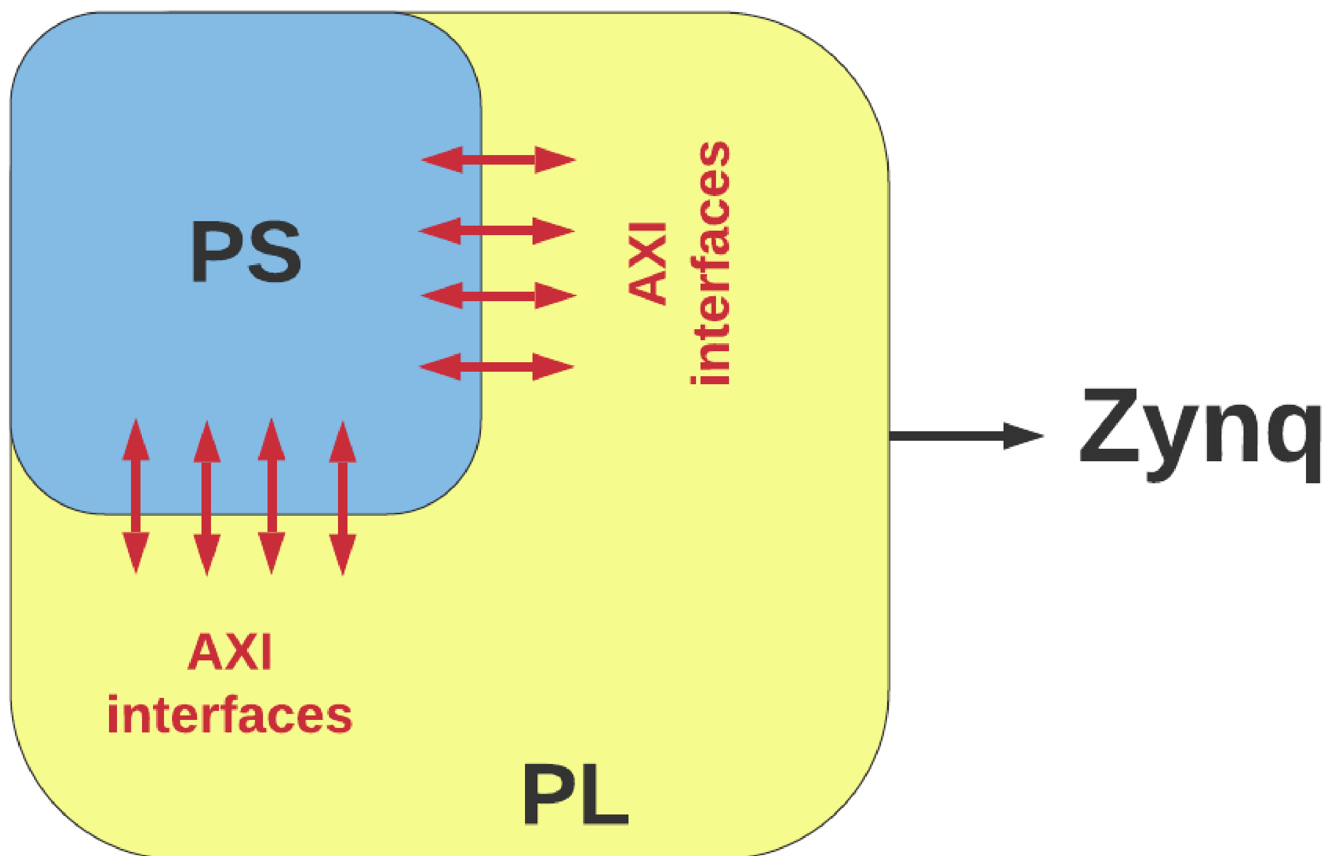

Figure 11: Sketch of the Zynq SoC architecture.

The idea behind SoC is that single silicon chip can be used to implement the functionality of an entire system, rather than several different physical chips being required. A SoC can combine physically separated devices together into a system. The SoC solution is a low-cost, it enables faster data transfers between the various system elements, it has
515 lower power consumption, smaller physical size, and better reliability. The high level model of the Zynq architecture is sketched in Figure 11 and comprises two parts: a Processing System (PS) formed around a dual-core ARM Cortex-A9 (32 bit) processor, and a Programmable Logic (PL), which is the FPGA. It also features integrated memory and high-speed communication interfaces, called Advances eXtensible Interface (AXI) connections. OS, applications are hosted on the PS, whereas the PL gives to the designer a "black canvas" where to create custom peripherals (also

520 called IP functional blocks), or to reuse standard ones. The power circuitry is configured with separate domains for each, enabling either the PS or PL to be powered down if not in use.

In the next subsections we give an in-depth description of the Zynq. Extended information can be found in the *Zynq-7000 Technical Reference Manual*[18].

6.2. Processing System

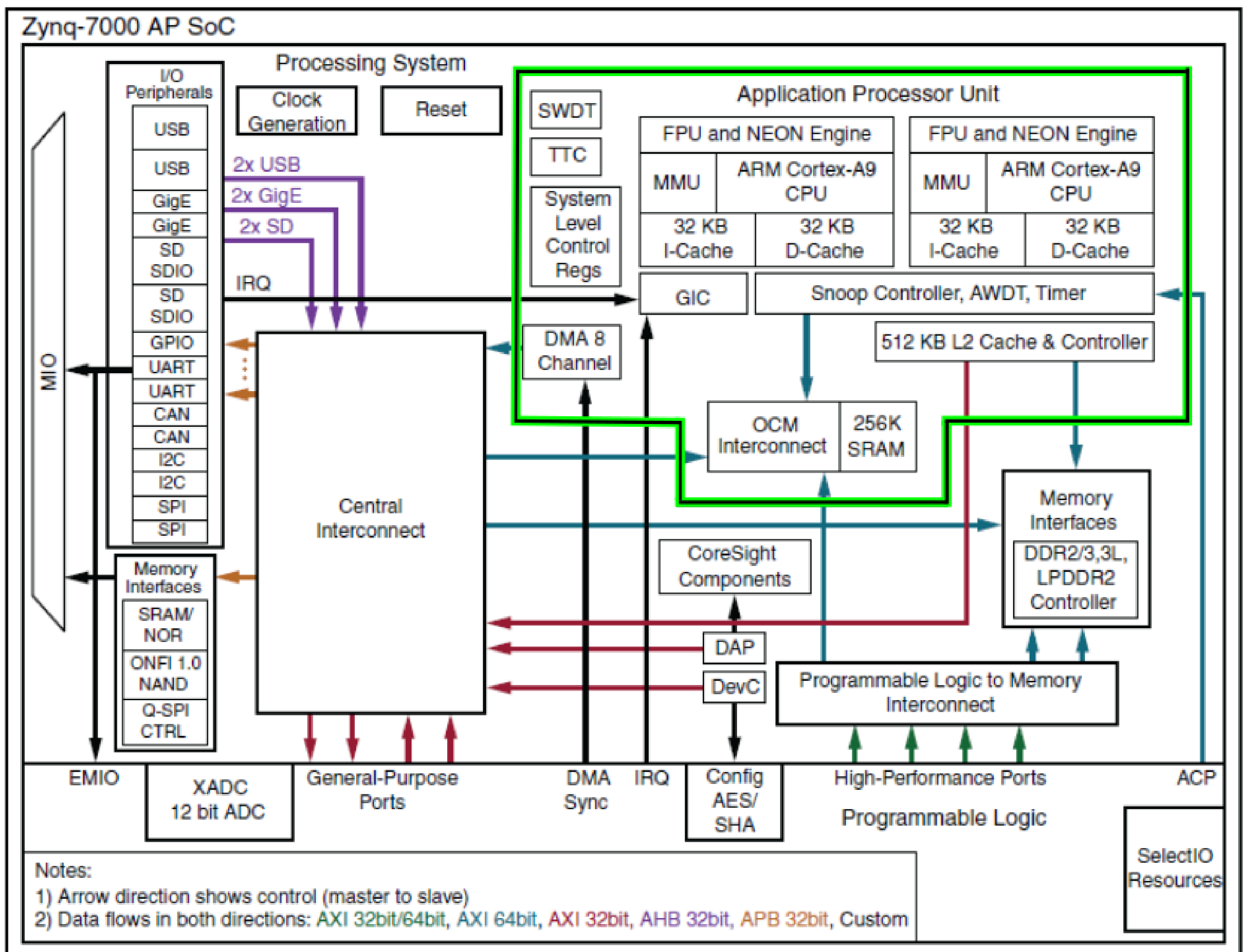

Figure 12: The Zynq Processing System. Credit Xilinx.

525 The Zynq PS encompasses not just the ARM processor, but a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces. A block diagram of the PS is shown in Figure 12, where the APU is highlighted in green.

The APU is composed by two ARM processing cores, each with associated computational units:

[18]Zynq-7000 Technical Reference Manual: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

- `NEON Media Processing Engine (MPE)` and `Floating Point Unit (FPU)`. NEON engine provides Single Instruction Multiple Data (SIMD) facilities, so it can accept multiple sets of input vectors upon which the same operation is performed simultaneously. NEON instructions can be explicitly used, or inferred by the compiler. NEON supports a variety of data types including integers and single precision point, but not double precision. The FPU, which does not have SIMD capability, is required if double precision computation is needed. The FPU unit provides hardware acceleration for floating point IEEE-754 compliant operations in single and double precision;
- `Memory Management Unit (MMU)`, which translates between virtual to physical addresses;
- `Level 1 cache memory`, 32KB for each core, in two sections for data and instructions;
- `Level 2 cache memory`, 512KB shared between cores;
- `On Chip Memory (OCM)`, 256KB on-chip memory;
- `Snoop Controller Unit (SCU)`, which interfaces the processors with L1 and L2 memories (cache coherency). SCU manages transactions that take place between the PS and PL through the Accelerator Coherency Port (ACP).

6.2.1. Processing System External Interfaces

Communications between the PS and the external interfaces is achieved via the Multiplexed Input/Output (MIO), or through the Extended MIO (EMIO), both shown at the left hand side of Figure 12.

The available I/O includes standard communications interfaces. The complete set of I/O peripheral interfaces can be found in the *Zynq-7000 Technical Reference Manual*. For our purposes we use:

- `Universal Asynchronous Receiver Transmitter (UART)`: it is a low data rate modem interface for serial communication, and it is used for terminal connections to a host PC;
- `SD`: for interfacing with SD card;
- `GigE`: Ethernet MAC peripheral, supporting 10Mbps, 100Mbps, and 1Gps modes.

6.3. Programmable Logic

The general FPGA architecture has been already presented in Section 2, thus, in this Section, we focus on the features of the `Zynq` PL, using Xilinx-specific terms.

The second part of the `Zynq` SoC is the PL. This is based on the Artix-7 and Kintex-7 FPGA fabric. Features of the PL are the following:

- `Configurable Logic Block (CLB)`: they are small groups of logic elements connecting among them through programmable interconnects. Each of them contains two logic *slices*. Figure 13 shows a sketch of a CLB;
- `Slice`: a sub-unit within the CLB composed by 4 *Lookup Tables*, 8 *Flip-Flops* and other logic;
- `Lookup Table (LUT)`: a flexible resource capable of implementing:

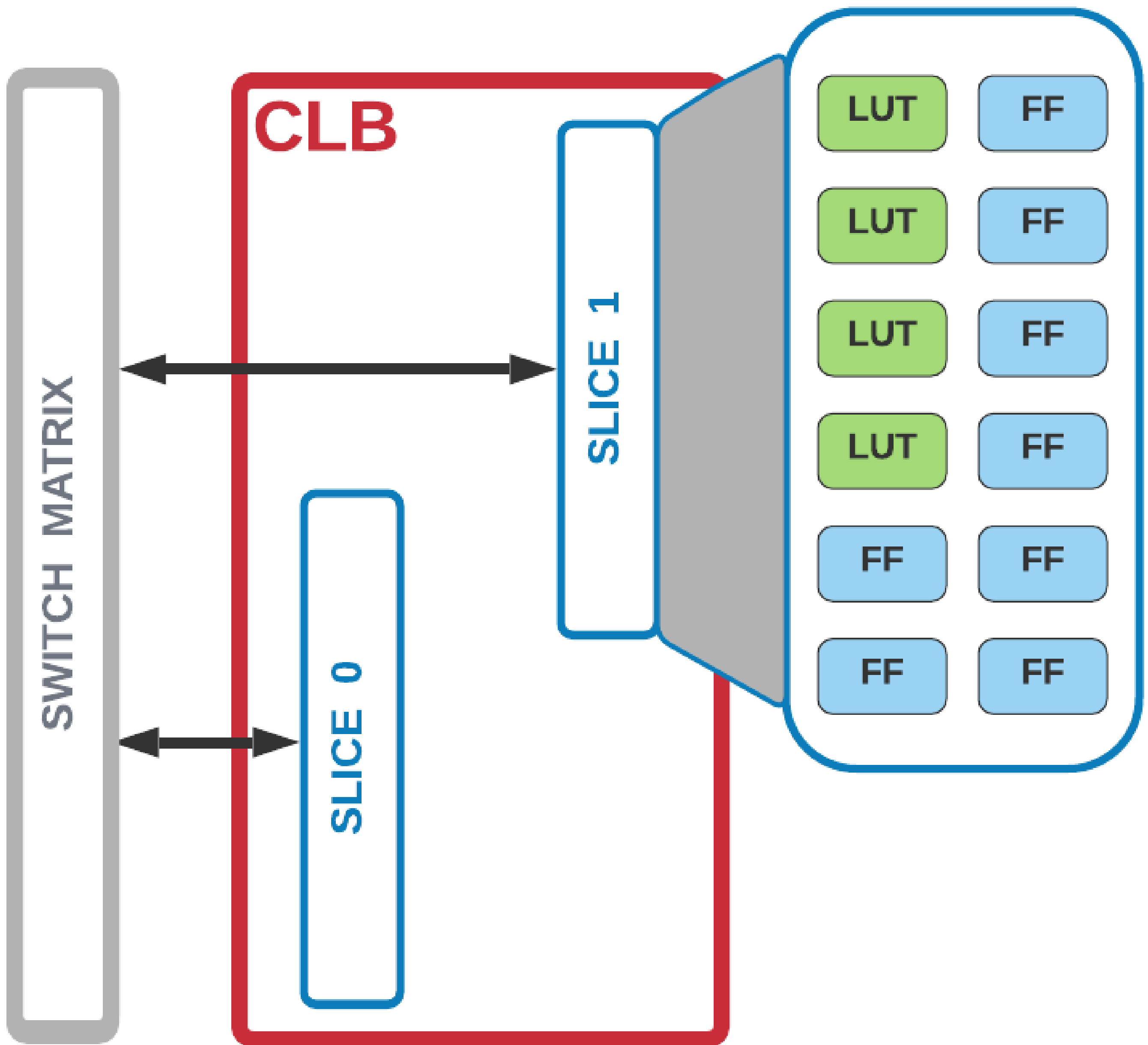

Figure 13: Sketch of the composition of a `Configurable Logic Block` (CLB).

i) shift register;

ii) logic function;

iii) small Read Only Memory (ROM);

iv) small Random Access Memory (RAM).

565 LUTs can be combined together to form a larger unit as required;

- `Flip-Flop (FF)`: a circuit element implementing a 1-bit register;
- `Switch Matrix`: it provides a flexible routing connections between elements within a CLB, and from one CBL to other resources within the PL;

- `Carry Logic`: it comprises a chain of routes and multiplexers to link slices in a vertical column;

570 - `Input/Output Blocks (IOBs)`: they provide interfacing between the PL and the external circuitry.

From the point of view of the software developer, there is not the need to specifically target these resources, because the Xilinx tools (called by the `OmpSs` AIT wrapper) automatically infer the required LUTs, FFs, IOBs, etc. from the design, and map them accordingly.

6.3.1. Additional resources: `Block RAMs` and `DSP48s`

575 The `Zynq` PL hosts special resources: `Block RAMs (BRAMs)` for high-speed memory requirements, and `DSP48s` slices for high-speed arithmetic. The two components are integrated into the PL normally in proximity to each other. `BRAMs` can implement Random Access Memory (RAM), Read Only Memory (ROM), and First In First Out (FIFO) buffers, also supporting Error Correction Coding (ECC). Each `BRAM` can store up to 36Kb of data, and can be configure either as two independent 18Kb RAMs, or one 36Kb RAM. Each RAM comprises 2048 memory elements of default size of 580 18-bits, however can be configured such that it contains smaller (e.g. 4096 elements x 9 bits), or larger elements (e.g. 1024 elements x 36 bits). It is possible to form larger capacity memories combining more `BRAMs` together.

Data can also be stored into `distributed RAM`, which is built up from `LUTs`. However a large number of `LUTs` is required to achieve a memory size comparable to one `BRAM`, and the resulting implementation suffers from restricted timing performance due to increased logic and routing complexity. On the other hand, sometimes is necessary to 585 implement small memories using `LUTs` due to the limited number of read/write ports of the `BRAMs` (the techinique is often implemented when either pipilining or unrolling is used, and it will be shown in the examples presented in this technical report).

`DSP48s` are meant for implementing high-speed arithmetic. They comprise a adder/subtractor, and multiplier. The unit can compute either functions like:

$$D = (A + B) \times C \tag{5}$$

or

$$D^{'} = D + A \tag{6}$$

It is also capable of SIMD operations, and complex floating point arithmetic can be performed combining multiple `DSP48s`. An in-depth description of all the functionalities of `DSP48` and `BRAM` can be found in *Xilinx 7 Series DSP48E1* 590 *Slice*[19], and *Xilinx 7 Series FPGAs Memory Resources*[20], respectively.

6.4. PS – PL interfaces

The appeal of `Zynq` SoC lies on the ability to use PS and PL in tandem to form an integrated system. The bridge between the two parts are the `Advanced eXtensible Interface (AXI) interfaces and connections`.

We briefly describe the connections and how they can be used.

[19]Xilinx 7 Series DSP48E1 Slice: `https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`.

[20]Xilinx 7 Series FPGAs Memory Resources: `https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`.

595 ### 6.4.1. *The standard* `AXI`

The `AXI` has become de facto a standard for ARM on-chip communication, and Xilinx contributed strongly to the definition of the current version `AXI4`.

There are three types of `AXI4`:

- `AXI4`: used for memory-mapped links. Burst-data transfer can be enable providing high-performance. Data can
600 be transfer in burst (up to 256 data words) after an address in supplied;
- `AXI4-Lite`: like `AXI4` but without the burst mode;
- `AXI4-Stream`: no-memory mapped, used for high-speed streaming data, supporting bust transfers of unrestricted size.

If a protocol is memory-mapped, in the case of `AXI4` bursts, the address for the first data word (read or write) is
605 issued by the master (PS), and the slave (PL) must then calculate the addresses for the data words that follow.

6.4.2. `AXI` *interface and interconnection*

First of all it is useful to briefly define the concept of interface and interconnection:

- `interface`: a point-to-point connection for transfer data between master and slave clients;
- `interconnection`: a switch that manages data movements between attached `AXI` interfaces.

Interface	**Description**	**Master**	**Slave**
M_AXI_GP0	general purpose	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	general purpose	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	cache coherent transaction	PL	PS
S_AXI_HP0	high performance ports with read/write FIFOs	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Table 2: Interfaces between PS and PL.

610 Table 2 summarizes the available interfaces, in accordance with convention that the master is in control of the bus, and initializes the transaction, and the slave responds.

The four S_AXI_HP[0-3] ports are high performance AXI interfaces that include FIFO buffers to allow burst read/write transactions, and support high rate communications between the PL and memory (DDR) in the PS. The data width is either 32 or 64 bits, and the PL acts as the master.

6.5. *The Zynq-7000 series*

The Zynq-7000 product comprises six different devices, with different features. The Table 3 lists the prominent features of such devices. Basically, the main difference between devices is the density of the resources embedded within the PL. Smaller devices are based on Xilinx Artix-7 FPGA logic fabric, and the larger ones on Xilinx Kintex-7 logic fabric. The ZedBoard is equipped with the Z-7020. The PS in standard across devices, the only difference is the maximum core frequency.

	Z-7010	**Z-7015**	**Z-7020**	**Z-7030**	**Z-7045**	**Z-7100**
Processor	Dual core ARM Cortex-A9 (32bit) with NEON and FPU					
Max CPU clock frequency	866 MHz			1 GHz		
PL	Artix-7			Kintex-7		
# FFs	35200	96400	106400	157200	437200	554800
# LUTs	17600	46200	53200	78600	218600	277400
# BRAM_18K	120	190	280	530	1090	1510
# DSP48	80	160	220	400	900	2020

Table 3: Zynq-7000 series. Regarding the PL, the table reports the number of FFs, LUTs, BRAM_18K, and DPS48 elements available.

7. Basic algorithm case studies

This Section contains examples and basic exercises. The main purpose is to provide guidance in learning the `Xilinx HLS` tool and the `OmpSs@FPGA` programming model. The source codes of the examples are available on GitLab (`https://www.ict.inaf.it/gitlab/david.goz/oats-technical-report-ompss_at_fpga.git`) under GPL3[21].

7.1. Loops

Loops are used extensively in software programming, and constitute a natural method of expressing operations that are repetitive in some way. Using `Vivado HLS` directives the software developer can change in a straightforward way how loops are converted in hardware.

During the reminder of the subsection, the default synthesis loop, and optimizations through the use of directives are discussed. In this first overview, very simple example codes are shown in order to clearly discuss the `HLS`-optimizations.

7.1.1. Default loop synthesis

```
darkenergy@EMPOKNOR:~$ docker ps -a
CONTAINER ID   IMAGE                COMMAND     CREATED        STATUS                    PORTS   NAMES
460428a3f109   ompss_at_fpga:2.1.0  "bash"      4 months ago   Up 3 hours                        sleepy_cannon
de6b2e8e8cdc   ompss_at_fpga:2.1.1  "bash"      4 months ago   Exited (0) 4 months ago           naughty_hertz
0634c0d74c3c   hello-world          "/hello"    4 months ago   Exited (0) 4 months ago           cool_napier
darkenergy@EMPOKNOR:~$
```

Figure 14: List of active docker containers.

```
darkenergy@EMPOKNOR:~$ docker ps -a
CONTAINER ID   IMAGE                COMMAND     CREATED        STATUS                     PORTS   NAMES
460428a3f109   ompss_at_fpga:2.1.0  "bash"      4 months ago   Exited (0) 7 seconds ago           sleepy_cannon
de6b2e8e8cdc   ompss_at_fpga:2.1.1  "bash"      4 months ago   Exited (0) 4 months ago            naughty_hertz
0634c0d74c3c   hello-world          "/hello"    4 months ago   Exited (0) 4 months ago            cool_napier
darkenergy@EMPOKNOR:~$ docker start -i -a sleepy_cannon

    - Welcome to the OmpSs@FPGA docker image  (version 2.1.0)
    - Mercurium and AIT (formerly autoVivado) tools are available in the PATH
    - Please, add vivado and vivado_hls in the PATH to make them available
      for ait

    - Please contact ompss-fpga-support@bsc.es for questions

ompss@EMPOKNOR:~$
```

Figure 15: Docker container start.

Listing 4: Example code of a loop adding the elements of two arrays. No `HLS`-pragmas are applied. The `Vivado HLS` synthesis estimate of resources is shown.

```
typedef int32_t MyData;
```

[21]GNU General Public License 3, `https://www.gnu.org/licenses/gpl-3.0.html`

```
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma  omp target device(fpga)
 #pragma  omp task in([BSIZE]in1, [BSIZE]in2) out([BSIZE]out)
void acc_vadd(const MyData *const restrict in1,
              const MyData *const restrict in2,
                    MyData *const restrict out)
{
 loop_vector_add:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
      out[i] = (in1[i] + in2[i]);
    }

  return;
}

/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/************************************************************/
/* DSP48E      0 used |    220 available -  0.0% utilization */
/* BRAM_18K   11 used |    280 available - 3.93% utilization */
/* LUT      8701 used |  53200 available -16.36% utilization */
/* FF       5565 used | 106400 available - 5.23% utilization */
/************************************************************/
```

We start from a simple code that adds the elements of two arrays. On the repository, the source code is under the *oats-technical-report-ompss_at_fpga/loop/default* folder:

1. **OmpSs@FPGA environment**: first of all, we need to start the Docker container:
   - run: `"docker ps -a"`. Although it may change the release of the package, the command output should be the list of active docker containers (the command might require `sudo` privileges, depending on how docker has been configured). An example output is shown in Figure 14;
   - look for the name of the container (multiple-version of `OmpSs@FPGA` might be installed). Run: `"docker start -i -a name_container"` (the command might require `sudo` privileges). An example output is shown in Figure 15;
   - to exit from the container it is enough running the command: `"exit"`.
2. **Cross-compiling**:
   - `Makefile` and example code (*vadd.c*) are prepared to compile and generate an OmpSs binary and the bitstream targeting the ZedBoard. The function to be ported on the FPGA is shown in 4;
   - in order to estimate the performance of the accelerator, before starting the time consuming process of generating the hardware, we will make the compilation stop after the `Vivado HLS` project is generated. At

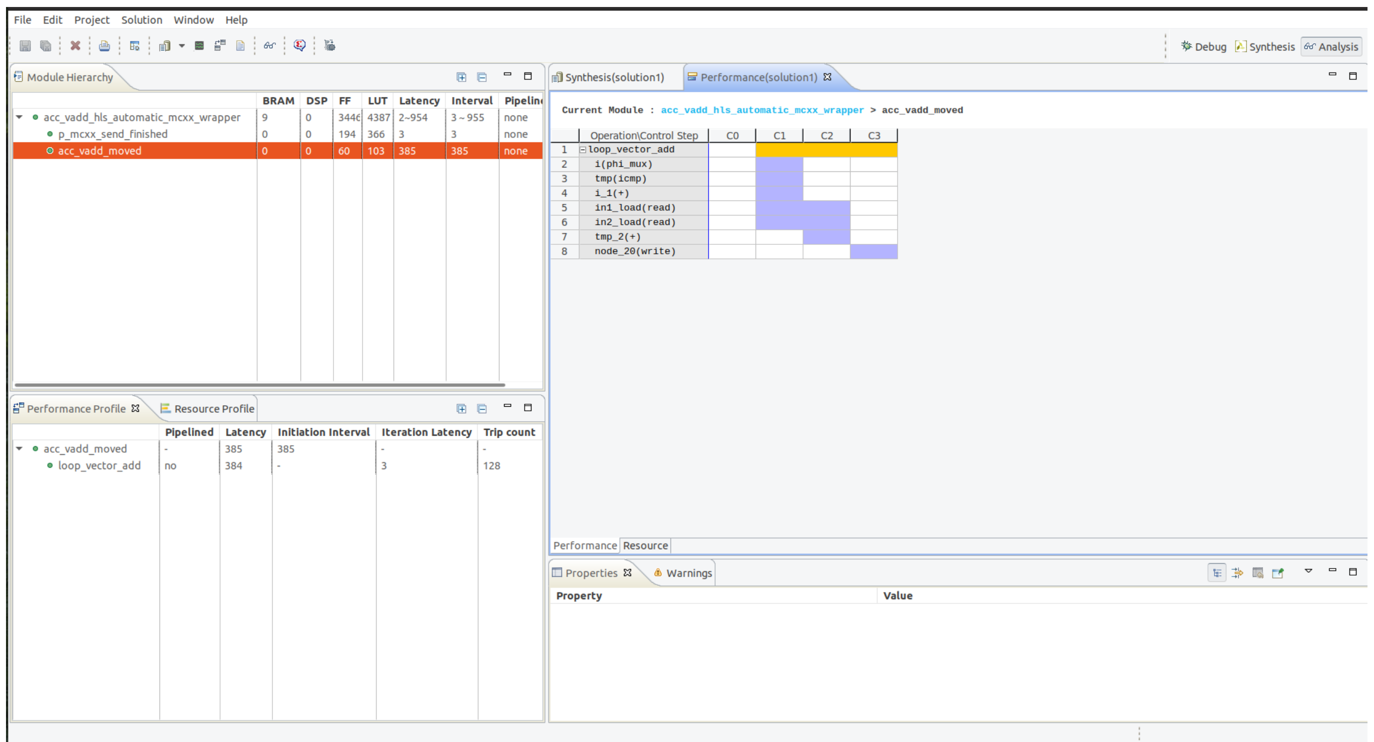

Figure 16: `Vivado HLS` Analysis Perspective.

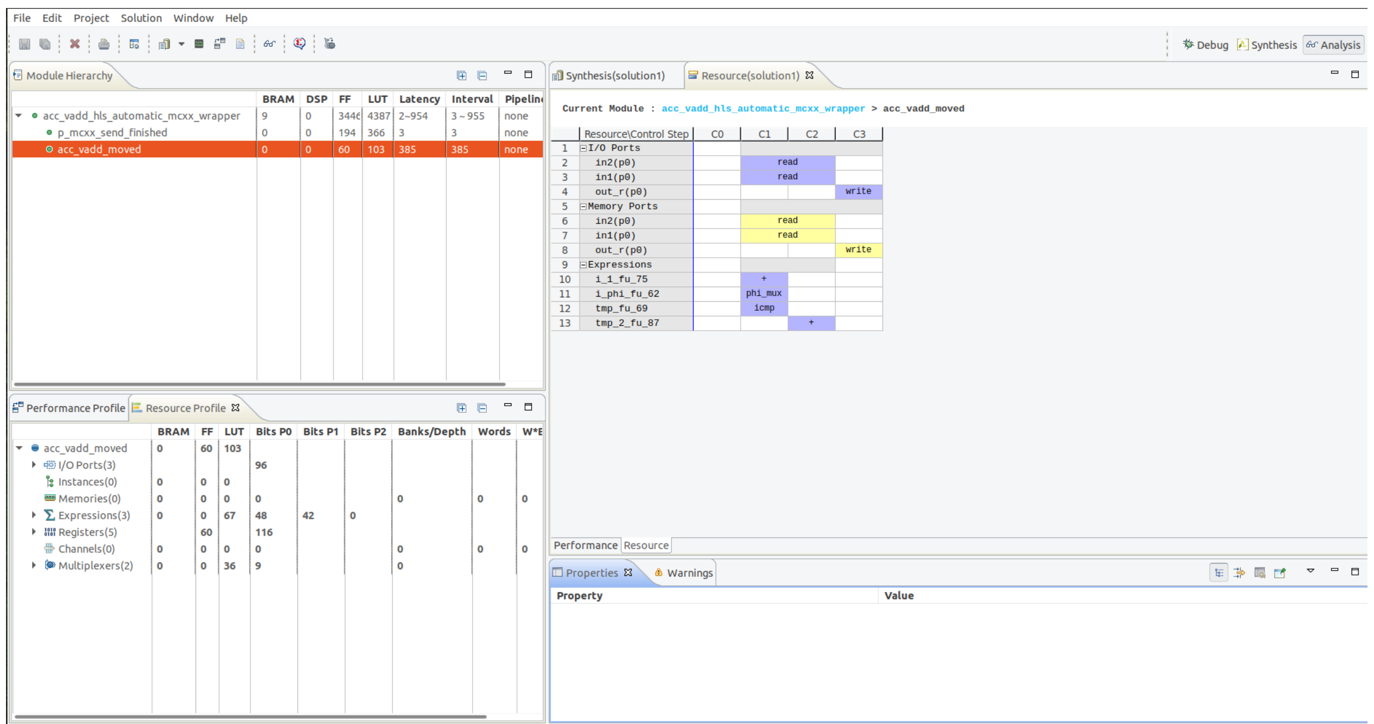

Figure 17: `Vivado HLS` Analysis Perspective with resource profile.

the moment we compile without hardware instrumentation, which will allow us to profile our application during execution.

675

Run: `"make HLS_p"`. The command issues the compilation with the flag `--to_step=HLS`, stopping the compilation after the `HLS` synthesis. The folder *zedboard_HLS_p/* is created and inside it the file *vadd_HLS_p.resources.txt* reports the resources estimate (`DSP48s`, `BRAMs`, `LUT`, `FF` are reported in 4). The `Vivado HLS` resource estimate shows that `DSP48s` are not utilized, because no floating-point arithmetic is used.

680 3. **Vivado HLS analysis**:
by default, `Vivado HLS` seeks to optimize area, so minimizing the resource to perform the loop. This means that the repetitive operations described by the loop (load and adder) are realised by a single piece of hardware implementing the body of the loop. The loop is designed to add the individual elements of two arrays. From the hardware point of view, one single adder is instantiated, shared BLOCK times according to the trip count.
685 The `Vivado HLS` project can be analyzed using the following command in the working directory:
`"vivado_hls -p zedboard_HLS_p/vadd_HLS_p_ait/xilinx/HLS/acc_vadd/"`
Once it is open, we can select the *Analysis* perspective on top-right corner. There, we can look for the accelerated function, called *acc_vadd_moved* by `OmpSs@FPGA`, to analyse the latency and the resource requirement, as show in Figure 16. From the Performance Profile (on bottom-left corner), we can see that the latency to perform the vector addition is 384 clock cycles (task latency): 3 cycles (iteration latency) for 128 iterations (without including reading the inputs from memory, and writing the output to memory). Hence, the total number of clock cycles to complete the execution of a loop, C_{loop}, would be:

$$C_{loop} = (N \times C_{body}) + C_{overhead} \tag{7}$$

where N is the number of iterations, C_{body} the number of clock cycles to execute all the operations in the loop body (i.e. iteration latency), and $C_{overhead}$ represents the overhead of transitions into and out the loop (e.g. memory transfers). We can see that pipeline is not applied (it must be explicitly define by the appropriate
690 `#pragma HLS`), and the initialization interval is not reported because is equal to the task latency.
Also from the Performance Profile, the label *loop_vector_add* of our code is shown, helping the programmer to analyze the code.
The *Performance* view (top-right) allows the programmer to interactively analyze the results in detail. The *Performance* view shows the following:

695
- the design starts in the $C0$ state;
- it then starts to execute the logic in *loop_vector_add*;
- the loop executes over 3 states: $C1$, $C2$, and $C3$; at $C1$ the design enters the loop, checks the loop increment counter and exit condition. The design then reads data into variable *in1_load* and *in2_load*, which require two clock cycles. Two clock cycles are required because the design is accessing a `BRAM`, requiring an address
700 in one cycle and a data read in the next. At $C2$ the design performs the sum and at $C3$ writes output;
- the loop return to the start.

In general, in the *Performance* view, the operations resulting from loops in the source code are colored yellow, standard operations are purple, and (possible) sub-blocks are green. Programmer can select an operation and right-click the mouse to open the associated operation in the source code view.

The Resource Profile pane (top-right) shows the resources used for every clock cycle. In our case (see Figure 17), two adders are required, one for the trip count and another for the vector add, so there is no sharing of the adders. More than one add operation on each horizontal line indicates the same resource is used multiple times in different states or clock cycles.

4. **Application execution on the FPGA**: it is necessary to completely compile the application to generate the bitstream, which is required to program the FPGA. The compilation may take a while, depending on the computation capability of the x86 development computer. It could be useful to use the `nohup` command to run the compilation so that it is possible to continue working while the application is compiling.

    - Run: `"nohup make bitstream_fpga_p >& \ output_bitstream_fpga_p.txt &"`. The x86 development computer starts the time consuming compilation;
    - run: `"tail -f output_bitstream_fpga_p.txt"` to observe the progress of the compilation;
    - transfer the following files from the x86 development computer to the `ZedBoard`[22]. This can be accomplished using the `scp` command:
        - `executable`: in our case the *vadd_fpga_p* file;
        - `bitstream (*.bin)`: this file should be under the folder */zedboard_fpga_p/vadd_fpga_p_ait*. It is the bitstream used to program the PL;
        - `configuration runtime (*.xtasks.config)`: this file should be under the folder */zedboard_fpga_p/vadd_fpga_p_* It is the configuration runtime file that is necessary when running the application. It contains the name of the accelerator, its working frequency, and the number of instances;
    - login into the `ZedBoard`. After that do the following steps:
        - run: `load_bitstream vadd_fpga_p.bin`, as shown in Figure 18. With this command the FPGA is programmed;
        - run the application through the executable. An example using the size vector of 1024 is shown in Figure 18. The application performs the calculation of the vector addition first on the CPU, then on the FPGA, and after that compares the results in order to check the correctness of the calculation on the FPGA. The application gets also time measurements, as shown in the Figure, so comparing the performance on the CPU (software execution time) and the FPGA (hardware execution time);
    - Shutdown the `ZedBoard`: correctly shut down the board with `sudo halt` of `sudo shutdown now`.

7.1.2. Pipelining loop

The loop synthesis of the previous code leads to a sequential execution in hardware, i.e. each iteration cannot start before the previous one has finished.

Listing 5: Example code of a loop adding the elements of two arrays. `HLS`-pipeline is applied. The `Vivado HLS` synthesis estimate of resources is shown.

[22] Section 5.1 shows in detail how to connect to the board.

```
ubuntu@zynq-bsc:~/technical_report/loop/default$ ll
total 3.9M
-rwxr-xr-x 1 ubuntu ubuntu  14K Aug 28 13:10 vadd_fpga_p
-rw-r--r-- 1 ubuntu ubuntu 3.9M Aug 28 13:10 vadd_fpga_p.bin
-rw-r--r-- 1 ubuntu ubuntu   80 Aug 28 13:10 vadd_fpga_p.xtasks.config
ubuntu@zynq-bsc:~/technical_report/loop/default$ load_bitstream vadd_fpga_p.bin
ubuntu@zynq-bsc:~/technical_report/loop/default$ ./vadd_fpga_p 1024

         TEST passed

         ================== RESULTS ==================
         Benchmark: vadd (OmpSs)
         Data type: int32_t
         Software execution time (secs): 3.61347e-05
         Hardware execution time (secs): 0.0025198
         =============================================

ubuntu@zynq-bsc:~/technical_report/loop/default$
```

Figure 18: ZedBoard application run.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma omp target device(fpga)
 #pragma omp task in([BSIZE]in1, [BSIZE]in2) out([BSIZE]out)
void acc_vadd(const MyData *const restrict in1,
              const MyData *const restrict in2,
                    MyData *const restrict out)
{
 loop_vector_add_pipeline:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
#     pragma HLS pipeline II=1

      out[i] = (in1[i] + in2[i]);
    }

  return;
}

/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/***********************************************************/
/* DSP48E      0 used |    220 available -  0.0% utilization */
/* BRAM_18K   11 used |    280 available - 3.93% utilization */
/* LUT      8729 used |  53200 available -16.41% utilization */
```

```
/* FF        5569 used | 106400 available - 5.23% utilization */
/*****************************************************************/
```

In order to improve the throughput of the design, a pipeline can be created inserting registers between each functional block, as described in Section 3.3 (on the repository the source code is under the *oats-technical-report-ompss_at_fpga/loop/pipelining* folder). The `PIPELINE pragma`, inserted in the loop body, reduces the initiation interval by allowing the concurrent execution of operations. The default initiation interval for the `PIPELINE pragma` is 1, which processes a new input every clock cycle. The software programmer can also specify the initiation interval through the use of the `II` option for the `pragma`. If `Vivado HLS` cannot create a design with the specified `II`, it:

- issues a warning (the programmer receives a hint to overcome the problem);
- create a design with the lowest possible `II`.

In general, the total number of clock cycles to complete the execution of the pipelined loop would be:

$$C_{loop} = C_{body} + [II \times (N - 1)] + C_{overhead} \tag{8}$$

where II is the initiation interval.

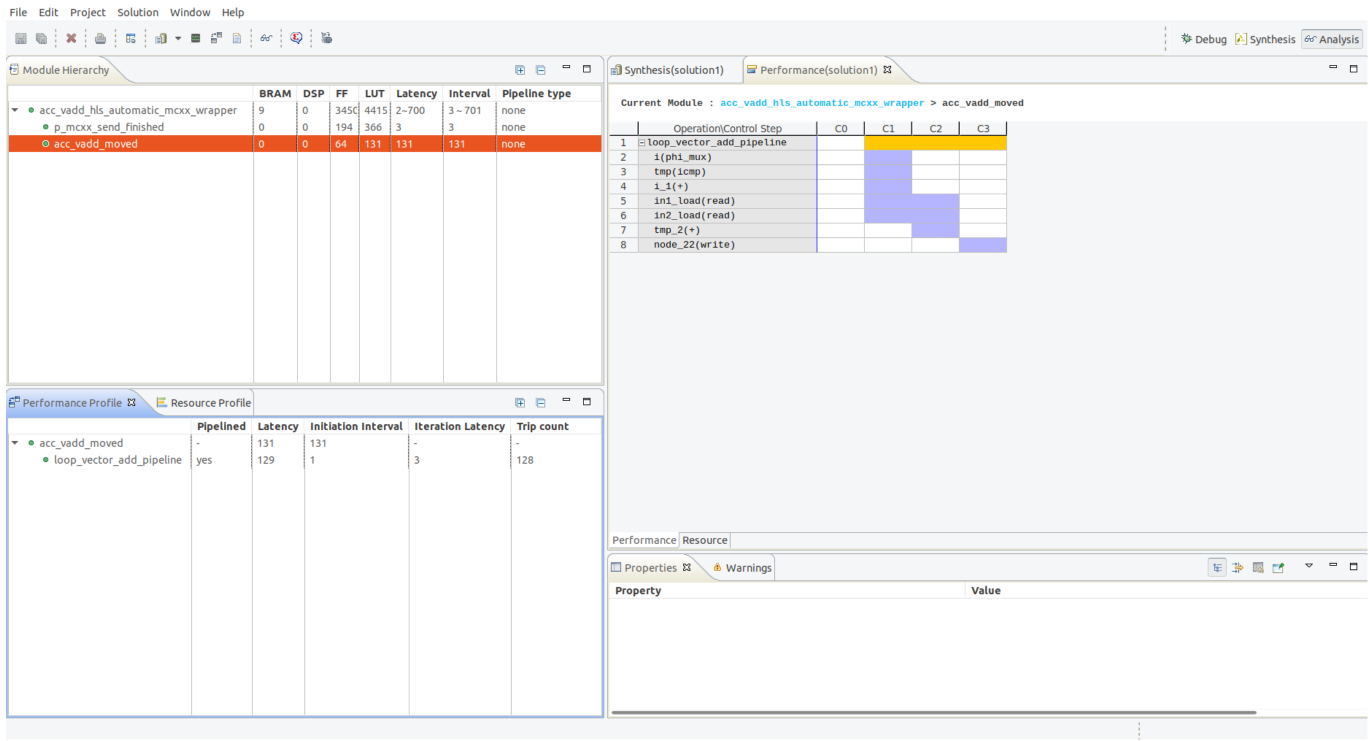

Figure 19: `Vivado HLS` Analysis Perspective when pipelining in applied.

In our example reported in the source code 5, `Vivado HLS` is able to create a pipelined loop with $II = 1$, so that the iteration latency is still 3 clock cycles, but the total task latency, with a modest increased resource usage of LUTs and FFs, is reduced to 130 clock cycles ($C_{loop} - C_{overhead}$), as shown in Figure 19. The effect of adding a pipelining directive can therefore be considerable, especially where there are multiple operations within the loop body, and many

iterations of the loop are performed. It is worth to be noticed that, in the case there is a loop inside a pipeline it needs to be completely unrolled, otherwise the pipeline cannot be created. This leads to an increased FPGA resource usage, and likely a resource contention.

7.1.3. Unrolling loop

Loop unrolling is another technique to exploit parallelism between loop iterations, as already discussed in Section 3.3.1 (on the repository the source code is in the *oats-technical-report-ompss_at_fpga/loop/unrolling* folder). It creates multiple copies of the loop body and adjust the loop iteration counter accordingly.

Listing 6: Example code of a loop adding the elements of two arrays. `HLS`-unroll by a factor of 2 is applied. The `Vivado HLS` synthesis estimate of resources is shown.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma omp target device(fpga)
 #pragma omp task in([BSIZE]in1, [BSIZE]in2) out([BSIZE]out)
void acc_vadd(const MyData *const restrict in1,
              const MyData *const restrict in2,
                    MyData *const restrict out)
{
 loop_vector_add_unroll:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
#     pragma HLS unroll factor=2

      out[i] = (in1[i] + in2[i]);
    }

  return;
}

/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/************************************************************/
/* DSP48E       0 used |    220 available - 0.0%  utilization */
/* BRAM_18K    14 used |    280 available - 5.0%  utilization */
/* LUT       8790 used |  53200 available -16.52% utilization */
/* FF        5603 used | 106400 available - 5.27% utilization */
/************************************************************/
```

So, in theory, unrolling a loop by a factor of F basically creates F copies of the loop body, and the task latency is expected to be reduced by a factor of F, i.e.:

$$C_{loop} = \left(C_{body} \times \frac{N}{F} \right) + C_{overhead} \tag{9}$$

In our example reported in the source code 3.3.1, we create a partial unroll by a factor of two. As expected the task latency is reduced to 192 clock cycles ($C_{loop} - C_{overhead}$).

The unrolling technique could lead one to believe that, if there are enough FPGA resources available, simply by putting "`#pragma HLS unroll factor=N`" it is possible to achieve a speedup by a factor of N.

TIMING (clock cycles)				**PL RESOURCES**			
Task latency	**Iteration latency**	**Trip count**	**Unroll factor**	**DSP48E**	**BRAM_18K**	**LUT**	**FF**
384	**3**	**128**	**0**	**0.0%**	**3.93%**	**16.36%**	**5.23%**
192	**3**	**64**	**2**	**0.0%**	**5.0%**	**16.52%**	**5.27%**
128	**4**	**32**	**4**	**0.0%**	**5.0%**	**16.76%**	**5.28%**
96	**6**	**16**	**8**	**0.0%**	**5.0%**	**17.01%**	**5.3%**
80	**10**	**8**	**16**	**0.0%**	**5.0%**	**17.44%**	**5.34%**
72	**18**	**4**	**32**	**0.0%**	**5.0%**	**18.4%**	**5.41%**
68	**34**	**2**	**64**	**0.0%**	**5.0%**	**20.09%**	**5.53%**
65	**65**	**1**	**128**	**0.0%**	**5.0%**	**20.25%**	**5.3%**

Table 4: A comparison of timing and PL resource usage using different unroll factors.

The Table 4 shows a comparison of the timing and the PL resource usage using different loop unrolling factors. The Equation 17 is no longer valid when the loop unrolling factor is larger than two, because the iteration latency (C_{body}) starts to increase. At the same time, the total amount of PL resources marginally increases.

As already discussed in Section 2, `BRAMs` offer high capacity, but access to data is limited to two different addresses each clock cycle. This is the reason why C_{body} begins increasing when the loop unrolling factor is four. On the contrary, `FF` based memories allow for multiple reads at different addresses in a single clock cycle, but their number is limited, even in the largest devices.

In our example (source code 6), two elements of the arrays $in1$, $in2$, and out, are accessed at a time (factor=2), saturating the maximum number of accesses of the `BRAMs` per clock cycle. So, using the `BRAMs`, the right equation should be:

$$C_{loop} = \left[C_{body} + \left(\frac{F-2}{2}\right)\right] \times \frac{N}{F} + C_{overhead} \tag{10}$$

where C_{body} is the (constant) iteration latency when no unrolling is applied (3 clock cycles in our example) and F is the unrolling factor (power of two).

From this example it is clear that accessing multiple array elements stored in `BRAMs` each clock cycle prevents high performance implementations. One possible solution is to divide arrays into smaller `BRAM` memories, a process called `array partitioning`. `Vivado HLS` performs some array partitioning automatically, but, in general, array

partitioning is a rather design-specific technique that it is necessary to guide the HLS tool for best results. Using the array_partition directive, arrays can be explicitly partitioned, resulting in a FF based memory.

The last part of this Section is devoted to present the most widely applied directive-based array-optimizations. Other Sections will provide enlightening examples where those techniques will be used.

7.2. Array-optimizations

In Vivado HLS arrays are usually synthesized into memories, which are mapped to physical resources on the PL. Often, in order to achieve a better performance, there is the need to exert influence over the array-mapping to physical resources, and the software developer can achieve this through the usage of HLS directive, as detailed below.

Array manipulation can therefore be considered a flexible and powerful technique available to the software developer, whether the default Vivado HLS implementation aims to minimize the resource utilization, often no maximizing performances.

7.2.1. #pragma HLS array_map

It combines multiple smaller arrays into a single large array to help reduce BRAM resources.

- *motivation*: each array is mapped into a block BRAM (or UltraRAM when supported by the device). If many small arrays do not use the full BRAM 18K, a better use of the BRAM resources is to map many small arrays into a single larger array;
- *syntax*: #pragma HLS array_map variable=<name> \
  instance=<instance> <mode> offset=<int>

  where:

  - variable=<name>: required argument that specifies the array variable to be mapped into the new target array <instance>;
  - instance=<instance>: specifies the name of the new array to merge;
  - <mode>: optionally specifies the array map as being either horizontal or vertical. Horizontal mapping is the default mode, and concatenates the arrays to form a new array with more elements. The vertical mapping creates a new array with longer words;
  - offset=<int>: it specifies an integer value offset to apply before mapping the array into the new array <instance>. It applies to horizontal type array mapping only.

7.2.2. #pragma HLS array_partition

It partitions an array into smaller arrays or individual elements.

- *motivation*: multiple small memories or multiple registers instead of one large memory are used, increasing the amount of read and write ports for the storage.

- *syntax*: #pragma HLS array_partition variable=<name> \
  <type> factor=<int> dim=<int>

  where:

  - variable=<name>: required argument that specifies the array variable to be partitioned;
  - <type>: it specifies the partition type. Three are supported:
    * complete: it decomposes the array into individual elements. This is the default type if <type> argument is not specified;
    * cyclic: the array is partitioned cyclically into the number of smaller arrays specified by the factor argument;
    * block: it creates smaller arrays from consecutive blocks of the original array. This splits the array into factor equal blocks.
  - factor=<int>: it specifies the number of smaller arrays that are to be created. For complete partitioning the factor argument is not necessary;
  - dim=<int>: it specifies the dimension to be partitioned in the case of a multi-dimensional array.

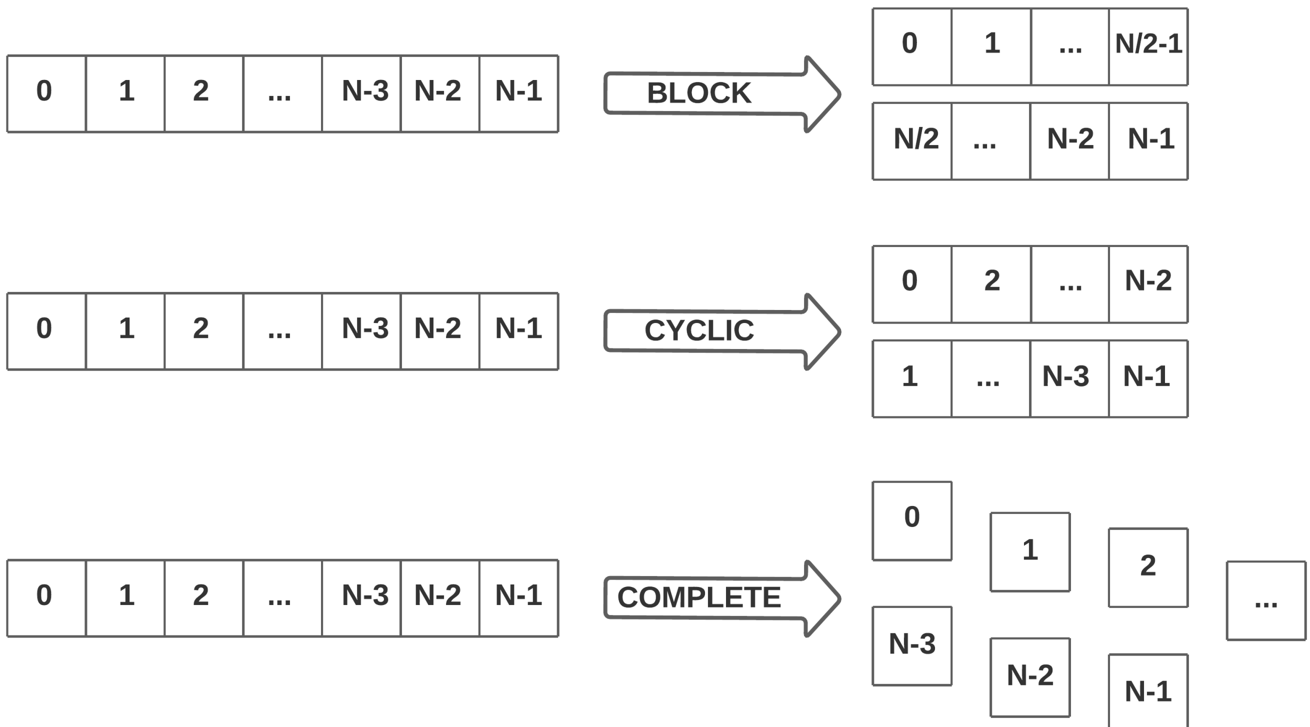

Figure 20: The #pragma HLS array_partition transforms the array in different ways according to the <type>. In this case factor = 2.

Figure 20 shows how the array partition works:

- block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into factor equal blocks;

- `cyclic` partitioning creates smaller arrays interleaving elements from the original array. If `factor=2`, as in Figure 20 , then the element 0 is assigned to the first new array, element 1 to the second new array, and the element 2 to the first new array;

- `complete` partitioning decomposes the original array into temporary individual elements.

7.2.3. #pragma HLS array_reshape

It combines array partitioning with vertical array mapping.

- *motivation*: this pragma combines the effect of the array_partition, breaking an array into smaller arrays, increasing the bit-width of each element. A new array is created with fewer elements but with greater bit-width. The purpose is to reduce the number of `BRAMs` consumed while providing the benefit of partitioning, i.e. the multiple accesses to the data;

- *syntax*: `#pragma HLS array_reshape variable=<name> \`
  `<type> factor=<int> dim=<int>`

  where:

  – `<name>`: required argument that specifies the array variable to be reshaped;

  – `<type>, factor=<int>, dim=<int>`: they are the same as for the array partition (see Section 7.2.2).

Figure 21 shows how the array reshape works:

- `block` reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into `factor` equal blocks, and then combines the blocks into a single array with elements with greater bit-width of `(word · factor)`, where `word` is the width of each original element;

- `cyclic` reshaping creates smaller arrays interleaving elements from the original array. If `factor=2`, as in Figure 21 , then the element 0 is assigned to the first new array, element 1 to the second new array, and the element 2 to the first new array. The final array is a vertical concatenation (word concatenation) of the new arrays into a single array;

- `complete` reshaping decomposes the original array into temporary individual elements and recombines them into a single element that contains all the elements of the original array (very-wide register).

7.2.4. #pragma HLS data_pack

It packs the data fields of a structure (C/C++ struct) into a single scalar with a wider word width.

- *motivation*: it is used for packing all the element of a structure in a single wide scalar with the benefit of reducing the memory required for the variable, and allowing all the members of the structure to be read/written from/to simultaneously. If the structure contains arrays, then this pragma performs a similar operation as the `array_reshape` pragma and combines the reshaped array with the other elements in the structure. It is worth to be noticed that `data_pack`, `array_partition`, and `array_reshape` pragmas are mutually exclusive.

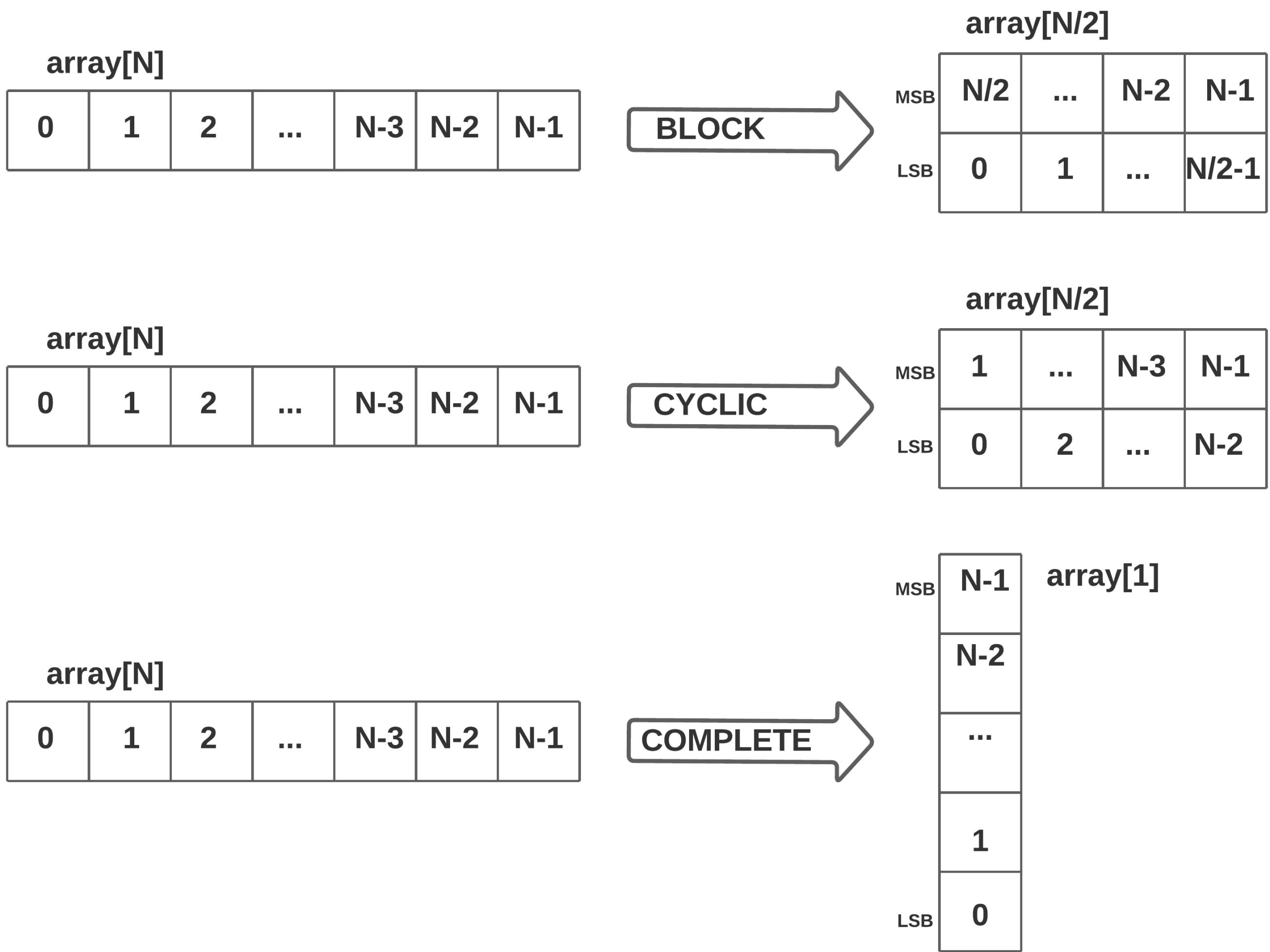

Figure 21: The `#pragma HLS array_reshape` transforms the array in different ways according to the `<type>`. In this case `factor = 2`.

910 • *syntax*: `#pragma HLS data_pack variable=<variable> \`
`instance=<name> <byte_pad>`

where:

– `variable=<variable>`: is the variable to be packed;

– `instance=<name>`: optional argument to specify the name of resultant variable after packing;

915 – `<byte_pad>`: optional argument that specifies whether to pack data on an 8-bit boundary (8-bit, 16-bit, 24-bit...). The two supported values for this option are:

∗ `struct_level`: it packs the whole structure first, then it pads it upward to the next 8-bit boundary;

∗ `field_level`: first it pads each individual element of the structure on an 8-bit boundary, then it packs the structure.

7.2.5. Vector addition using arrays partition

Coming back to the previous example of the simple code that adds the elements of two arrays, the contention of the concurrent accesses to the BRAMs can be fixed using the technique of the array partition when the loop is unrolled.

Listing 7: Example code of a loop adding the elements of two arrays. Complete unrolling is applied. Arrays are completely partitioned.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma  omp target device(fpga)
 #pragma  omp task in([BSIZE]in1, [BSIZE]in2) out([BSIZE]out)
void acc_vadd(const MyData *const restrict in1,
              const MyData *const restrict in2,
                    MyData *const restrict out)
{
 #pragma  HLS array_partition variable=in1 complete
 #pragma  HLS array_partition variable=in2 complete
 #pragma  HLS array_partition variable=out complete

 loop_vector_add_unroll:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
#     pragma HLS unroll factor=BSIZE

      out[i] = (in1[i] + in2[i]);
    }

  return;
}

/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/***********************************************************/
/* DSP48E       0 used |    220 available - 0.0%  utilization */
/* BRAM_18K    8  used |    280 available - 2.86% utilization */
/* LUT      15125 used |  53200 available -28.43% utilization */
/* FF       17696 used | 106400 available -16.63% utilization */
/***********************************************************/
```

`Vivado HLS` in the source code in Figure 7 completely unrolls the loop and decomposes the arrays into individual registers. In this way all the elements can be accessed in the same clock cycle. Compared to the full loop-unroll resource usage of Table 4 (last row), the arrays partition requires fewer BRAMs but a larger amount of FFs. Combining a full loop-unrolling and a complete array partition, `Vivado HLS` is able to create a design with a task latency of one clock cycle, a 384X speed-up compared to the default implementation.

In general, if `Vivado HLS` fails to satisfy an optimization directive, it automatically relaxes the optimization and

seeks to create a design with a lower performance target. If it cannot relax the target, it will halt with an error. In any case `Vivado HLS` provides a list of reasons why it was not able to satisfy the higher performance target. However, it is a good rule to always inspect carefully the HLS Analysis Synthesis estimates to check if the optimization directives have been fully applied.

7.3. `Vivado HLS` coding examples

The Vivado Design Suite installation provides examples of various coding techniques. Coding examples are usually in *Vivado/20XX.X/examples/coding* folder.

8. Algorithm case studies

This section outlines the various optimizations and techniques you can use to direct `Vivado HLS` to produce a micro-architecture that satisfies the desired performance and area goals. The source codes of the examples are available on GitLab (`https://www.ict.inaf.it/gitlab/david.goz/oats-technical-report-ompss_at_fpga.git`).

8.1. Running total

A running total is the summation of a sequence of numbers which is updated each time a new number is added to the sequence, by adding the value of the new number to the previous partial sum. So, if *input* is a input array of values of a given size, then the elements of the *output* array that contains the running total is evaluated as:

$$\begin{cases} output_0 = input_0 \\ output_i = output_{i-1} + input_i, \quad i = 1, ..., size - 1. \end{cases}$$

While the algorithm is very simple to implement, its FPGA optimization is possible applying the arguments discussed in the previous Section 7.

Listing 8: Example code of the naive running total algorithm. Arrays are partitioned in block.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma omp target device(fpga)
 #pragma omp task in([BSIZE]in) out([BSIZE]out)
void accumulation(const MyData  *const restrict in,
                        MyData  *const restrict out,
                  const u_int32_t               el)

{
 #pragma HLS inline off

   static MyData sum;

```

```
  out[0] = ((el == 0) ? in[0] : (sum + in[0]));

 loop_acc:
  for (u_int16_t i=1 ; i<BLOCK ; i++)
    {
      out[i] = (out[i - 1] + in[i]);
    }

  sum = out[BLOCK - 1];

  return;
}

/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                  */
/***********************************************************/
/* DSP48E      0 used |    220 available -  0.0% utilization */
/* BRAM_18K    8 used |    280 available - 2.86% utilization */
/* LUT      8107 used |  53200 available -15.24% utilization */
/* FF       5115 used | 106400 available - 4.81% utilization */
/***********************************************************/
```

The code listed in 8 is the naive FPGA implementation of the running total algorithm. The input and output arrays (the first contains the values and the second the partial summation) are divided in chunks of size $BLOCK$ (fixed to 128), so the accelerator is called $size/BLOCK$ times, where $size$ is the number of elements of the input array. Each time the accelerator is called, the first element of the output chunk must be initialized before starting to evaluate the partial summation; if it is the first chunk of data, then the first element of the output is simply initialized as $out[0] = in[0]$, otherwise a static variable (its value is preserved between function calls) is used to store the last element of the output block.

Without applying `HLS` optimizations, the resources usage is reported in the Listing 8, and the analysis perspective in Figure 22. Every time the accelerator is called, three clock cycles are needed to set the first element of the output block. The loop (labelled as *loop_acc*) is the most time consuming task, with an iteration latency of three clock cycles, and a total latency of 381 clock cycles (the trip count is 127).

The loop pipelining is the first optimization that we try in order to increase the throughput of the design. The Figure 23 shows the analysis perspective adding the `#pragma HLS pipeline II=1` to the loop labelled as *loop_acc*. Two operations are shown in red. These are conflicts (`out_load_1(read)` and `node_34(write)` operations), which have a dependency between loop iterations, and indeed the achievable initiation interval is two clock cycles instead of one. During the scheduling process, `Vivado HLS` issues a warning, as shown in Figure 24, saying that it is unable to enforce a carried dependence constraint (II=1) between store and load operations on the same array (the `out` array in the source code 8). With a double-click on any of these red operations and the Critical Pipeline Dependence Information windows opens, showing two iterations of the loop over multiple clock cycles. The first iteration of the loop shows the states in which the operations occur. The read in states 4 and 5, and the write in state 5. The operation in the

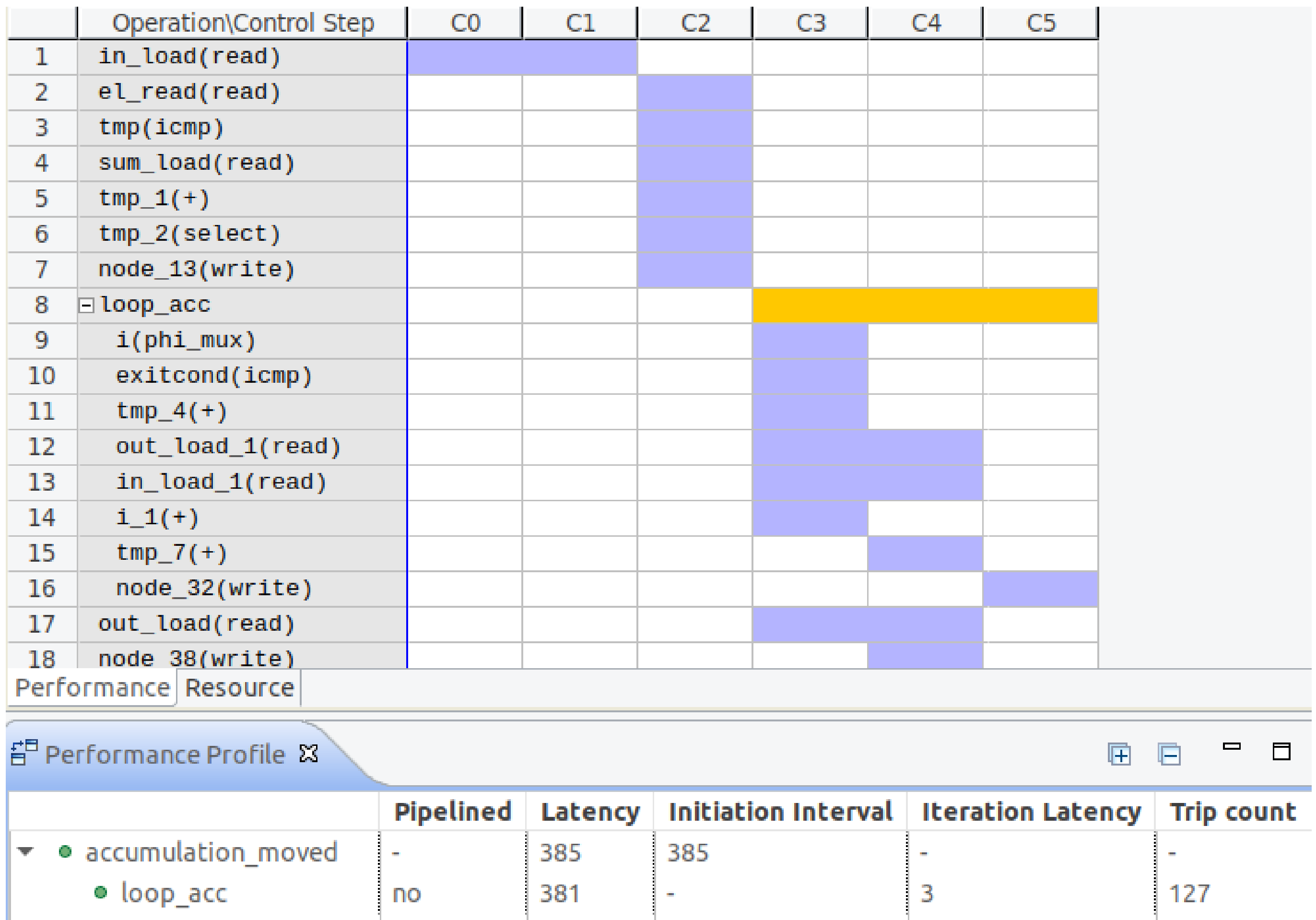

Figure 22: `Vivado HLS` Analysis Perspective of the naive running total algorithm listed in the Listing 8.

next iteration must start 1 cycle after this, because the second read cannot occur until the first write has finished: the operations in each iteration of the loop are to a different address and only one address can be applied at the same time.

The target II=1 can be achieved by splitting the `out` array into multiple smaller arrays, allowing the load of one elements of the *out* array in one clock cycle. Table 5 summarizes timing and PL resource usage using different `#pragma HLS`.

In our example, loop-carry dependence, i.e. the same element is accessed in different loop iteration, is obvious. In complex algorithms automatic dependence analysis can be too conservative and fail to ruling out false dependencies. The `DEPENDENCE` HLS pragma[23] allows the programmer to explicitly specify the dependence and resolve a false dependence. Specifying a false dependency, when in fact the dependency is true, can result in incorrect hardware. Indeed, in our code if inside the loop the `"#pragma HLS dependence variable=out inter false"` is specified, then the resulting hardware is incorrect and the check test fails.

[23]HLS pragma reference: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1253-sdx-pragma-reference.pdf`

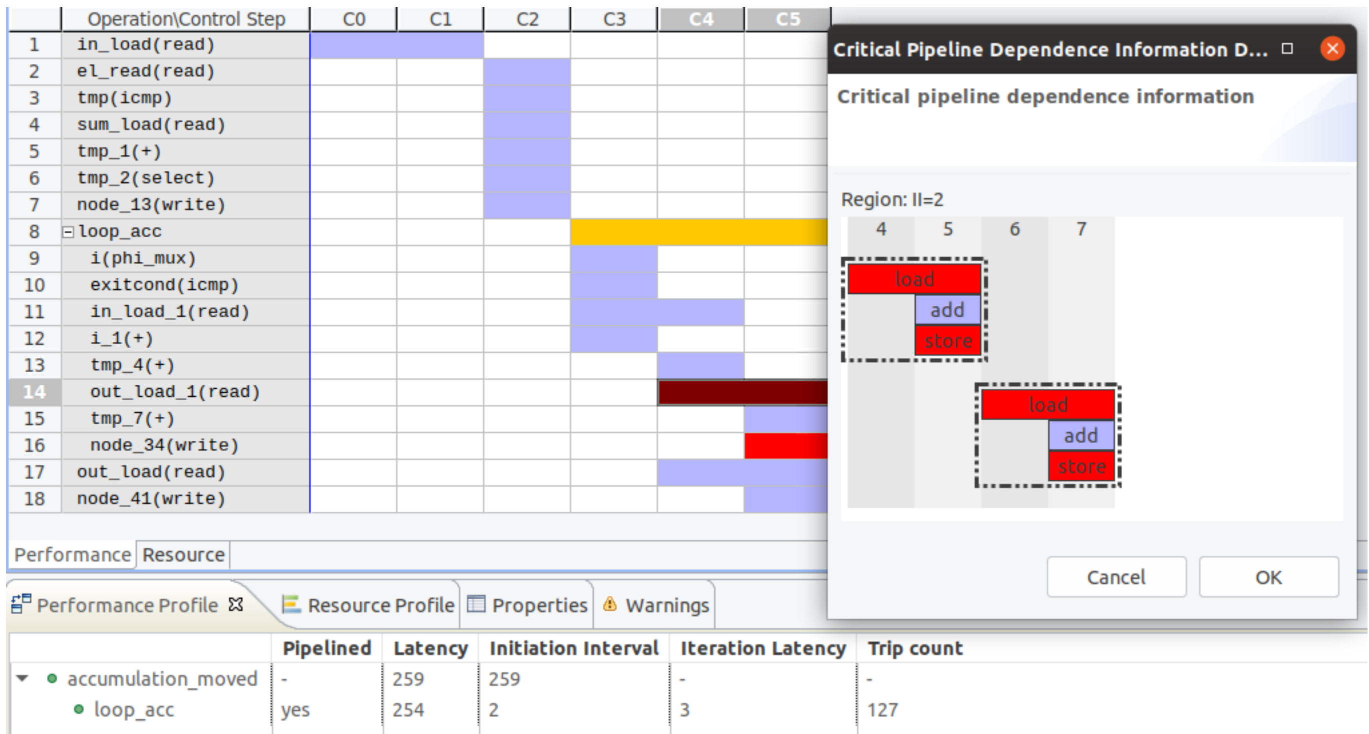

Figure 23: `Vivado HLS` Analysis Perspective of the pipeline version of the running total algorithm listed in the Listing 8. Red indicates conflicts. Clicking the red cycles it is possible to see where the conflicts are.

```
INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'loop_acc'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependence constraint (II = 1, distance =
1, offset = 1)
   between 'store' operation (accumulation/accumulation_hls_automatic_mcxx.cpp:39) of variable
 'tmp_7', accumulation/accumulation_hls_automatic_mcxx.cpp:39 on array 'out_r' and 'load' oper
ation ('out_load_1', accumulation/accumulation_hls_automatic_mcxx.cpp:39) on array 'out_r'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 2, Depth = 3.
```

Figure 24: `Vivado HLS` warnings.

The careful reader may notice that there is a relatively easy way to rewrite the code avoiding the loop-carry dependence. The accumulation can be performed on a separate scalar variable (using only a register that stores the accumulated value), rather than reading the previous value back from the array, so avoiding one memory access to the array, as shown in the code listed in 9. This trick allows to achieve a pipeline with II=1 without using array partitioning and saving FPGA resources.

Listing 9: Code for implementing an optimized running total algorithm. Arrays are partitioned in block.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma omp target device(fpga)
```

TIMING (clock cycle)					HSL pragmas	PL RESOURCES			
Function latency	Loop latency	Iteration latency	Initiation interval	Trip count	Optimization	DSP48E	BRAM_18K	LUT	FF
385	381	3	-	127	none	0.0%	2.86%	15.24%	4.81%
259	254	3	2	127	- pipeline II=1	0.0.%	2.86%	15.31%	4.79%
131	127	2	1	127	- pipeline II=1 - out array_partition cyclic factor=32	0.0%	2.5%	21.61%	8.58%
131	127	2	1	127	- pipeline II=1 - out array_partition block factor=32	0.0%	2.5%	21.6%	8.58%
131	127	3	1	127	- pipeline II=1 - out array_partition complete	0.0%	2.5%	27.2%	12.5%
129	127	2	1	127	- pipeline II=1 - out array_partition complete - in array_partition complete	0.0%	2.14%	27.08%	16.27%
65	-	-	-	-	- unroll	0.0%	3.57%	28.11%	6.73%
42	-	-	-	-	- unroll - out array_partition complete - in array_partition complete	0.0%	2.14%	29.73%	23.87%

Table 5: A comparison of timing and PL resource usage using different `#pragma HLS` applied to the *running total* algorithm.

```
#pragma omp task in([BSIZE]in) out([BSIZE]out)
void accumulation(const MyData    *const restrict in,
                        MyData    *const restrict out,
                  const u_int32_t                 el)

{
#pragma HLS inline off

  static MyData sum;

  MyData A = ((el == 0) ? in[0] : (sum + in[0]));

  out[0] = A;

 loop_acc:
  for (u_int16_t i=1 ; i<BLOCK ; i++)
    {
```

```
#    pragma HLS pipeline II=1

       A += in[i];
       out[i] = A;
     }

  sum = out[BLOCK - 1];

  return;
}

/********************* VIVADO HLS REPORT ******************/
/* Target Board: ZedBoard                                  */
/*********************************************************/
/* DSP48E       0 used |    220 available -  0.0% utilization */
/* BRAM_18K     8 used |    280 available - 2.86% utilization */
/* LUT       8107 used |  53200 available -15.29% utilization */
/* FF        5115 used | 106400 available - 4.84% utilization */
/*********************************************************/
```

This example shows that the HLS compiler is not able to automatically optimize the memory loads and stores of a single basic block. However, if the unroll of the original loop (code listed in 8) is performed, HLS is able to eliminate most of the read operations of the out array within the body loop, but still not achieving the same performance as when array partition is used.

8.2. Matrix-Vector multiplication

The matrix-vector multiplication is the core of many algorithms, e.g. the Discrete Fourier Transform. The source codes of the examples are available on GitLab (https://www.ict.inaf.it/gitlab/david.goz/oats-technical-report-omps at_fpga.git).

Listing 10: Example code of the matrix-vector algorithm. Arrays are partitioned in chunks. The source code in the repository is under the folder *matrix_vector*.

```
#define BLOCK 32
const unsigned int BSIZE = BLOCK;

#if defined(_SMP_)
 #pragma omp target device(smp)
#else
 #pragma omp target device(fpga)
#endif
 #pragma omp task in ([BSIZE]matrix, [BSIZE]vector) inout([1]out)
void acc_matrix_vector_mul(const MyData *const restrict matrix,
                           const MyData *const restrict vector,
                                 MyData *const restrict out)
{
```

```
# pragma HLS inline off

  MyData local_out = (MyData)0.0;

 loop_matrix_vector_product:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
      local_out += (matrix[i] * vector[i]);
    }

  out[0] += local_out;

  return;
}

void hw_matrix_vector_mul(const MyData *const restrict matrix,
                          const MyData *const restrict vector,
                                MyData *const restrict out,
                          const u_int32_t              size)
{
  const u_int32_t s2ize = (size * size);

  for (u_int32_t el=0 ; el<s2ize ; el+=BLOCK)
    {
      const u_int32_t vec_index = (el % size);
      const u_int32_t out_index = (el / size);

      acc_matrix_vector_mul(&matrix[el], &vector[vec_index], &out[out_index]);

#    pragma omp taskwait
    }

  return;
}
```

The basic code is shown in the listing 10. We use a custom data type called `MyData`, that through the discussion is mapped into `int32`, `float`, and `double`. The *hw_matrix_vector_mul* running on the host has three arguments. The first two arguments `MyData *matrix` and `MyData *vector` are the input pointers to the matrix and the vector. If `size` is the dimension of the output vector, then for the matrix is allocated a buffer of (`size * size * sizeof(MyData)`) bytes, so the `matrix(i,j)` element is mapped as `matrix[(i * size) + j]`. The host function *hw_matrix_vector_mul* splits the calculation in chunks of size `BLOCK` (whose value is set at 32), with the constraint (`size % BLOCK == 0`). The function *acc_matrix_vector_mul* is the FPGA target kernel. It performs the scalar product between the rows of the matrix and the vector in chunks. The loop labelled as *loop_matrix_vector_product* is the core loop, with its ranges hard coded, allowing us to apply HLS optimizations (unrolling is not possible if loop ranges are not specified at compile time) and/or obtain HLS estimates.

	Resource\Control Step	C0	C1	C2	C3	C4
1	⊟I/O Ports					
2	out_0_read	read				
3	vector(p0)		read			
4	matrix(p0)		read			
5	ap_return					
6	⊟Memory Ports					
7	vector(p0)		read			
8	matrix(p0)		read			
9	⊟Expressions					
10	local_out_phi_fu_60		phi_mux			
11	i_phi_fu_72		phi_mux			
12	i_1_fu_85		+			
13	out_fu_97		+			
14	tmp_fu_79		icmp			
15	tmp_3_fu_102				*	
16	local_out_1_fu_106					+

Figure 25: `Vivado HLS` resource estimates for default implementation of the matrix-vector algorithm with integer (`int32`) data type. Source code is listed in Figure 10.

	Operation\Control Step	C0	C1	C2	C3	C4
1	out_0_read_1(read)					
2	⊟loop_matrix_vector_product					
3	local_out(phi_mux)					
4	i(phi_mux)					
5	tmp(icmp)					
6	i_1(+)					
7	matrix_load(read)					
8	vector_load(read)					
9	tmp_3(*)					
10	local_out_1(+)					
11	out(+)					

Performance | Resource

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
acc_matrix_vector_mu_1	-	129	129	-	-
loop_matrix_vector_product	no	128	-	4	32

Figure 26: `Vivado HLS` performance estimates for default implementation of the matrix-vector algorithm with integer (`int32`) data type. Source code is listed in Figure 10.

Figure 25 shows a sequential (hereafter default, i.e. without HLS directives) architecture for the matrix-vector multiplication with one multiply and one addition operation instantiated (*tmp_3_fu_102* and *local_out_1_fu_106* in 1160 Figure 25 at $C3$ and $C4$, respectively). Logic is created to access the matrix and the input vector that are stored in `BRAMs`. It does not consume a lot of area, as expected, but the task latency and task interval are relatively large, as shown in Figure 26. Using 32-bit integer data type, the read requires two clock cycles (as already seen in previous examples), while multiply and adder (accumulation) operations are performed in one clock cycle (so the iteration latency is four clock cycles).

Basically, if each read operation takes C_R clock cycles (one element of either the matrix or the vector), each adder operation takes C_A clock cycles, and each multiply operation takes C_M clock cycles, then the loop latency, C_L^d (default loop), is:

$$C_L^d = BLOCK \cdot (C_R + C_M + C_A) = BLOCK \cdot C_I^d \tag{11}$$

1165 where $C_I^d = (C_R + C_M + C_A)$ is the (default) iteration latency.

The expected task latency (i.e. a single call to the synthesized kernel), C_T^d, is:

$$C_T^d = C_L^d + C_{overhead} \tag{12}$$

where $C_{overhead}$ is the clock cycles required to update (read and write) the `out` variable. From the synthesis performance analysis shown in Figure 26 this operation requires one clock cycle.

The kernel is called $S^2/BLOCK$ times, where S is the size of the output vector. Finally the total matrix-vector (default) multiplication latency, C_{M-V}^d, is:

$$C_{M-V}^d = C_T^d \cdot \left(\frac{S^2}{BLOCK}\right) \tag{13}$$

The minimum latency is achieved when $BLOCK = S$.

It is worth to be noticed that the latency of moving data from/to the main host memory cannot be inferred by 1170 `Vivado HLS`, because it depends on the interaction between the PL and the PS.

8.2.1. Pipelining

The matrix-vector algorithm has substantial opportunity to exploit parallelism. In the default implementation the expression `local_out += (matrix[i] * vector[i])` is executed in each iteration of the loop. The variable `local_out` is being reused in each iteration and takes a new value, since it is an accumulator. Looking at Figure 26, it is apparent that while the hardware is performing the multiply and add operations on the i-th elements, the read operations can start on the next elements. Through the `#pragma HLS pipeline II=1` it is easy to achieve a loop pipelined behaviour with the initiation interval of one clock cycle, requiring few more FPGA resources. The pipeling does not affect the iteration latency (it is still four clock cycles), but significantly reduces the loop latency, from Eq 11 to:

$$C_L^p = C_I^d + II(BLOCK - 1) \tag{14}$$

Hence, using the pipelining the achievable speedup can be estimated as:

$$speedup = \frac{C_{M-V}^d}{C_{M-V}^p} = \frac{\left(C_I^d \cdot BLOCK\right) + C_{overhead}}{C_I^d + II\,(BLOCK - 1) + C_{overhead}} \tag{15}$$

In our case $C_I^d = 4$, $C_{overhead} = 1$, $II = 1$, $BLOCK = 32$, so $speedup \simeq 3.58$.

8.2.2. Unrolling

We always focus on the loop where the scalar product is performed. The maximum amount of parallelism can be achieved unrolling the loop so that multiplications can be performed simultaneously, and the summations can be performed using an adder tree.

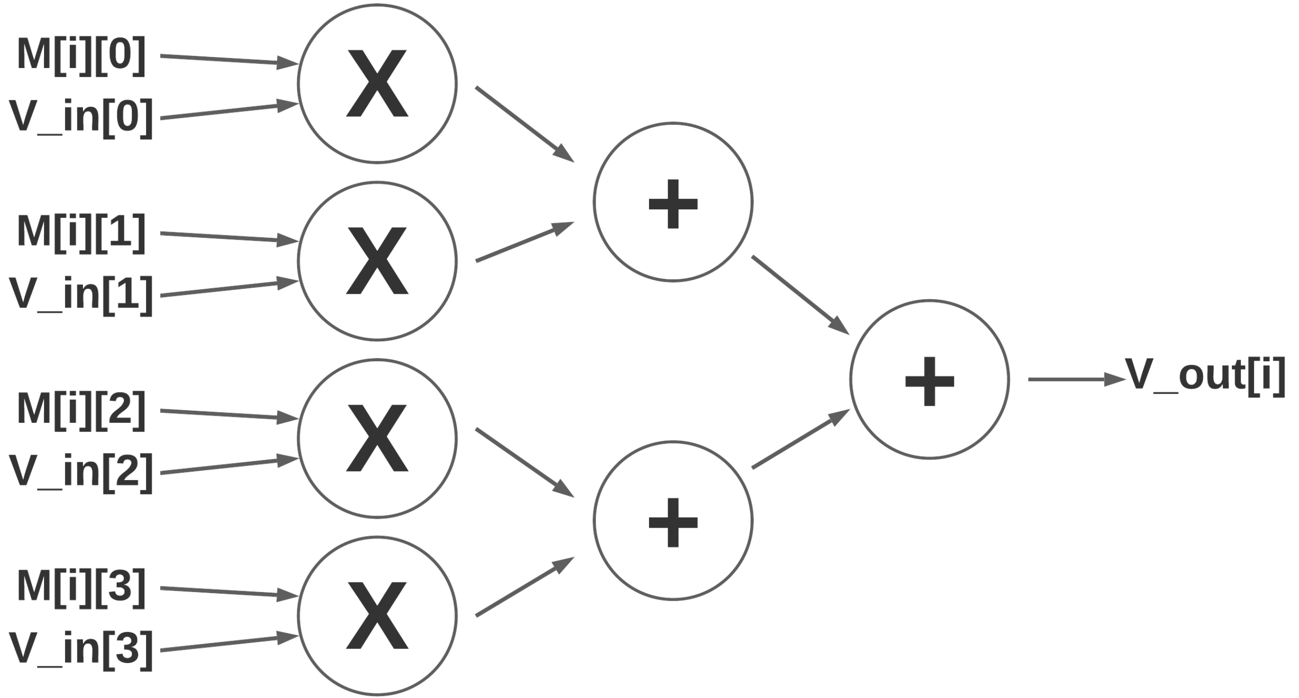

Figure 27: A data flow graph of the expression resulting from the unrolled loop performing the matrix-vector scalar product.

A sketch of the data flow of the computation (assuming $BLOCK = 4$) is shown in Figure 27. Assuming a complete unrolling, all the reads and multiplications should be executed in parallel, while the additions with a tree flow. In this scenario, $BLOCK$ multipliers and $BLOCK/2$ adders must be instantiated in the PL. Hence, the resulting loop latency is:

$$C_L^u = C_R^d + C_M^d + \left(C_A^d \cdot \log_2(BLOCK)\right) \tag{16}$$

and $C_T^u = (C_L^u + C_{overhead})$ is the task latency, accordingly. The achievable speedup using the unrolling can be estimated as:

$$\frac{C_{M-V}^d}{C_{M-V}^u} = \frac{BLOCK\left(C_R^d + C_M^d + C_A^d\right) + C_{overhead}}{C_R^d + C_M^d + \left(C_A^d \cdot \log_2(BLOCK)\right) + C_{overhead}} \tag{17}$$

Using $C_R^d = 2$, $C_M^d = C_A^d = C_{overhead} = 1$, and $BLOCK = 32$, then $speedup \simeq 14$.

However, applying full unrolling to the loop, the `Vivado HLS` performance analysis, shown in Figure 28, reveals a different behaviour, i.e. the Eq. 17 is not valid. At most two elements of both the matrix and the vector are read concurrently (the read requires always two clock cycles), so all the multiplications cannot be performed in the same clock cycle. Indeed, only two multipliers are instantiated by `Vivado HLS`, as shown in Figure 29, and are shared through the kernel execution. Array partitioning is necessary in order to fully exploit the unrolling, because the `BRAMs`

	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6
1	matrix_load(read)							
2	vector_load(read)							
3	matrix_load_1(read)							
4	vector_load_1(read)							
5	matrix_load_2(read)							
6	vector_load_2(read)							
7	matrix_load_3(read)							
8	vector_load_3(read)							
9	tmp_3(*)							
10	tmp_3_1(*)							
11	matrix_load_4(read)							
12	vector_load_4(read)							
13	matrix_load_5(read)							
14	vector_load_5(read)							
15	tmp_3_2(*)							
16	tmp_3_3(*)							
17	matrix_load_6(read)							
18	vector_load_6(read)							
19	matrix_load_7(read)							
20	vector_load_7(read)							
21	tmp3(+)							
22	tmp_3_4(*)							
23	tmp_3_5(*)							
24	matrix_load_8(read)							
25	vector_load_8(read)							
26	matrix_load_9(read)							
27	vector_load_9(read)							
28	tmp4(+)							
29	tmp_3_6(*)							

Figure 28: HLS performance analysis when `#pragma HLS unroll` is applied.

allow at most two concurrent accesses. Applying complete array partitioning, so moving data from `BRAMs` to `FFs`, allows multiple reads at different addresses in a single clock cycle. It is also possible to read, modify, and write a flip-flop based memory in a single clock cycle, instead `BRAMs` read operation must have a latency of at least one clock cycle. In our case, the size of data to be stored in `FFs` is quite small (i.e. $BLOCK = 32$), so it is a good choice to apply this technique to obtain high performance implementation. As with many other directive-based optimizations, the same result can also be achieved by rewriting the code manually. In general, it is preferred to use the tool directives since it keeps the code easy to maintain and it avoids introducing bugs. Using complete array partitioning, `Vivado HLS` performance analysis shows that the kernel latency is now of three clock cycles, i.e. the scalar product and the output are performed in only three clock cycles, even better than the predicted speedup of Eq. 17, because reads, multiplies, and adders are overlapped. Hence, in summary, with only unrolling the kernel achieves a speedup (respect

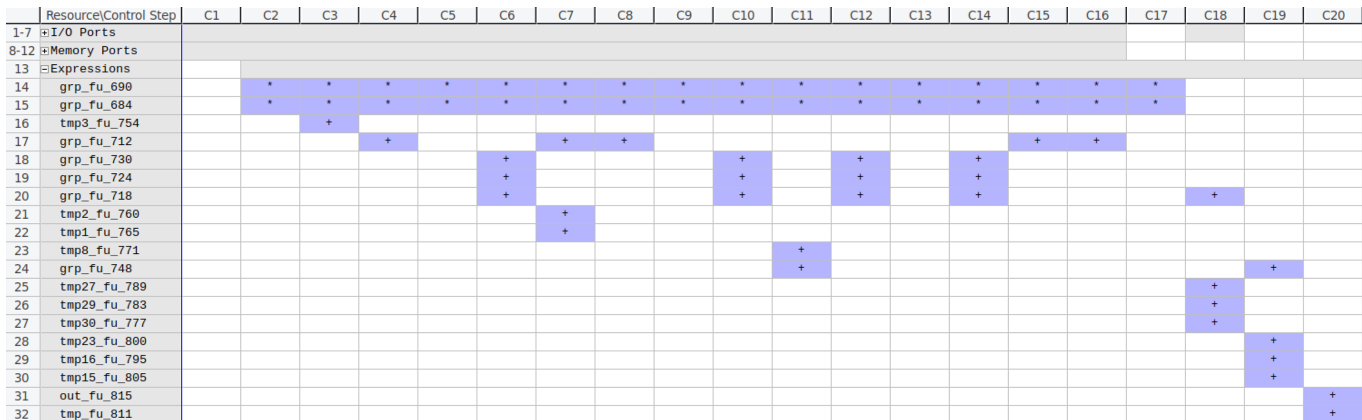

Figure 29: HLS resource analysis when `#pragma HLS unroll` is applied.

the default implementation) of $\simeq 6.5x$, adding the array partitioning it increases up to $\simeq 43x$, while the predicted speedup of Eq. 17 is $\simeq 14x$.

Table 6 shows a comparison of timing and PL resource usage using different `#pragma HLS` directives applied to the matrix-vector algorithm.

Data Type : int32									
TIMING (clock cycles)					**HSL pragmas**	**PL RESOURCES**			
Function latency	**Loop latency**	**Iteration latency**	**Initiation interval**	**Trip count**	**Optimization**	**DSP48E**	**BRAM_18K**	**LUT**	**FF**
129	**128**	**4**	**-**	**32**	**none**	**1.36%**	**3.57%**	**16.05%**	**5.29%**
36	**34**	**4**	**1**	**32**	**- pipeline II=1**	**1.36%**	**3.57%**	**16.13%**	**5.32%**
20	**-**	**-**	**-**	**-**	**- unroll**	**2.73%**	**4.29%**	**17.71%**	**5.66%**
3	-	-	-	-	**- unroll - matrix array_partition complete - vector array_partition complete**	**43.64%**	**2.86%**	**18.82%**	**8.21%**

Table 6: A comparison of timing and PL resource usage for integer data type using different `HLS` optimizations applied to the matrix-vector algorithm.

We experiment also with other data types applying the same `HLS` directives. Now, as shown in Figure 30, using single-precision (`float`) data type, the iteration latency of the default implementation takes 16 clock cycles, $4x$ the integer version. The different timing occurs in the multiply adder block, which performs a multiplication of two operands and adds the full-precision product to a third operand ($C_M = 4$, $C_A = 5$). Due to the *fadd* (floating point add) operation the pipeline cannot start in one clock cycle, but in 5 at least, regardless the arrays partition.

When unrolling and complete data partition are applied, `Vivado HLS` cannot perform concurrent reads and multiplications. The read can start while the *fadd* operation of the previous element of arrays is performing, as shown in Fig. 31. The reader has not to infer that this is a general scenario with floating point arithmetic. The resulting hardware is the combination of the ability of `Vivado HLS` to map algorithm into hardware resources (helped by the

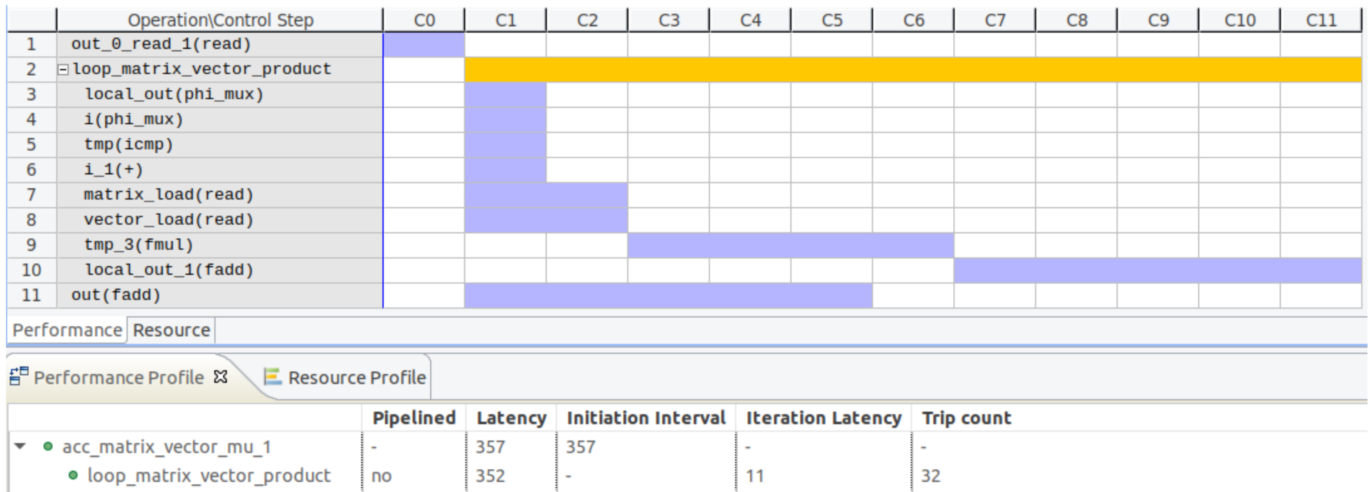

Figure 30: `Vivado HLS` resource estimates for default implementation of the matrix-vector algorithm with single-precision (`float`) data type. Source code is listed in Figure 10.

`HLS` directives added by the programmer) and the compute capability of the underlying FPGA.

The next Chapter is devoted to addressing how to estimate the floating-point capability of an FPGA.

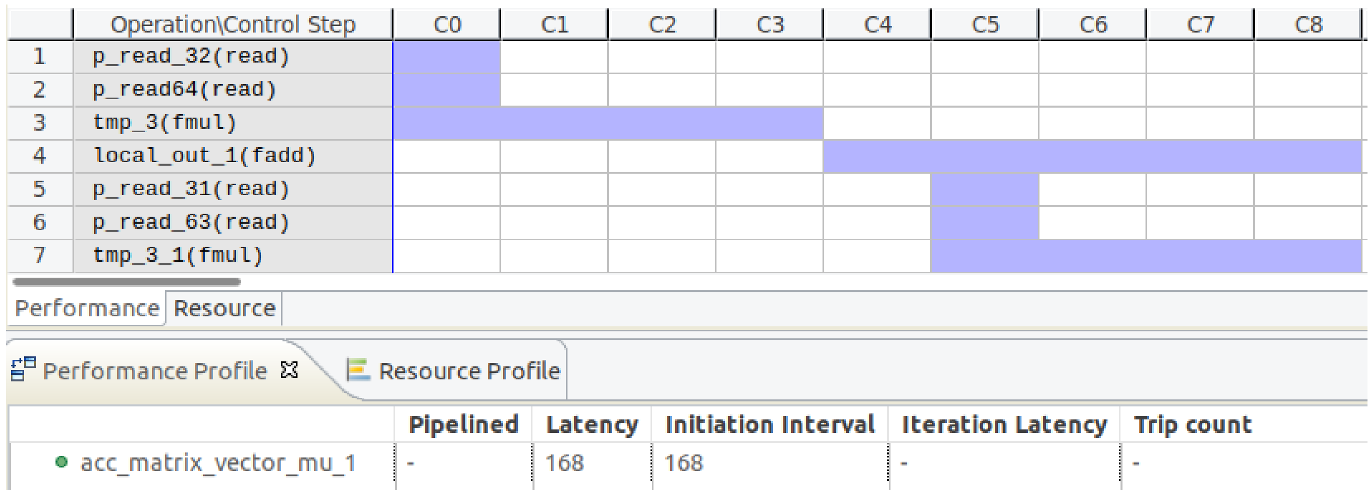

Figure 31: `Vivado HLS` resource estimates for unroll and complete data partition implementation of the matrix-vector algorithm with single-precision (`float`) data type. Source code is listed in Figure 10.

Table 7 summarizes the results for the single-precision implementation, while Table 8 reports numbers for double-precision arithmetic.

8.2.3. Matrix-vector multiplication optimizations

The source code listed in 10 performs the scalar product between the rows of the matrix and the vector in chunks. Every time the kernel is called, the PL gets `(((2 * BLOCK) + 1 ) * sizeof(MyData))` bytes as input, and returns `(sizeof(MyData))` bytes as output. To accomplish the entire calculation the kernel is called `(size`^{2} `/ BLOCK)` times, where `size` is the dimension of the output vector. Then, the total memory traffic (input and output) to perform the

Data Type : float									
TIMING (clock cycles)					HSL pragmas	PL RESOURCES			
Function latency	Loop latency	Iteration latency	Initiation interval	Trip count	Optimization	DSP48E	BRAM_18K	LUT	FF
357	352	11	-	32	none	2.27%	3.57%	17.35%	5.66%
171	165	11	5	32	- pipeline II=1	2.27%	3.57%	17.37%	5.66%
170	-	-	-	-	- unroll	2.27%	3.57%	19.09%	5.79%
168	-	-	-	-	- unroll - matrix array_partition complete - vector array_partition complete	2.27%	2.86%	20.11%	7.82%

Table 7: A comparison of timing and PL resource usage for single-precision data type using different `HLS` optimizations applied to the matrix-vector algorithm.

Data Type : double									
TIMING (clock cycles)					HSL pragmas	PL RESOURCES			
Function latency	Loop latency	Iteration latency	Initiation interval	Trip count	Optimization	DSP48E	BRAM_18K	LUT	FF
421	416	13	-	32	none	6.36%	6.43%	20.32%	6.50%
173	167	13	5	32	- pipeline II=1	6.36%	6.43%	20.39%	6.56%
172	-	-	-	-	- unroll	6.36%	6.43%	22.12%	6.76%
170	-	-	-	-	- unroll - matrix array_partition complete - vector array_partition complete	6.36%	5.0%	21.86%	10.23%

Table 8: A comparison of timing and PL resource usage for double-precision data type using different `HLS` optimizations applied to the matrix-vector algorithm.

entire calculation is (in bytes):

$$T^{chunk}(s, b, \alpha) = \frac{s^2}{b}(2b + 1)\alpha \tag{18}$$

where s is the size, b the `BLOCK`, and α = `sizeof(MyData)`. The minimum memory traffic is achieved when `BLOCK = size`, $T^{chunk}_{min}(s, s, \alpha) = s(2s + 1)\alpha$. Regarding the arithmetic operations, the kernel performs the following total operations:

$$W^{chunk}(s, b, [\otimes], [\oplus]) = \frac{s^2}{b}\left(b[\otimes] + (b + 1)[\oplus]\right) \tag{19}$$

where $[\otimes]$ and $[\oplus]$ are the multiplication and addition operations, respectively. The arithmetic intensity, I, is the ratio of the operations to the memory traffic. In this case:

$$I^{chunk}(b, \alpha, [\otimes], [\oplus]) = \frac{W^{chunk}}{T^{chunk}} = \frac{b[\otimes] + (b + 1)[\oplus]}{\alpha(2b + 1)} \tag{20}$$

Another approach is to decompose the matrix in blocks (sub-matrices), whose size is PL dependent. The source

code of the matrix-vector-block algorithm is listed in 11. The host, through the function *hw_matrix_vector_mul*, splits the matrix in blocks and issues the kernel, called *acc_m_v_block*, that performs on the PL the calculation. The scalar product of the old kernel is now nested inside a loop over the columns of the sub-matrix (labelled as *loop_data*).
1215 The `#pragma HLS pipeline II=1` is applied to the loop, then loop unrolling of the nested loop (*loop_dot_product*) is automatically performed by `Vivado HLS`.

The latest rises the warning during the compilation: *unable to schedule 'load' operation on array 'matrix' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'matrix'.* In the case of integer implementation, the pipeline has II=16. The II=1 can be achieved partitioning the arrays. Since
1220 the inner loop is fully unrolled (by a factor of BLOCK because of the pipeline of the outer loop), the programmer has to guarantee that no more than two addresses in the arrays are accessed per loop iteration. While the vector is allocated as a unique chunk of memory, the sub-matrix is allocated as a chunk of contiguous memory following the row-major order (also called C-order), a common method of storing multidimensional arrays in linear storage. The cyclic partitioning by a factor of BLOCK/2 guarantees that all columns of the sub-matrix can be accesses simultaneously.
1225 The same line of reasoning is applied to the vector. Moreover, using the cyclic partitioning the iteration latency is six clock cycles, two of them needed to load data (stored on `BRAMs`). The complete sub-matrix partitioning allows an iteration latency of five clock cycles (only one clock cycle is required to load data from `FFs`), but the `FFs` utilization grows up to 39.33%.

In this case the kernel is issued `(size / BLOCK)`^{2} times to accomplish the entire calculation, i.e. `BLOCK` times less than the previous implementation. Every time the kernel is issued, the PL gets `((BLOCK`^{2} `+ (2 * BLOCK)) * sizeof(MyData))` bytes as input, and returns `(BLOCK * sizeof(MyData))` bytes as output. In this case the total memory traffic is (in bytes):

$$T^{block}(s, b, \alpha) = \frac{s^2}{b}(b + 3)\alpha \tag{21}$$

The minimum memory traffic is achieved when `BLOCK = size`, so $T^{block}_{min}(s, s, \alpha) = s(s+3)\alpha$. The kernel performs the following total operations:

$$W^{block}(s, b, [\otimes], [\oplus]) = \frac{s^2}{b}\left(b[\otimes] + (b+1)[\oplus]\right) = W^{chunk}(s, b, [\otimes], [\oplus]) \tag{22}$$

So the arithmetic intensity is:

$$I^{block}(b, \alpha, [\otimes], [\oplus]) = \frac{W^{block}}{T^{block}} = \frac{b[\otimes] + (b+1)[\oplus]}{\alpha(b+3)} \tag{23}$$

Comparing both approaches when `BLOCK = 32`,

$$\frac{T^{chunk}(s, 32, \alpha)}{T^{block}(s, 32, \alpha)} \simeq 1.9$$

the second reduces the total memory traffic between the PS and the PL by almost a factor two. So, the resulting
1230 arithmetic intensity of the second approach is almost twice the first.

The reader has always to keep in mind that the estimation of the arithmetic intensity of one kernel is fundamental in order to provide performance estimates of that kernel running on a specific architecture (or device), by showing inherent hardware limitations, and potential benefit and priority of optimizations. The `Roofline model` is an intuitive

visual performance model by which it is possible to compare the machine (device) peak performance, the machine peak bandwidth, and the kernel arithmetic intensity. Usually the machine peak performance can be derived from architectural manuals, the machine peak bandwidth is instead obtained via benchmarking. Since an FPGA has not a fixed architecture, its Roofline might appear misleading. However, the Roofline model has been extended to better suit specific architectures and the related characteristics, such as FPGAs. Roofline model applied to the FPGAs is discussed in Section 9.

The estimation of the kernel arithmetic intensity can give insights if that kernel is compute-bound or memory-bound on the target machine (device). Typically, on accelerators, like GPUs or FPGAs, the best benefit in terms of performance is achieved by a compute-intensive kernel, due to the massively-parallel architecture of such devices.

Listing 11: Example code of the matrix-vector-block algorithm. Arrays are partitioned in block. Pipeline and array partition are applied in order to achieve II=1.

```
#define BLOCK 32
const unsigned int BSIZE = BLOCK;

#if defined(_SMP_)
 #pragma omp target device(smp)
#else
 #pragma omp target device(fpga)
#endif
 #pragma omp task in ([BSIZE*BSIZE]matrix, [BSIZE]vector) inout([BSIZE]out)
void acc_m_v_block(const MyData *const restrict matrix,
                   const MyData *const restrict vector,
                         MyData *const restrict out)
{
# pragma HLS inline off

# pragma HLS array_partition variable=matrix cyclic factor=16
# pragma HLS array_partition variable=vector cyclic factor=16

  /* loop over rows of blockMatrix */
 loop_data:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
#     pragma HLS pipeline II=1

      MyData local_out = (MyData)0.0;

    loop_dot_product:
      for (u_int16_t j=0 ; j<BLOCK ; j++)
        {
          local_out += (matrix[(i * BLOCK) + j] * vector[j]);
        }
      /* store result */
```

```
      out[i] += local_out;
    }

  return;
}

void copy_block(const MyData *const restrict matrix,
                      MyData *const restrict block,
                const u_int32_t              matrix_size,
                const u_int32_t              row,
                const u_int32_t              col)
{
  for (u_int32_t i=0 ; i<BLOCK ; i++)
    {
      const u_int32_t ii = (i + row);

      for (u_int32_t j=0 ; j<BLOCK ; j++)
        {
          const u_int32_t jj = (j + col);

          block[(i * BLOCK) + j] = matrix[(ii * matrix_size) + jj];
        }
    }

  return;
}

void hw_matrix_vector_mul(const MyData *const restrict matrix,
                          const MyData *const restrict vector,
                                MyData *const restrict out,
                          const u_int32_t              size)
{
  /* init to zero out */
  memset(out, 0, (sizeof(MyData) * size));

  for (u_int32_t bi=0 ; bi<size ; bi+=BLOCK)
    {
      for (u_int32_t bj=0 ; bj<size ; bj+=BLOCK)
        {
          /* local block of matrix */
          MyData blockMatrix[BLOCK2];

          /* copy block of matrix from global to local memory */
          copy_block(matrix, blockMatrix, size, bi, bj);

          /* perform matrix_vector_block multiplication on hardware */
          acc_m_v_block(blockMatrix, &vector[bj], &out[bi]);
```

```
#         pragma omp taskwait
        } /* bj */
    } /* bi */

  return;
}

/******************** VIVADO  HLS  REPORT ********************/
/* Target Board: ZedBoard                                    */
/*************************************************************/
/* DSP48E   96    used |    220 available-43.64% utilization */
/* BRAM_18K 42    used |    280 available-15.00% utilization */
/* LUT      11011 used |  53200 available-20.70% utilization */
/* FF       9034  used | 106400 available- 8.49% utilization */
/*************************************************************/
```

8.3. Matrix-matrix multiplication

The matrix-matrix multiplication is a fundamental algorithm widely applied in many research areas and, for that reason, it is usually discussed when the reader is introduced to either a new computational framework or device.

8.3.1. Background

The matrix multiplication is a binary operation that combines two matrices into a third. Basically the operation is a recursive scalar product between the vectors that compose the two matrices. Starting from the matrices A, with dimensions $n \times m$, and B, with dimensions $m \times p$, the matrix product creates an $n \times p$ matrix C defined as:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} B_{k,j} \tag{24}$$

In order to estimate the FLOPs, we need to count all the math operations (i.e. multiplications and sums) required to obtain the result. Following the Eq. 24, to compute all the elements of C, $n \times p$ operations are needed to be performed (i.e. one for each C_{ij} element), where each operation involves m multiplications and $(m-1)$ sums. So, the total number of FLOPs needed is given by $(n \times p \times (2m-1))$. In case of square matrices (i.e. $n = p = m$) then the total number of FLOPs is $(2n^3 - n^2)$.

In order to estimate the memory traffic, we need to take into account the data of input and output. Strictly speaking, we should count all the memory accesses needed to obtain the result. However, the memory traffic is heavily dependent on the properties of the chosen platform/device, such as for instance the structure of the cache hierarchy in case of CPUs. So, we define the memory traffic as the total bytes of input and output of the kernel, i.e. $\alpha\left[(n \times m) + (m \times p) + (n \times p)\right]$, where α is the number of bytes of the data type. Thus, for square matrices, the memory traffic is $3\alpha n^2$.

To get the computational intensity (FLOPs/bytes), it is needed to divide the count of FLOPs by the memory traffic in bytes. Finally, for square matrices we get:

$$I(n, \alpha) = \frac{2n^3 - n^2}{3\alpha n^2} \approx \frac{2n}{3\alpha} \quad \text{(when } n \text{ is large)}. \tag{25}$$

Listing 12: The common loop structure for matrix multiplication.

```
void matrixmul(Mydata A[N][M],
               MyData B[M][P],
               MyData C[N][P])
{
 row: for (int i=0 ; i<N ; i++)
    {
    col: for (int j=0 ; j<P ; j++)
        {
          MyData ABij = (MyData)0.0;

        product: for (int k=0 ; k<M ; k++)
            {
              ABij += (A[i][k] * B[k][j]);
            }

          C[i][j] = ABij;
        } /* cols */
    } /* rows */

  return;
}
```

The most common method to compute the matrix multiplication is using three nested loops, as in the source code listed 12. The outer for loops, labeled as `row` and `col`, iterate across the rows and columns of the output matrix C. The innermost for loop computes the dot product of one row of A and one column of B. It is important to note that each dot product is a completely independent set of computations that yields one element of C. Basically, the calculation performs P matrix-vector multiplications, one for each column of the matrix B.

8.3.2. Block matrix multiplication

A matrix can be thought as composed by sub-matrices, called blocks. Hence, when we talk about matrix multiplication, in Equation 24, we could think about single elements $A_{i,k}$ and $B_{k,j}$ as blocks of the parent matrices A and B. In this case, in order to compute the block $C_{i,j}$, it is needed to compute two block products and two matrix sums, as shown in Figure 32.

Matrix blocking turns out to be a very useful approach for a number of reasons. First is that blocking is an easy way to maintain the common structure for matrix multiplication (listed in 12) and experiment with some optimizations already used in previous Sections of the technical report. Second is that the size of the block can be chosen according to the hardware resources available on the FPGA instead of the size of the problem. Another advantage is that we can operate on the data in parallel using multiple devices (e.g. some blocks are handled by the PS and others by the PL) or we can exploit multiple instances inside the PL (e.g. it is like to exploit multiple FPGAs).

We start from a source code that allows to handle matrices of arbitrary size and performs the calculation in blocks, listed in 13.

Figure 32: Blocked decomposition of the matrix multiplication of two 2 x 2 matrices. The entire AB product is decomposed into four matrix multiply operations (only the first is shown) operating on a 1 x 2 block of A and 2 x 1 block of B. $C_{00} = A_{00}B_{00} + A_{01}B_{10}$.

Listing 13: The structure for blocking matrix multiplication.

```
#define BLOCK  4
#define BLOCK2 (BLOCK * BLOCK)
typedef int32_t MyData;
const unsigned int B2SIZE = (BLOCK * BLOCK);                       1395

#if defined(_SMP_)
 #pragma omp target device(smp)
#elif (_SMP_FPGA_)
 #pragma omp target device(smp, fpga)                               1400
#else
 #pragma omp target device(fpga)
#endif
 #pragma omp task in([B2SIZE]A, [B2SIZE]B) out([B2SIZE]C)
void acc_block_block_mul(const MyData *const restrict A,           1405
                         const MyData *const restrict B,
                               MyData *const restrict C)
{
# pragma HLS inline off
                                                                    1410
 loop_row:
  for (u_int8_t row=0 ; row<BLOCK ; row++)
    {
    loop_col:
      for (u_int8_t col=0 ; col<BLOCK ; col++)                      1415
        {
          MyData sum = (MyData)0.0;

          loop_product:
            for (u_int8_t k=0 ; k<BLOCK ; k++)                      1420
```

```
            {
              sum += (A[(row * BLOCK) + k] * B[(k * BLOCK) + col]);
            }
          C[(row * BLOCK) + col] = sum;
        } /* loop col */
    } /* loop row */

  return;
}

void hw_matrix_matrix_block_mul(const MyData *const restrict A,
                                const MyData *const restrict B,
                                      MyData *const restrict C,
                                const u_int32_t            size)
{
  for (u_int32_t bi=0 ; bi<size ; bi+=BLOCK)
    {
      for (u_int32_t bj=0 ; bj<size ; bj+=BLOCK)
        {
          /* initialize to zero the block */
          init_block(C, size, bi, bj);

          for (u_int32_t bk=0 ; bk<size ; bk+=BLOCK)
            {
              /* local block of matrix A */
              MyData blockA[BLOCK2];
              /* local block of matrix B */
              MyData blockB[BLOCK2];
              /* local block of matrix C */
              MyData blockC[BLOCK2];

              /* copy block of A from global to local memory */
              copy_block(A, blockA, size, bi, bk);
              /* copy block of B from global to local memory */
              copy_block(B, blockB, size, bk, bj);

              /* perform block multiplication */
              acc_block_block_mul(blockA, blockB, blockC);

#             pragma omp taskwait

              /* store blockC to global memory */
              store_block(blockC, C, size, bi, bj);
            } /* bk */
        } /* bj */
    } /* bi */
```

```
    return;
}
```

The host through the function *hw_matrix_matrix_block_mul* splits the matrices A and B in blocks, called $blockA$ and $blockB$, respectively. After that, the kernel *acc_block_block_mul* is issued and the calculation is performed on the `smp`, or `fpga`, or both (selected by the user through the Makefile). In further coming Technical Report about `OmpSs@FPGA` ecosystem we will describe more in detail the `OmpSs` directives that allow to exploit multiple devices.

The size of the block is chosen at compile time, and, as discussed above, it is a trade-off between the size of the problem and the available hardware resources.

The Table 9 shows a comparison of timing and PL resources usage for integer data type using different `HLS` optimizations applied to the source code listed in 13, when $BLOCK = 4$.

We start our optimization process with the usual `#pragma HLS pipeline II=1` applied to the *loop_col* loop. The result is that the inner-most loop, labelled as *loop_product*, is automatically fully enrolled by `Vivado HLS`. The loop trip count (referred to *loop_row*) becomes 16 because `Vivado HLS` automatically performs a loop flattening. This optimization allows nested loops (in our case labelled as *loop_row* and *loop_col*) to be flattened into a single loop hierarchy with improved latency. In the RTL implementation, it requires one clock cycle to move from an outer loop to an inner loop, and from an inner loop to an outer loop. Flattening nested loops allows them to be optimized as a single loop. This saves clock cycles, potentially allowing for greater optimization of the loop body logic. This optimization is automatically applied by `Vivado HLS` because only the inner-most *loop_col* loop has loop body content, i.e. the unrolled *loop_product*. `Vivado HLS` makes available the `pragma HLS loop_flatten` to be put in the source code within the boundaries of the nested loop. The `pragma` takes the optional `off` keyword that prevents flattening from taking place.

Without any array partitioning/reshaping of the matrices it is not possible to achieve an II=1 due to the limited memory port accesses, as expected. In the body of *loop_product* the matrix A is accessed by rows, while the matrix B is accessed by columns, and both are stored in row-major order (i.e. the consecutive elements of a row reside next to each other). Thus, the matrix A requires a cyclic partitioning/reshaping, while the matrix B requires a block partitioning/reshaping, both using a `factor=BLOCK`. Through this approach it is possible to achieve a II=1 with the same function latency, but with different FPGA resource usage. It is worth noticing that reshaping is more resource-eager than partitioning with the same function latency, as shown in Table 9.

8.4. Histogram

Creating an histogram is a widely used function in many fields, e.g. image processing, and database processing.

Listing 14: Example code of the naive histogram algorithm. Input array (in) is partitioned in blocks and the local histogram is copied out only at the last iteration.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma  omp target device(fpga)
```

Data Type : integer (int32)									
TIMING (clock cycles)					HSL pragmas	PL RESOURCES			
Function latency	Loop latency (*loop_row*)	Iteration latency (*loop_row*)	Initiation interval (*loop_row*)	Trip count (*loop_row*)	Optimization	DSP48E	BRAM_18K	LUT	FF
297	296	74	-	4	none	1.36%	2.86%	16.64%	5.47%
36	34	5	2	16	loop_col: - pipeline II=1	2.73%	2.86%	17.06%	5.58%
20	18	4	1	16	loop_col: - pipeline II=1 - A array_reshape cyclic factor=4 - B array_reshape block factor=4	5.45%	2.86%	21.79%	7.03%
20	18	4	1	16	loop_col: - pipeline II=1 - A array_partition cyclic factor=4 - B array_partition block factor=4	5.45%	2.86%	16.98%	6.63%
8	6	4	1	4	loop_row: - pipeline II=1 - A array_partition cyclic factor=4 - B array_partition complete - C array_partition cyclic factor=4	21.82%	2.86%	20.37%	7.95%
2	-	-	-	-	function: - pipeline II=1 - A, B, C array_partition complete	87.27%	2.86%	21.73%	8.43%

Table 9: A comparison of timing and PL resource usage for integer (int32_t) data type using different HLS optimizations applied to the matrix-matrix-block algorithm. $BLOCK = 4$. The loop latency, the iteration latency, the initiation interval and the trip count refer to the outer-most loop labelled as *loop_row*.

```
#pragma omp task in([BSIZE]in) out([BSIZE]histog)
void histogram(const MyData  *const restrict in,
                     MyData  *const restrict histog,
               const u_int8_t                copy_out,
               const u_int8_t                reset)

{
#pragma HLS inline off
```

```
  static u_int32_t local_histog[BLOCK];

  if (reset)
    {
    loop_reset:
      for (u_int16_t i=0 ; i<BLOCK ; i++)
        {
          local_histog[i] = 0;
        }
    }

 loop_histogram:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
      local_histog[in[i]]++;
    }

  /* copy local_histog to histog */
  if (copy_out)
    {
    loop_copy:
      for (u_int16_t i=0 ; i<BLOCK ; i++)
        {
          histog[i] = local_histog[i];
        }
    }

  return;
}


/********************* VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/***********************************************************/
/* DSP48E       0 used |     220 available -  0.0% utilization */
/* BRAM_18K     9 used |     280 available - 3.21% utilization */
/* LUT       8372 used |   53200 available -15.74% utilization */
/* FF        5237 used | 106400 available - 4.92% utilization */
/***********************************************************/
```

The code, listed in 14, ends up to be very similar to the running total in Section 8.1. The main difference is that the latest performs one accumulation, while in histogram computes one accumulation for each bin. As expected, when pipelining the for loop (labelled as `loop_histogram`), it turns out the same problem as with the code in Figure 23. It is possible to achieve only a loop II of 2 due to load/store memory conflict, as stated by the analysis perspective shown in Figure 33. This is due to the fact that the algorithm is reading from the histogram and writing to the same address

in every iteration of the loop. `Vivado HLS` cannot make any assumption on the correlation of the input values. The worst case happens when the input array contains the same value in two contiguous addresses, so `Vivado HLS` must alternate between reads and writes. If the input data never have two consecutive values[24], extra information must be provided to `Vivado HLS` tool so that it will be able to synthesize a circuit that reads at one location while writing at the previous one (i.e. different addresses correspond to different values).

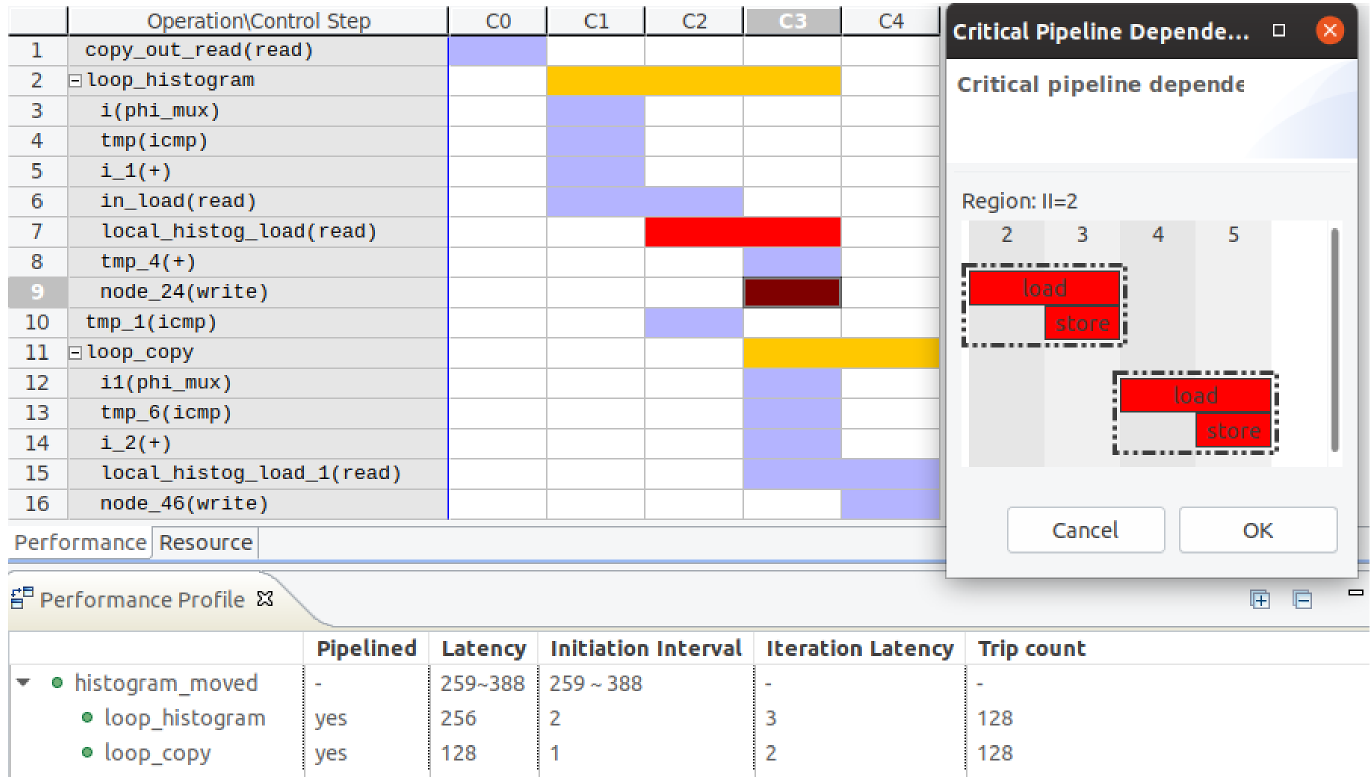

Figure 33: `Vivado HLS` Analysis Perspective of the pipeline version of the histogram algorithm listed in the Listing 14. Red indicates conflicts. Clicking the red cycles it is possible to see where the conflicts are.

In `Vivado HLS` this is done using the `HLS dependence` directive, that asserts the function has some preconditions, i.e. it indicates to `Vivado HLS` that reads and writes to the `local_histog` array are dependent only in a particular way. In our case, the `inter-iteration` dependencies (i.e. dependence between different loop iterations) consist of a read operation after a write operation (RAW), meaning that the write instruction uses a value used by the read instruction. The directive allows the programmer to specify the inter-iteration distance for array access. Since the input data never has two consecutive values, than the correct distance is 2, because it could be the case that $in[i+2] == in[i]$.

Another way to achieve II=1 in the `loop_histogram` is to partition completely the array `local_histog` using `FF` resources, since data written on `FF` on one clock cycle is available on the next clock cycle. This circuit results in a large number of `FF` resources, so that preventing to store large histograms. Table 10 summarizes the timing achieved and PL resources using the `HLS` optimizations discussed above.

[24]It is our case since input array is filled with a pseudo-random uniform distribution of values in the range $[0, BLOCK)$, so that two consecutive values cannot be the same - source code on GitHub repository

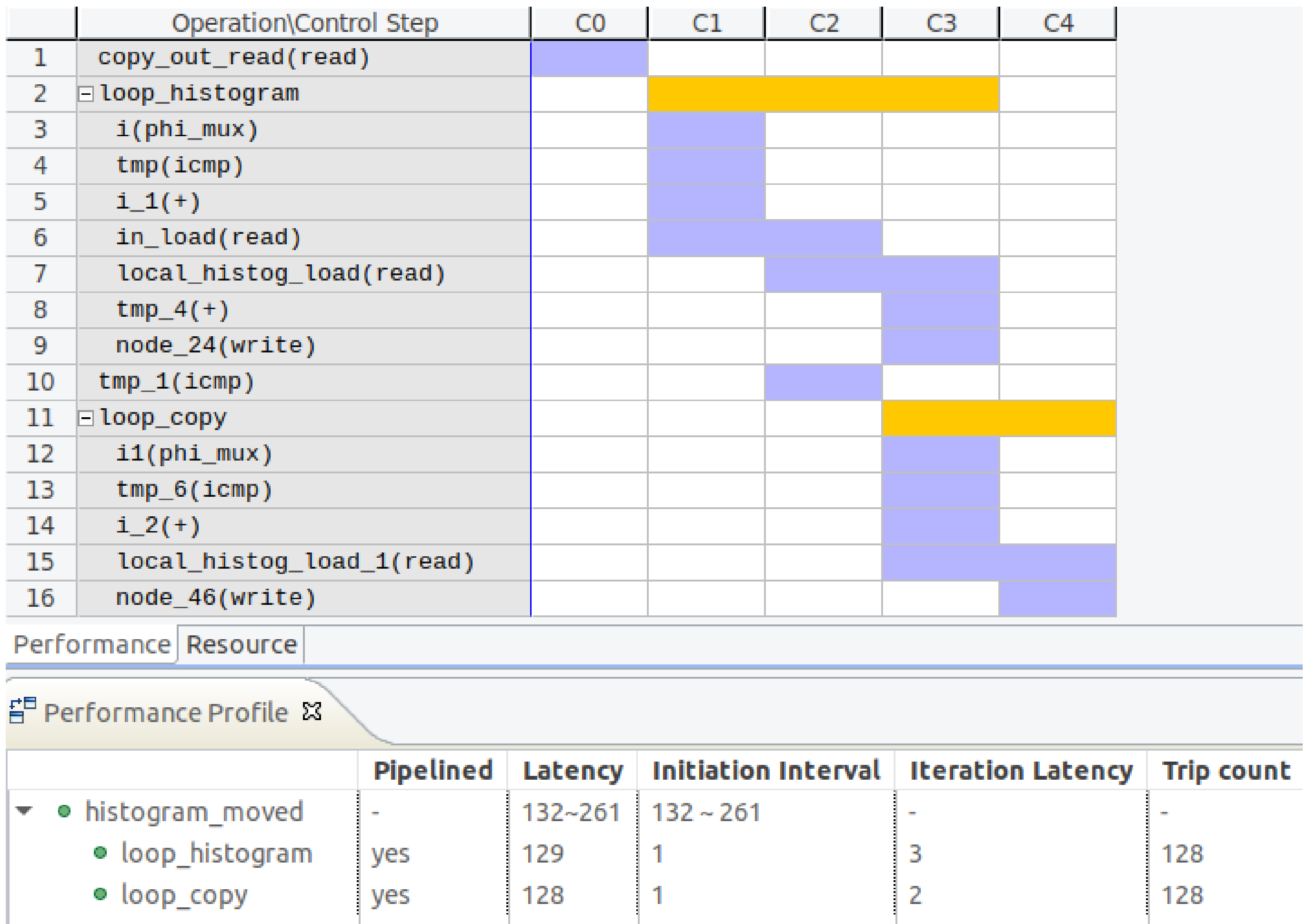

Figure 34: `Vivado HLS` Analysis Perspective of the pipeline version of the histogram algorithm listed in the Listing 14, adding `#pragma HLS DEPENDENCE variable=local_histog inter RAW distance=2` to the body of the function. Conflicts of Figure 33 disappear, achieving II=1 on `loop_histogram`.

Data Type : int32								
TIMING (clock cycles)				**HLS pragmas**	**PL resources**			
Loop latency	**Iteration latency**	**Initiation interval**	**Trip count**	**Optimization**	**DSP48E**	**BRAM_18K**	**LUT**	**FF**
512	**4**	**-**	**128**	**None**	**0.0%**	**3.21%**	**15.74%**	**4.92%**
256	**3**	**2**	**128**	**- pipeline II=1**	**0.0%**	**3.21%**	**15.87%**	**4.89%**
129	**3**	**1**	**128**	**- pipeline II=1 - dependence variable=local_histog inter RAW distance=2**	**0.0%**	**3.57%**	**15.83%**	**4.88%**
129	**3**	**1**	**128**	**- pipeline II=1 - array_partition variable=local_histog complete**	**0.0%**	**2.86%**	**19.0%**	**8.72%**

Table 10: A comparison of timing and PL resource usage for integer data type using different `HLS` optimizations applied to the histogram algorithm listed in 14. Timing refers to the `loop_histogram` loop, while resource usage for the whole synthesized kernel.

Up to now HLS optimizations rely on the fact that it is guaranteed the input data set never has two consecutive values. However, the general algorithm has to take into account that consecutive values may be the same. In this case, a possible solution is to use a simple register to perform the histogram accumulation with a minimal amount of FPGA-resources and delay. Instead, when the input does not contain two consecutive values, the circuit performs a write and a read at two different addresses of the histogram. The first is needed to store the count of the `(i-1)-th` element of the input, while the latest performs the load the count of the `i-th` element of the input array. The code to accomplish this is listed in 15. The code uses a local variable `old_value` to store the bin for the previous loop iteration and another local variable `acc` to store the count for such a bin. Within the loop, the algorithm checks if the bin is the same as the previous iteration. If so, the accumulator (`acc`) is incremented, otherwise the accumulator is store in the histogram at the address relative to the old bin (`old_value`), and then the accumulator is updated and incremented to the correct new value (`value`) in the histogram.

Listing 15: Example code of the alternative histogram algorithm. The **if/else** structure attempts to remove the read/write dependency from the for loop labelled as `loop_histogram`. Input array (in) is partitioned in blocks and the local histogram is copied out only at the last iteration.

```
typedef int32_t MyData;
#define BLOCK 128
const unsigned int BSIZE = BLOCK;

 #pragma omp target device(fpga)
 #pragma omp task in([BSIZE]in) out([BSIZE]histog)
void histogram(const MyData  *const restrict in,
                     MyData  *const restrict histog,
               const u_int8_t                copy_out,
               const u_int8_t                reset)

{
 #pragma HLS inline off

  static u_int32_t local_histog[BLOCK];

  if (reset)
    {
    loop_reset:
      for (u_int16_t i=0 ; i<BLOCK ; i++)
        {
          local_histog[i] = 0;
        }
    }

/*****************************************************************/

  /* load the first value */
  MyData old_value = in[0];
```

```
  /* accumulator */
  MyData acc = local_histog[old_value];

 loop_histogram:
  for (u_int16_t i=0 ; i<BLOCK ; i++)
    {
      /* load value */
      MyData value = in[i];

      if (old_value == value) /* get the same value */
        {
          acc++;
        }
      else /* different value */
        {
          local_histog[old_value] = acc;
          acc = (local_histog[value] + 1);
        }

      /* update the old value */
      old_value = value;
    } /* loop */

  local_histog[old_value] = acc;

/************************************************************/

  /* copy local_histog to histog */
  if (copy_out)
    {
    loop_copy:
      for (u_int16_t i=0 ; i<BLOCK ; i++)
        {
          histog[i] = local_histog[i];
        }
    }

  return;
}


/********************** VIVADO HLS REPORT *******************/
/* Target Board: ZedBoard                                   */
/************************************************************/
/* DSP48E      0 used |    220 available -  0.0% utilization */
/* BRAM_18K    9 used |    280 available - 3.21% utilization */
```

```
/* LUT       8372 used |  53200 available -15.91% utilization */
/* FF        5237 used | 106400 available - 5.04% utilization */
/*************************************************************/
```

Synthesizing the code listed in 15, adding a pipeline to the `loop_histogram`, `Vivado HLS` issues a warning that states that it is unable to enforce a carried dependence constrain between "add" operation of variable `acc` (the operation `acc++`) and "store" operation of the same variable on `local_histog` (the operation `local_histog[old_value] = acc`). The conflict is shown in Figure 35.

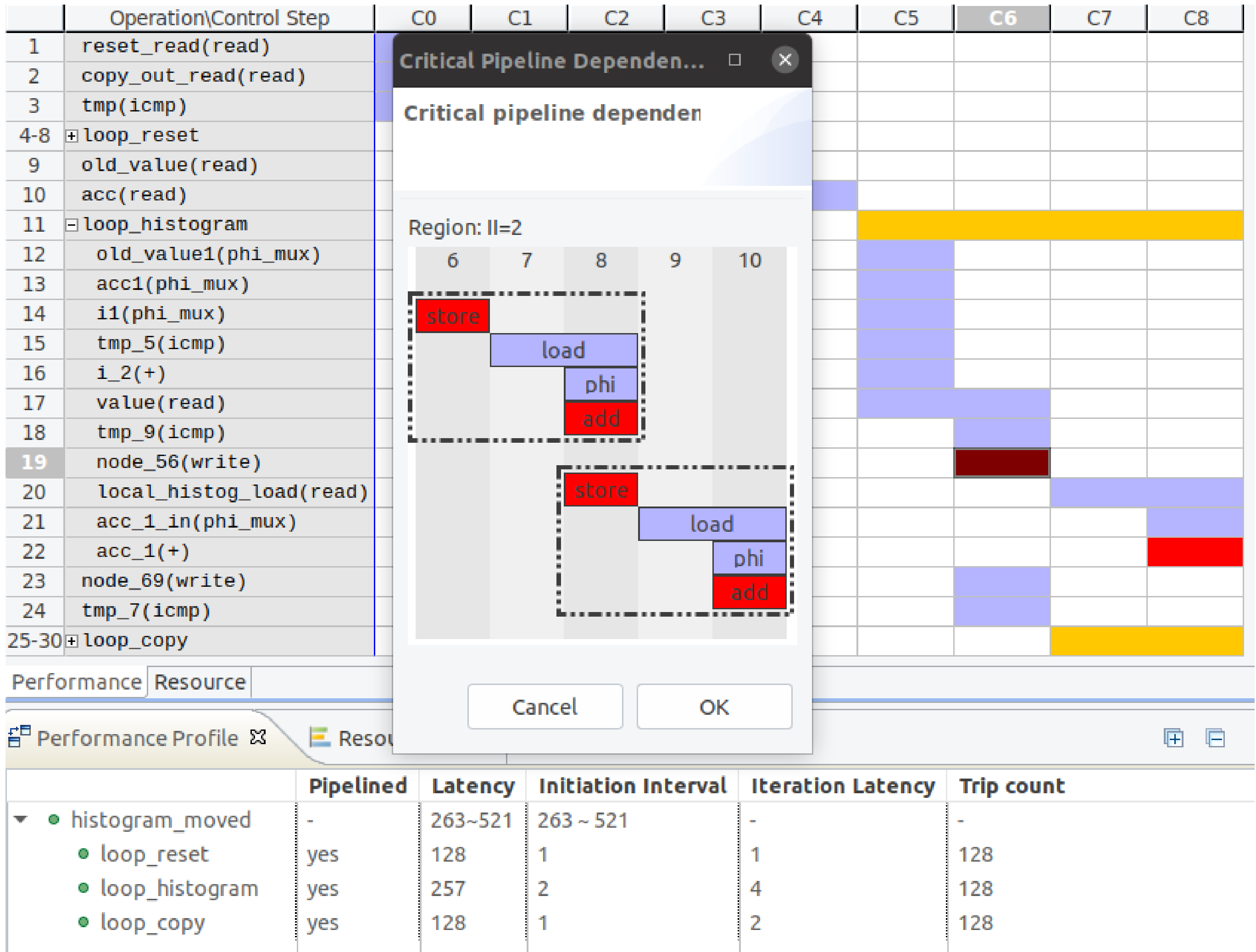

Figure 35: `Vivado HLS` Analysis Perspective of the pipeline version of the "general" histogram algorithm listed in the Listing 15. Red indicates conflicts. Clicking the red cycles it is possible to see where the conflicts are.

However, looking at the code, these operations can never be performed in the same clock cycle because they are in different branches of the **if/else** construct, and hence it is a false intra-dependence. Despite the simple algorithm, this example shows that `Vivado HLS` cannot determine this property of the actual design without the addition of programmer information. Using the `dependence` directive it is possible to inform the `HLS` tool about a particular property of the code itself. We remind the reader that specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware, therefore a careful analysis of the synthesized circuit is highly recommended.

Data Type : int32								
TIMING (clock cycles)				**HLS pragmas**	**PL resources**			
Loop latency	**Iteration latency**	**Initiation interval**	**Trip count**	**Optimization**	**DSP48E**	**BRAM_18K**	**LUT**	**FF**
384 ∼512	**3 ∼4**	**-**	**128**	**None**	**0.0%**	**3.21%**	**15.97%**	**5.03%**
257	**4**	**2**	**128**	**- pipeline II=1**	**0.0%**	**3.21%**	**16.08%**	**5.04%**
129	**3**	**1**	**128**	**- pipeline II=1 - dependence variable=local_histog inter RAW false**	**0.0%**	**3.57%**	**16.07%**	**5.0%**
129	**3**	**1**	**128**	**- pipeline II=1 - array_partition variable=local_histog complete**	**0.0%**	**2.86%**	**20.62%**	**8.81%**

Table 11: A comparison of timing and PL resource usage for integer data type using different `HLS` optimizations applied to the "general" histogram algorithm listed in 15. Timing refers to the `loop_histogram` loop, while resource usage for the whole synthesized kernel.

9. Roofline model

In this Section we describe how to measure the computing performance and the energy-efficiency to be expected of an FPGA device. As a matter of fact, Astronomy and Astrophysics often rely on floating-point intensive HPC workloads, so we want to provide a general method for estimating their performance as accelerators for real scientific applications. The method we follow is based on the FER (FPGA Empirical Roofline) tool [3], developed within the EuroExa project[25], and avaible on GitHub[26]. FER allows to empirically estimate the maximum bandwidth between the DRAM memory and the FPGA, as well as the maximum single- or double-precision computing throughtput of the FPGA. At the moment, to do so, it streams an array of single- or double-precision elements into the FPGA, where it performs a chain of subsequent Fused Multiply-Add (FMA) operations, one for each array element, copying back the result in another array. The loop over array elements is pipelined reaching an II=1 and thus, once the pipeline is filled, FER performs one read, one write and FLOP_ELEM (defined by the programmer) FLOPs for each FPGA clock cycle. Half of the FLOP_ELEM will be multiplications and the other half additions. FER relies on OmpSs@FPGA, using the docker image version 1.3.2 and Vivado 2017.3.1. We slightly change the FER source code to make it working with the latest OmpSs@FPGA version 2.1.0, used in this technical report. As usual, our version of the FER code is available in the GitHub repository associated with the technical report.

Moreover, we will use FER also to assess the energy-efficiency of the FPGA on the ZedBoard, using an external power meter and thus obtaining the maximum reachable FLOP/Watt.

9.1. FPGA Empirical Roofline

FER is able to extract the floating-point throughput from an FPGA using its DSP48s. This synthetic benchmark has the capability of tuning the computational intensity, i.e. the FLOPs/Byte ratio performed by the kernel, allowing to find out the floating-point throughput (GFLOPs/sec) and the maximum memory bandwidth between the FPGA and the host DRAM (GByte/sec).

Listing 16: The FER kernel function. For each input array element are computed (FLOP_ELEM / 2) FMA operations, accounting for a computational intensity of (FLOP_ELEM / (2 * sizeof(MyData)).

```
/********** PARAMETERS ****************/
/* Data type */
typedef float MyData;

/* Number of floating-point operations */
#define FLOP_ELEM    2
/**************************************/

/******** MACROS for Ops **************/
#define REP2(S)        S ;        S
#define REP4(S)    REP2(S);    REP2(S)
```

[25] https://euroexa.eu

[26] https://baltig.infn.it/EuroEXA/FER

```
#define REP8(S)   REP4(S);   REP4(S)
#define REP16(S)  REP8(S);   REP8(S)
#define REP32(S)  REP16(S);  REP16(S)
#define REP64(S)  REP32(S);  REP32(S)

#define SUM(a,b,c)  ((a) = (b) + (c))
#define FMA(a,b,c)  ((a) = ((a) * (b)) + (c))
/*********************************************/



 #pragma  omp target no_localmem_copies device(fpga)
 #pragma  omp task in([BSIZE]input) out([BSIZE]output)
void hw_kernel(const MyData *const __restrict__ input,
                     MyData *const __restrict__ output)
{
 main_loop:
  for (u_int32_t i=0 ; i<DIM ; i++)
    {
#    pragma HLS pipeline II=1

      const MyData alpha = 0.5;

      /* load element from DRAM to FPGA register */
      const MyData elem = input[i];

      MyData beta = 0.8;

#if   (FLOP_ELEM == 16)              /* 16 FLOPs */
      REP8(FMA(beta, elem, alpha));
#endif

#if   (FLOP_ELEM == 32)              /* 32 FLOPs */
      REP16(FMA(beta, elem, alpha));
#endif

      ....

      /* store result */
      output[i] = beta;
    }

  return;
}
```

The kernel is reported in Listing 16 (`https://www.ict.inaf.it/gitlab/david.goz/oats-technical-report-ompss_at_fpga.git`). In the first `#pragma omp` directive targeting the FPGA device, we explicitly avoid data transfer with the clause `no_localmem_copies` in order to directly access the host DRAM from the FPGA. During the kernel execu-

tion, one element of the `input` array is read for each FPGA clock cycle (the `main_loop` is pipelined), moved into the FPGA, processed into a pipeline computing a chain of dependent fused multiply add operations and finally written back to the DRAM into the `output` array. Once the pipeline is filled, for each clock cycle the FPGA is performing a read and a write of one element, and `FLOP_ELEM` floating-point operations (e.g. the computational intensity) concurrently. Combining this information with the kernel execution time, it it possible to infer the FLOPs/sec performance. To obtain the maximum achievable performance it is enough to increase the value of the `FLOP_ELEM` up to the point at which the FPGA resources are not enough to synthesize a working bitstream.

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	...	...	...	...	...	...	...	...	...
1-5	⊞I/O Ports																			
6	⊟Instances																			
7	hw_kernel_hls_autcud_U2										fmul									
8	hw_kernel_hls_autbkb_U1														fadd					
9-12	⊞Expressions																			

Performance | Resource

Performance Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
hw_kernel_mc	-	1048598	1048598	-	-
main_loop	yes	1048586	1	12	1048576

Resource Profile | Properties | Warnings | C Source

File: /home/darkenergy/Dropbox/Osservatorio/Software/Xilinx_FPGA/OmpSs_fpga_technical_report/FER/zedboard_HLS

```
27 typedef float MyData;
28 typedef unsigned int u_int32_t;
29 static void hw_kernel_moved(const MyData *const __restrict input, MyData *const __restrict output)
30 {
31   main_loop : for (u_int32_t i = 0; i < 1024 * 1024; i++)
32   {
33 #pragma HLS pipeline II=1
34     const MyData alpha = 5.000000000000000000000000000000000000000000000000000e-01;
35     const MyData elem = input[i];
36     MyData beta = 8.000000000000000444089209850062616169452667236328125000e-01;
37     beta = beta * elem + alpha;
38     output[i] = beta;
39   }
40   return ;
41 }
```

Figure 36: `Vivado HLS` Resource analysis perspective of the `FER` kernel listed in the Listing 16 when single precision arithmetic (`sizeof(MyData) == 4`) is used. The kernel is set to perform one FMA (`FLOP_ELEM == 2`), so one `fmul` and one `fadd`. The first performs the operation $op_{fmul} = (beta * elem)$ taking 4 clock cycles, and the second performs the operation $op_{fadd} = (op_{fmul} + alpha)$ requiring 5 clock cycles.

9.2. Results

The `Zynq-7020` (details in Section 6.5) that equips the `ZedBoard` is not suitable for real HPC workload because it is meant as an educational/development kit, i.e. the board is ideal for rapid prototyping and proof-of-concept

1760 development like the `FER` tool. However we still use our `FER` tool to evaluate the maximum single-precision performance of the `ZedBoard` at least, since double-precision arithmetic is actually resource-eager on such a device.

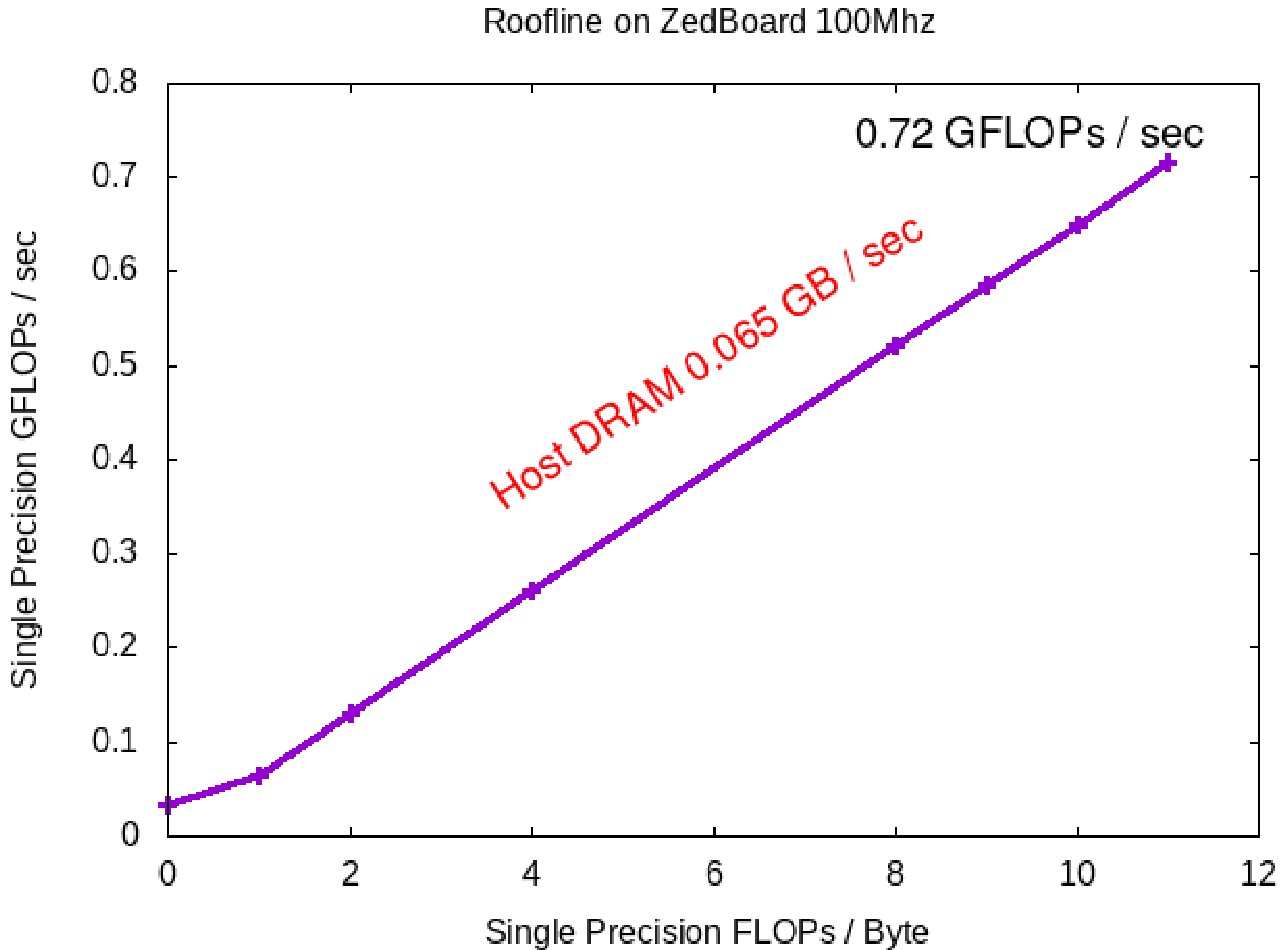

Figure 37: Experimental Roofline measure of the FPGA embedded on `ZedBoard` clocked at 100MHz. The maximum performance has been reached using 100% of the available `DSP48s`, computing 44 FMAs (i.e. 88 FLOP) per element, in single-precision (i.e 32-bit word).

The experimental Roofline is plotted in Figure 37. Using 100% of the available `DSP48s` on the FPGA, 0.72 GFLOPs/sec have been reached in single-precision, computing 44 FMAs operations on each element and processing one element per FPGA clock cycle. Concerning the bandwidth, 0.065 GB/sec of bidirectional bandwidth between the
1765 host DRAM and the FPGA have been reached, moving 32-bit element in and 32-bit element out.

The reader might argue that the expected bidirectional bandwidth should be 0.8 GB/sec, because 8 byte (2 x 32-bit word) are moved per clock cycle at 100MHz (when the pipeline is filled). Experimentally a value lower of an order of magnitude has been found. To investigate this, we run the FER tool without computing any FMA operation (it is enough to set the macro `#define FLOP_ELEM 0` in the header file `parameters.h`), so that the kernel simply performs a
1770 pipelined copy of the input array into the output one, achieving the same bidirectional bandwidth (i.e. $\simeq$0.06 GB/sec) for both single and double-precision arithmetic.

From the plot in Figure 37 we can derive the machine balance, i.e. the ratio $M_B = C/B \simeq 11$ FLOPs/byte,

where $C = 0.72$ GFLOPs/s is the maximum computational performance, and $B = 0.065$ Gbyte/s the maximum memory bandwidth. The Roofline assumes that a computational task performs a number of operations O on D data items exchanged with memory. The corresponding ratio $I = O/D$ is known as the arithmetic intensity. Kernel with arithmetic intensity lower than the machine balance are called memory-bound, and compute-bound otherwise.

9.3. Energy

A current sense resistor, which is installed on the `ZedBoard`, is used to calculate the power consumed by the `ZedBoard` while executing the designed application. The shunt-resistor is a $10m\Omega$ series resistor with the board's input supply line, and this resistor can be used to measure the entire board's current draw using the rule of Ohm. Since the voltage of the power supply is $12V$, the board's power consumption is provided by the formula:

$$P = \frac{V_{measured}}{10m\Omega} \times 12V \tag{26}$$

For our measurements we use the Benchtop Multimeter, Hewlett Packard model 34401A. After booting up the board, we measure the power consumption both in idle and during the execution of the FER benchmark, so we have been able to compute the average power drain in Watt at a sustained GFLOP/s rate, allowing us to compute the GFLOPs/Watt metric. In particular, we measure an average power drain in idle of 3.28W and running the FER benchmark of 3.47W performing 88 FLOP (i.e 44 FMAs), yielding an energy-efficiency of $\simeq$ 3.8 GFLOPs/W in single-precision for the `Artix-7` FPGA hosted on the `ZedBoard`.

Our FER synthetic benchmark allows the user to set the amount of FLOPs at compile time. When the number of FLOPs is set to zero, the kernel performs only the copy of the input array into the output array for all elements using a pipeline, allowing us to estimate the power drain for the data traffic from/to the main (host) memory. We measure $\simeq$ 0.5 GBytes/W.

DSPs usage [%]	**FLOP per Elem.**	**Performance [GFLOP/s] - [GByte/s]**	**Avg Power (without idle) [Watt]**	**Energy-efficiency [GFLOPs/W] - [GBytes/W]**
0.0	**0**	**0 - 0.065**	**0.13**	**0 - 0.5**
100	**88**	**0.72 - 0.065**	**0.19**	**3.79 - 0.34**

Table 12: Results concerning the execution of the FER synthetic benchmark using single-precision arithmetic, running on the `Artix-7` FPGA hosted by the `ZedBoard`. Average power drain is reported after subtracting the idle power drain.

For completeness, we investigate also the power-efficiency using double-precision arithmetic. At most 12 double-precision FMAs can be synthesized and in particular we measure a power draw of 0.26W (without idle) and a power-efficiency of 0.058 GFLOPs/W, accordingly. In comparison, 12 FMAs for single-precision floating-point operations result in a power draw of 0.18W (without idle) and a power-efficiency of 0.16 GFLOPs/W. This highlight the fact that this hardware is not only more efficient from the performance point of view in computing single-precision operations ($\approx 2\times$ faster), but is also much more energy-efficient in this condition ($\approx 2.7\times$ more energy-efficient).

10. Conclusions

1795 In this technical report we show how to use the directive based `OmpSs@FPGA` high-level programming framework, developed at the Barcelona Supercomputing Center, to enable FPGA-accelerated computing. We show how, thanks to the High Level Synthesis approach, FPGAs can be programmed using high level languages such as C/C++, highly reducing the programming effort required, allowing a faster design space exploration in terms of FPGA resource-usage and performance, and a much higher software portability. A large part of this technical report has been devoted
1800 to present algorithm case studies, which allow the reader to understand in detail how to re-engineer widely-applied algorithms to accomplish the goals of the designer.

In further coming technical report we will show more advanced features of `OmpSs@FPGA` framework and techniques to profile the target application in order to gather information regarding the application performance on the FPGA.

11. Acknowledgments

1805 The technical report was carried out within the EuroEXA FET-HPC (grant no. 754337) and ESCAPE (grant no. 824064) projects, funded by the European Unions Horizon 2020 research and innovation program.

12. Appendix: Compile OmpSs@FPGA programs

In this Appendix, we focus on the specific options of `Mercurium` to generate the bitstreams, hardware instrumentation and binaries.

To compile `OmpSs@FPGA` programs the user should follow the general `OmpSs` compilation procedure using the `Mercurium` compiler. Details are provided in the OmpSs User Guide[27].

12.1. Bitstreams

To generate the bitstream the user must enable the bitstream generation in the `Mercurium` compiler, using the `--bitstream-generation` flag and provide it the FPGA linker (aka AIT) flags with `--Wf` option.

For example, to compile an application, in debug mode, for the `ZedBoard`, with a target frequency of 100Mhz, the user can use the following command:

```
$ arm-linux-gnueabihf-fpgacc –debug –ompss –bitstream-generation <source_code.c> -o <executable_name_debug> –Wf, ”–board=zedboard, –clock=100, –name=<executable_name_debug>, –hwruntime=som”
```

12.2. Hardware instrumentation

The HW instrumentation can be enable using the flag `--instrument` (or `--instrumentation`). The HW instrumentation can be generated (it requires FPGA resources) but not used when running the application. The application binary also has to be compiled with this flag.

For example, the previous compilation with the instrumentation enable becomes:

```
$ arm-linux-gnueabihf-fpgacc –instrument –debug –ompss –bitstream-generation <source_code.c> -o <executable_name_debug> –Wf, ”–board=zedboard, –clock=100, –name=<executable_name_debug>, –hwruntime=som”
```

12.3. Binaries

Two Mercurium front-ends for the FPGA devices are available:

- `fpgacc` for C applications;
- `fpgacxx` for C++ applications.

If the user modifies only the host code (i.e. without change the source code issued to the FPGA) and a valid bitstream is already available, it is actually enough to recompile the application without launch the AIT, which is time consuming. This is done recompiling the application without using the `--bitstream-generation` flag. It is worth noticing that the flag `--ompss` must be used otherwise the `#pragma` targeting the FPGA will be ignored, thus the kernel will be run on the CPU.

[27] `https://pm.bsc.es/ftp/ompss/doc/user-guide/compile-programs.html`

1835 ## References

[1] M. Vstias, H. Neto, Trends of cpu, gpu and fpga for high-performance computing, in: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–6.

[2] A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: a proposal for programming heterogeneous multi-core architectures., Parallel Processing Letters 21 (2011) 173–193. doi:10.
1840 1142/S0129626411000151.

[3] E. Calore, S. F. Schifano, Energy-efficiency evaluation of fpgas for floating-point intensive workloads, in: I. T. Foster, G. R. Joubert, L. Kucera, W. E. Nagel, F. J. Peters (Eds.), Parallel Computing: Technology Trends, Proceedings of the International Conference on Parallel Computing, PARCO 2019, Prague, Czech Republic, September 10-13, 2019, Vol. 36 of Advances in Parallel Computing, IOS Press, 2019, pp. 555–564. doi:10.3233/APC200085.
1845 URL https://doi.org/10.3233/APC200085