Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

WP1 Specifications, Co-design & Benchmarking

D1.3 Proof of Concept Design

Part I

WP-1.2: Runtime Services – T.1.2.3: I/O filter

http://textarossa.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

TEXTAROSSA

Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale

Grant Agreement No.: 956831

Deliverable: D1.3 Proof of Concept Design

Part I – T.1.2.3

WP-1.2: Runtime Services – T.1.2.3: I/O filter

Project Start Date: 01/04/2021 **Duration**: 36 months

Coordinator: *AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE - ENEA , Italy.*

Deliverable No	D1.2 – Part I - T.1.5.1-2
WP No:	WP1
WP Leader:	ENEA
Due date:	M6 (November 30, 2021)
Delivery date:	31/05/2022

Dissemination Level:

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

This project has received funding from the European Union's Horizon 2020 research and innovation programme, EuroHPC JU, grant agreement No 956831

DOCUMENT SUMMARY INFORMATION

Project title:	Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale
Short project name:	TEXTAROSSA
Project No:	956831
Call Identifier:	H2020-JTI-EuroHPC-2019-1
Unit:	EuroHPC
Type of Action:	EuroHPC - Research and Innovation Action (RIA)
Start date of the project:	01/04/2021
Duration of the project:	36 months
Project website:	textarossa.eu

WP1 Specifications, Co-design & Benchmarking

Deliverable number:	D1.3					
Deliverable title:	Proof of Concept – Part I – T.1.2.3					
Due date:	M6					
Actual submission date:	31/05/2021					
Editor:	Francesco Iannone					
Authors:	List of Authors					
Work package:	WP1					
Dissemination Level:	Public					
No. pages:						
Authorized (date):	31/03/2022					
Responsible person:	Francesco Iannone and Bérenger Bramas					
Status:	Plan	Draft	Working	Final	Submitted	Approved

Revision history:

Version	Date	Author	Comment
0.1	15/05/2022	F.Iannone, P.Palazzari	Draft structure
0.2			

Quality Control:

Checking process	Who	Date
Checked by internal reviewer	Project Technical Committee	
Checked by Task Leader		
Checked by WP Leader	Francesco Iannone	
Checked by Project Coordinator	Massimo Celino	

COPYRIGHT

ACKNOWLEDGEMENTS

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no 956831. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Italy, Germany, France, Spain, Poland.

Please see http://textarossa.eu for more information on the TEXTAROSSA project.

The partners in the project are AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE (ENEA), FRAUNHOFER GESELLSCHAFT ZUR FOERDERUNG DER ANGEWANDTEN FORSCHUNG E.V. (FHG), CONSORZIO INTERUNIVERSITARIO NAZIONALE PER L'INFORMATICA (CINI), INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), BULL SAS (BULL), E4 COMPUTER ENGINEERING SPA (E4), BARCELONA SUPERCOMPUTING CENTER-CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), INSTYTUT CHEMII BIOORGANICZNEJ POLSKIEJ AKADEMII NAUK (PSNC), ISTITUTO NAZIONALE DI FISICA NUCLEARE (INFN), CONSIGLIO NAZIONALE DELLE RICERCHE (CNR), IN QUATTRO SRL (in4). Linked third parties of CINI are POLITECNICO DI MILANO (CINI-POLIMI), Università di Torino (CINI-UNITO) and Università di Pisa (CINI-UNIPI); linked third party of INRIA is Université de Bordeaux; in-kind third party of ENEA is Consorzio CINECA (CINECA); in-kind third party of BSC is Universitat Politècnica de Catalunya (UPC).

The content of this document is the result of extensive discussions within the TEXTAROSSA © Consortium as a whole.

DISCLAIMER

Table of Contents

Executive Summary

This report shows the co-design process within the WP1 of Textarossa project in order to design Proof of Concept for the Task 1.2.3 regarding I/O interfaces in HPC exascale systems composed of thousands of compute nodes, high performance networks and parallel I/O based on high performance storage systems capable of scaling to terabytes/second of IO bandwidth while providing tens of petabytes of capacity.

This deliverable has the subtitle *Part I T1.2.3*, because it'll be a living document reporting upgrade in several PoC design for developing on the whole stack of the co-design process. In particular this deliverable includes the design of I/O compression and decompression filters on accelerator devices.

Partner Report Activity

Task 1.2.3 TL: INRIA	Runtime Services: to design a PoC able to improve the I/O performances reducing the data movement by means compression and decompression filters. Participants: ENEA
Github address	The software developed and the benchmarks carried out during the activity are downloadable at github at the address: https://gitlab-tex.enea.it
Technology	ENEA HPC CRESCO Data Centre – ENEA FPGA LAB
Technical development	The technical development is performed by ENEA

List of Authors

ENEA	F. Iannone, P. Palazzari

List of Acronyms

AaaS	Accelerator as Service
ABI	Application Binary Interfaces
ACP	Acceleration Coherency Port
ADC	Analog Digital Converter
AFS	Andrew File System
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMG	Algebraic MultiGrid
AMS	Analog Mixed Signal
API	Application Program Interface
ASIC	Application Specific Integrating Circuit
AXI	Advanced eXtensible Interface (Xilinx IP)
BMC	Baseboard Management Controller
C/R	Checkpointing/Restart
CAPI	Common Application Programmer's Interface
CCXI	Cache Coherent Interconnect for Accelerators
CDU	Cooling Distribution Unit
CLB	Configurable Logic Block
CNN	Convolution Neural Network
CP	Common Platform
CPU	Central Processing Unit
CRDB	Co-design Recommended Daughter Board
CU	Compute Unit

CXL	Compute Express Link
DAG	Data-flow Graphs
DC	Direct Cooling
DCL	Data Control Language
DDR	Double Data Rate memory
DIMMs	Dual In-line Memory Modules
DL	Deep Learning
DLC	Direct Liquid Cooling
DPSNN	Distributed Polychronous Spiking Neural
DSL	Domain Specific Language
DSP	Digital Signal Processing
DTPC	Direct Two-Phase Cooling
ECC	Elliptic Curve Cryptography
ECC	Error correction code memory
EDP	Energy Delay Product
ED2P	Energy Delay Square Product
EDS	Embedded Design Suite
EFLOPS	Exa Floating Point Operations per Second
EPAC	EPI Accelerator
EPI	European Processor Initiative
ETS	Energy-To-Solution
FMM	Fast Multipole Method
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSB	Front Side Bus
FT	Fault Tolerance
FTI	Fault Tolerance Interface
GCD	Graphics Compute Die
GIC	Generic Interrupt Controller
gpm	Gallons per minute
GPU	Graphics Processing Unit
GRM	Global Resource Manager
HBA	Host Bus Adapter
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HEP	High Energy Physics
HLL	High-Level Language
HLS	High Level Synthesis
HPC	High Performance Computing
HPDA	High Performance Data Analytics
HPL	High Performance Linpack
HPS	Hard Processor System
HSS	High Speed Serial
HTC	High Throughput Computing
IoT	Internet of Things
IOB	Input/Output block
IP	Intellectual Property
IPMI	Intelligent Platform Management Interface
IR	Iterative Refinement
KPN	Kahn Process Network
L2HN	L2 cache Coherence Home Node
LCM	Last Common Multiple
LE	Logic Element
LRM	Local Resource Manager
MCM	Muti-Chip-Module
MD	Molecular Dynamic
MDS/T	Metadata Server/Target
ML	Machine Learning
MMU	Memory Management Unit
MPI	Message Passing Interface
MPPA	Multi-Purpose Processing Array
MPSoC	Multi-Processor System on Chip
NFIR	Non-linear Finite Impulse Response
NN	Neural Network
NoC	Network on Chip

NVIC	Nested Vectored Interrupt Controller
NVMe	Non-Volatile Memory
OAM	OCP Accelerator Module
OCP	Open Compute Project
QPI	Quick Path Interconnect
OSS/T	Object Storage Servers /Target
PCG	Preconditioned Conjugate Gradient
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PFLOPS	Peta Floating Point Operations per Second
PGMRES	Preconditioned Generalized Minimal Residual
PoC	Proof of Concept
PSU	Power Suppy Unit
PU	Processing Unit
PUE	Power Usage Effectiveness
QCD	Quantum ChromoDynamic
QFDB	Quad FPGA Daughter Board
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
RMS	Resources Management Systems
RoCE	RDMA over Converged Ethernet
RTC	Real Time Clock
RTRM	Runtime Resource Manager
SAS/SATA	Serial Attached SCSI/Serial ATA
SLR	Super Logic Region of FPGA
SoC	System on Chip
SpMM	Sparse Matrix-sparse Matrix
SpMP	Sparse Matrix Power
SpMV	Sparse Matrix-Vector
SSD	Solid State Drive
STX	Stencil/Tensor accelerator
TCO	Total Cost of Ownership
TDAQ	Trigger & Data Acquisition
TDP	Thermal Design Power
TOPS	Tera Operations per Second
TTS	Time-To-Solution
ULFM	User-level Fault Mitigation
ULL	Ultra-Low Latency
UVM	Unified Virtual Memory
VCS	Virtual Compute Server
VM	Virtual Machine
VPU	Vector Processing Unit
VRP	Variable Precision co-processor

1 Introduction

Scientific applications produce massive amounts of data as high-performance computing (HPC) systems are moving toward exascale. The ever-increasing volumes of data are posing challenges for scientists to store, share, analyze, and visualize. Compression algorithms have become a crucial component for data management in scientific workflows. Data reduction enables simulations to output more data without worrying about exceeding storage quotas, and could capture more insights in the simulation. However, due to the complexity and poor performance of I/O and compression libraries as well as parallel file systems, the overall compression and I/O performance varies significantly. Scientific simulations on HPC systems often execute on hundreds of thousands of CPU/GPU cores and generate tens of terabytes of data periodically, such as outputting time- history solutions and checkpoint files every certain number of simulation steps. While data is being transferred from the HPC compute nodes' memory to the storage system, the computation of scientific applications is forced to stall for long periods of time, due to the gap between the computational speed and the I/O bandwidth. This can contribute to a significant amount of total application execution time.

Data reduction techniques, such as compression, transformations and deduplication are straightforward solutions to minimize the energy consumption of storage systems by reducing the amount of storage hardware required to store the same amount of data. However, data reduction itself can consume significant amounts of energy, potentially negating its beneficial effects on energy efficiency. Indeed algorithms with a good compression ratio are also very CPU-intensive, which adds increased processing time. This is where FPGAs come into the picture offloading a CPU from specific tasks, such as compression, encryption, and other functionalities, and the FPGA can be programmed or purpose-built to do specific tasks (compression, in this case) more efficiently and in less time. This way, we can achieve higher compression of data in the same or a shorter amount of time by offloading the compression task to an FPGA, thus providing higher compression and freeing up the CPU for other tasks, thereby accelerating the overall workload. Next-generation bigdata systems will be data-driven heterogeneous architectures that leverage integration of CPU/GPU/FPGA-accelerated compute. By offloading compute-heavy compression tasks to the FPGA, the CPU is freed to perform other tasks, and the IT organization is free to take advantage of the significant savings in performance, power and cooling, and space that result from reducing the total number of systems they have to support.

.

2 FPGA data compression

In cascade data compression, a series of low-level compression building blocks that can be used in combination. Since there are many possible combinations of these building blocks, Cascaded compression can be configured in several different ways. In this specific compression schema, *bitpacking* is one of the most frequently compression block working a bit-level operation for reducing the number of bits required to store each value. However, (de)compression should not come with any additional cost during run time, but should be provided transparently without compromising the overall system performance also in term of energy efficiency. To achieve that, a Proof of Concept (PoC) focus on acceleration of *bitpacking* has been designed on FPGA.

The PoC design try to assess the ease of programming of the Vitis flow, verifying if and how often it must care about the low-level details of the FPGA programming. In the present PoC, the aim is to accelerate through FPGA the compression and processing of images coming from a high-speed video camera. The need originated from nuclear fusion experiments with plasmas magnetically confined, where fast cameras with high-performance CMOS sensor technology are usually adopted for acquiring and storing images of plasma discharges. We had access to a Photron FASTCAM SA4 camera installed on the Proto-Shera (Spherical Plasma for HElicity Relaxation Assessment) experiment [XXX] in ENEA Frascati (Rome) and used in several tokamak experiments [*Lampasi,2016*]. The FASTCAM SA4 camera provides up to 3600 frames/s at 1024x1024 pixel resolution collected on a 12-bit depth image. 12-bit data are packed in 16-bit words, exposing an alignment that eases subsequent processing, but wastes disk space [Fig.1]. The following picture reports a typical plasma discharge image taken with the SA4 camera.

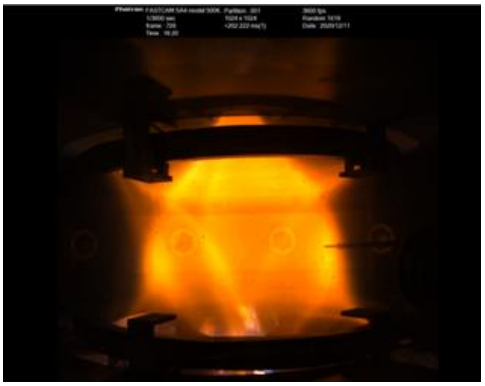


Fig. 1. Plasma discharge image from the FASTCAM SA4 camera

The raw data acquired with a digital camera like SA4 is a measure of the radiant intensity which refers to the magnitude or quantity of light energy actually reflected from or transmitted through the object being imaged by an analogue or digital device. it is the only variable that can be utilized by processing techniques in quantitative scientific experiments. The 12-bit pixel depth, associated to the huge amount of data produced by the SA4 camera, poses a problem of storage versus data integrity. In fact, due to the fact that the smallest addressable unit in a digital processor is a byte, to preserve data information it would be necessary to save each pixel in a two-byte format, thus getting a 25% storage overhead. The alternative could be compression, but this would cause undesirable data loss. In order to save both storage space and data integrity, a bit-packing compression algorithm operating at bit

level, has been developed to reduce the number of bit requested to store 12-bit raw data of SA4 camera. Roughly speaking, the bit-packing algorithm maps two 12-bit pixels to three 8-bit pixels as follow:

Let $A(X)=[x_{ij}]$ a rectangular matrix $n \times 2m$ of n row and $2m$ colunms with X the subset N_r : r-bit natural numbers such as:

$$X \equiv \{ x \in N_r : x \in [0..2^p-1],\ p=12;\ r=16 \}$$

Let us define the *bitpacking* function f transforming the $A(X)$ matrix into the $B(Y)=[y_{ij}]$ rectangular matrix $n \times 3m$ with Y as subset N_q: q-bit natural numbers, such that:

$$Y \equiv \{ y \in N_q : y \in [0..q^p-1],\ q=8 \}$$

The *bitpacking* function f is given on:

$$f:\ \forall x_{i,2j+1}\ ,\ x_{i,2(j+1)} \in A(X)\ (i=1..n;\ j=0..m-1)$$
$$\Downarrow$$
$$Y_{i,3j+1}\ ,\ y_{i,3j+2}\ ,\ y_{i,3(j+1)} \in B(Y)$$

where:

$$y_{i,3j+1} = x_{i,2j+1}\ \&\ (2^q-1)$$
$$y_{i,3j+2} = (\ x_{i,2j+1} >> q\ )\ |\ (x_{i,2(j+1)} << r-p\ )$$
$$y_{i,3(j+1)} = (\ x_{i,2(j+1)} >> r-p\ )\ \&\ (2^q-1)$$

with: $\&$is the AND operator, $/$ is the OR operator and $>>$ $(<<)$ is the RIGHT (LEFT) shift operator.
The inverse function f^{-1}:

$$f^{-1}:\ \forall y_{i,3j+1}\ ,\ y_{i,3j+2}, y_{i,3(j+1)} \in B(Y)\ (i=1..n;\ j=0..m-3)$$
$$\Downarrow$$
$$x_{i,2j+1}\ ,\ x_{i,(j+1)} \in A(X)$$

where:

$$x_{i,2j+1} = y_{i,3j+1}\ |\ (\ (y_{i,3j+2} << q)\ \&\ \ (2^p-2^q)\ )$$
$$x_{i,2(j+1)} = (\ y_{i,3j+2} >> r-p\ )\ |\ (\ y_{i,3(j+1)} << r-p\ )$$

The PoC design uses the HLS Vitis flow to implement also a video processing library which is used to set up a pipeline that, while *bitpacking* compressing video data by removing the heading four bits in every pixel component, processes the video stream through some basic image manipulation blocks properly arranged to form a network of communicating processes.

2.1 The compression kernel

As the original need was to compress the image coming from the SA4 camera, re-moving the four leading bits in the pixel components, the first functionality designed is the compression function which receives through the input stream the input image and produces through the output stream the compressed image. If m and n are the numbers of bits used to store a component of a color $N \times N$ image, the input image has dimension $3mN^2/8$ bytes and the output image $3nN^2/8$ bytes, so the compression ratio is m/n; indicated with S the size, in bit, of the input and output streams of the kernel; S is constrained to be a power of 2. In the beginning, input and output images are aligned with respect to S, i.e., both the input and output streams have a pixel component starting at bit 0 of the input/output; to determine how many components must be read to be aligned again both at the input and the output streams, the smallest number of pixel components $k \in \mathbb{N}+$ satisfying the following equation ($\%$ is the modulus operator)

$$(k \cdot m)\ \%\ S \equiv (k \cdot\ n)\ \%\ S \qquad (1)$$

From (1) it is easy to verify that the solution is:

$$k=LCM(LCM(m,S)/m,LCM(n,S)/n) \qquad (2)$$

Where *LCM(a,b)* returns the Last Common Multiple of the *a* and *b*. In our case the stream size is *S=512* bits (thus saturating the PCIe bandwidth in the reasonable hypothesis to use *fck=300 MHz*), *m=16* bits and *n=12* bits; the expression (2) gives k=*128*. As read from the stream *S* bits, data are newly aligned after $k \cdot m/S=4$ reads from the input stream and $k \cdot n/S=3$ writes to the output stream. From previous computations, the structure of the compression function is in the Algorithm 1, The corresponding Vitis HLS code, in the case *S=512*, *m=16*, *n=12*, is in the code 2:

Algorithm 1: compression kernel	**Algorithm 2:** HLS code

for (i=0; i<InputImageSize/S; i+= km/S){
read km/S words from the input stream;
while (not all the input bits have been copied) {
copy n bits from the input word to the output word;
skip (n-m) bits of the input word
}
write the kn/S words that have just been filled
}
	for (i = 0; i < ImgSize/(S/m)-3; i+=4){
	#pragma HLS pipeline
	di = inStream.read();
	for (int j=0; j<(S/m); j++){
	#pragma HLS unroll
	*do.range((j+1)*12-1,12*j)=di.range(16*(j+1)-5,16*j);}*
	di = inStream.read();
	for (j=0; j<10; j++){
	#pragma HLS unroll
	*do.range(384+(j+1)*12-1,12*j+384)=(di.range(16*(j+1)-5,16*j));}*
	*do.range(511,504)=(di.range(16*10+7,16*10));*
	out_stream << do; //output of the first word
	…//repeat till 4 words have been transmitted
	}

in the previous code, thanks to the unroll pragma, all the assignments from the *S/m=32* input components to the 32 output components are executed in parallel, in the same clock cycle. As we specified the HLS pipeline pragma, the loop body will be pipelined and, every 4 clock cycles, a new execution of the loop body will start, thus resulting in a continuous reading from the input stream of 64 bytes/cycle.

2.2 The flow management kernels

When performing image processing elaborations through streamed kernels that forks one stream to more streams and, conversely, join more streams into one stream. The following kernels have been designed: *splitRGBChannels*, *mergeRGB-Channels*, *streamCopy*, *streamBinaryOperator* (*BinaryOperator = Add, Diff, Mul, Div, Max, Min, Average)*, *maskImages*. Let's review their behavior.

Unlike of the frame images acquired by FASTCAM SA4 in a format raw of a *1024x1024* pixel matrix on 12 bit depth, a general digital image processing works with RGB images, therefore a kernel function *splitRGBChannels* provides to split the color components. It has one input stream, where it receives the input image in the form *r1, g1, b1, r2, g2, b2*, … and three output streams, one for each color component; its counterpart is *mergeRGBChannels* that has three input streams one output stream;

Fig. 2. The *splitRGBChannels* and *mergeRGBChannels* kernels

The *streamCopy* kernel copies data from its input stream to two output streams:

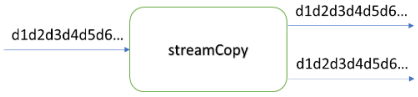


Fig. 2. The *streamCopy* kernel

The *streamBinaryOperator* receives two images from two input streams and pro-duces the output image computed applying the binary operator to each pair of input components.

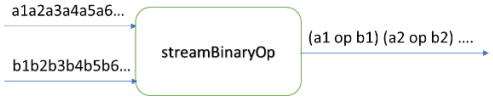


Fig. 4. The *streamBinaryOp* kernel

The *maskImages* receives two images from two input streams and a binary masking image from the mask stream; it produces the output image taking the output pixels from one or the other image, depending on the value of the mask image.

Fig. 5. The *maskImages* kernel

The structure of all previous kernels is quite similar: the kernel reads, at each cycle, *S* bit from each of its input streams and writes the output whenever it is available (only the *mergeRFBChannels* has the opposite behavior, writing *S* output bits at each cycle and reading when possible). A fragment of the HLS code implementing the *streamBinaryOp* kernel is reported in the Algorithm 3.

Algorithm 3: *streamBinaryOp* kernel

```
for (int i = 0; i < (ImgSize*m/S); i++) {
#pragma HLS pipeline
                img1 = inStream1.read();
                img2 = inStream2.read();
                for (int j=0; j<PIXEL_IN_INPUT_WORD; j++){
        #pragma HLS unroll
                        img1Val = (img1.range(m * (j + 1) - 1, m * j);
                        img2Val = (img2.range(m * (j + 1) - 1, m * j);
                        outVal = abs(img1Val-img2Val);
                        outImg.range(m*(j+1)-1,m*j) = outVal;}
                outStream.write(outImg);
        }
```

Due to the HLS pipeline pragma, at each clock cycle *S* bits of *img1* and *img2* are read from the two input streams and all the *S* bits of the output are computed in parallel, thanks to the unroll pragma in the inner loop. As Alveo U280 Board that does not allow streaming from the host to the FPGA card, the design implements 2 kernels that access the external memory; one, *loadInput*, reads data from the external memory and writes them to the output stream and the other, *storeOutput*, reads data from the input streams and writes them to the external memory. Both the streams and the memory ports have width *S*. The interface with external memory uses the AXI4 standard, but this is completely transparent to the user.

Fig. 3. The loadInput and storeOutput kernels, both interfacing with the external memory

the code to implement *loadInput* and *storeOutput* functions is in Algorithm 4 and 5.

Algorithm 4: *loadInput* kernel	**Algorithm 5:** *storeOutput* kernel
for (int i = 0; i < (BitImgSize)/S; i++) *inStream << in1[i];*	*for (int i = 0; i < (BitImgSize)/S; i++)* *out1[i] = out_stream.read();*

In both the cases, at each clock cycle *S* bits are read (written) from (to) memory and written (read) to (from) the output (input) stream

2.3 The elaboration kernels

In addition to the functions defined above, in the image processing library, several functions have been designed. They receive an image through the input stream and produce an output image through the output stream. Among these functions designed:

- *doNegative*: computes the negative of the input image,
- *linearScaling*: perform a linear scaling on each pixel of the input image,
- *RGB2YUV*: converts the input image from the RGB color space to YUV,
- *YUV2RGB*: converts the input image from the YUV color space to RGB,
- *firFilter*: performs the convolution of the input image with a 3x3 filtering kernel.

Apart from the *firFilter*, which has a more complex structure because it needs to read and store three lines before starting the processing, all previous kernels have the same structure which allows, at each cycle, to read *S* bits from the input stream and produce *S* bits to the output stream. The generic kernel has the behavior shown in Algorithm 6 (design, as an example, to the function *doNegative*).

Algorithm 6: *doNegative* kernel

```
for (int i = 0; i < (ImgSize*m/S); i++) {
#pragma HLS pipeline
                img = inStream.read();
                for (int j=0; j < (S/m); j++){
        #pragma HLS unroll
                        imgVal = img.range(m * (j + 1) - 1, m * j);
                        outVal = MAX_Y-imgVal;
                        outImg.range(m*(j+1)-1,m*j) = outVal;}
                outStream.write(outImg);
        }
```

In the previous code, the unrolled inner loop allows the contemporaneous execution of the pixel transformation on all the (*S/m*) values contained in the *S* bits input word; thanks to the HLS pipeline pragma, the pipelined outer loop is started at each clock cycle, so the kernel, at each clock cycle, reads *S* bits from the input stream and writes S bits to the output stream.

2.4 Integration design

Once the kernels have been designed, the desired processing is obtained collecting the kernels on a network that performs the desired elaboration. for instance, the Fig.7 shows a network that takes as input an RGB picture and produces as output an image which resembles a pencil drawing of the input image .

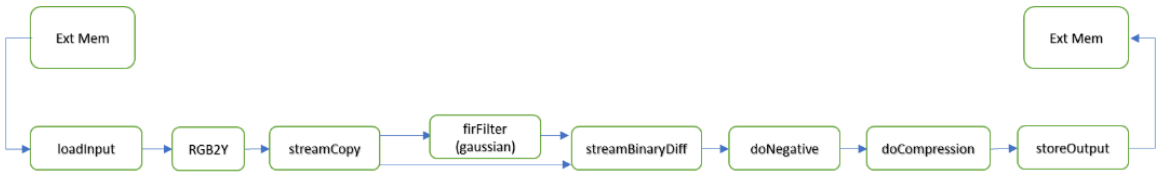
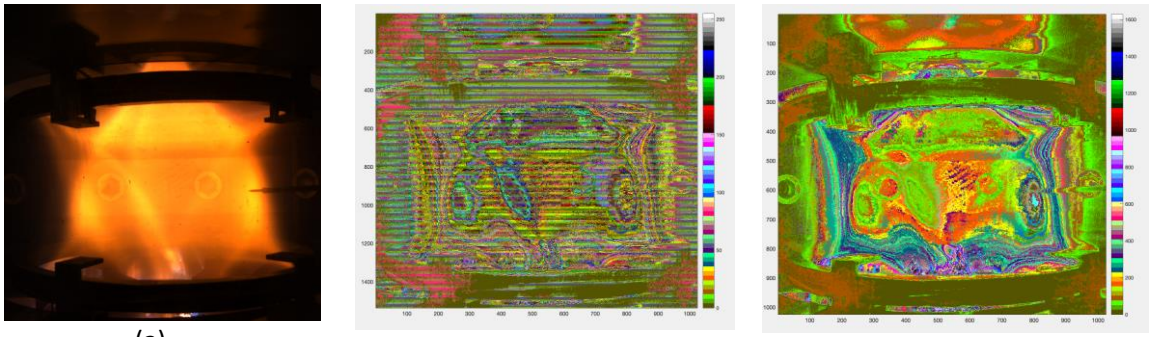


Fig. 7. Example of a network for image processing

The Fig.8 shows an example of image processing produced by FASTCAM S4 camera in a plasma discharge of Protosphera tokamak experiment. In this case the input 1024x1024 16 bit image is compressed by *bitpacking* algorithm for storing in a compressed output 1536x1024 8 bit image or filtered with pencil drawing.



(a) (b) (c)

Fig. 8. Image processing of FASTCAM SA4 camera. (a) Input image; (b) compressed image; (c) processed image

The code needed to implement the previous network is the following Algorithm 7 (we neglect the declaration of the streams).

Algorithm 6: Image processing network

```
#pragma HLS STREAM variable = s1 depth = 128
…
#pragma HLS STREAM variable = s4 depth = 4096
#pragma HLS dataflow
        loadInput(inRAM, s1,ImgSize);
        RGB2Y(s1, s2, ImgSize);
        streamCopy(s2, s3, s4, YImgSize);
        firFilter(s3,s5,Nr,Nc,c00,c01,c02,c10,c11,c12,c20,c21,c22);
        streamBinaryDiff(s3, s5, s6, YImgSize);
        doNegative(s6, s7, YImgSize);
        doCompression(s7, s8, YImageSize);
        storeOutput(outRAM, s8, compressedYImgSize);
```

The HLS dataflow pragma instructs the compiler to activate in parallel all the follow-ing functions, whose execution is purely dataflow; data flow through the streams and they are consumed by the kernels implementing the functions as soon as they are available. Apart from the streams, the other parameters are scalars indicating the size of the images (color image, grey level image, compressed image), the filter coefficients to implement a *3x3* gaussian filter, and the number of rows and columns contained in the input image. The HLS stream pragma is used to specify the size of the FIFO buffers associated with the streams; please note that *s4* streams, which bypasses the *firFilter*, has a larger depth to avoid deadlock. In fact, *s4* must be able to store all the data till *firFilter* starts producing its output: this is because the *streamBinaryDiff* kernel reads in parallel the two input images and does not proceed reading one image till the other is not available. For this reason, the stream *s4* connecting *streamCopy* to the *streamBinaryDiff* must be able to buffer the data coming from *streamCopy* till data do not start to be produced by *firFilter*. In general, all the time that convergent paths are present, particular care has to be taken to analyze and set the buffer sizes to avoid deadlock situations due to the saturation of some intermediate buffer.

2.5 Synchronization scheme

To allow the overlapping of communication between host and card with the computation on the FPGA card, we adopted a double buffering scheme that uses two HBM banks *HBMreadA* and *HBMwriteA* in one phase (A) of the computation and *HBMreadB* and *HBMwriteB* in the other phase (B). The host code has three threads: producer thread (Algorithm 8) that sends images to be processed to an input host buffer accessible to the card, consumer thread (Algorithm 9) that receives processed images from an output host buffer accessible to the card, and the main thread that controls the start of the kernels on the card and the data transfers between HBM memory banks on the card and the buffers on the host.

Algorithm 8: *Producer*

```
while(){
        sem_wait(&canProduceIn)
        write image to Bin
       sem_post(&canConsumeIn)
}
```

Algorithm 9: *Consumer*

```
while(){
    sem_wait(&canConsumeOut)
    read image from Bout sem_post(&canProduceOut)
}
```

The main thread is synchronized with the producer and consumer threads through semaphores (*canProduceIn(1), canConsumeIn(0), canProduceOut(0), can-ConsumeOut(0)* – in brackets we indicate the initial value for the semaphore) which control the access to the input and output buffers, shared with the FPGA card. The synchronization between the main thread and the FPGA card is achieved through *readEvent*, *writeEvent*, and *RunEvent* that inform the main thread about the end of a read transfer, of a write transfer, and of a kernel execution (all data transfers and kernel executions are used as asynchronous, non-blocking functions).

Algorithm 10: Main thread

```
while(){
         sem_wait(&canConsumeIn);
        doTransferHost2Card(Bin, &H2C_event);
       H2C_event.wait();
       startKernels(krnl, &krnl_event);
      sem_post(&canProduceIn);
     krnl_event.wait();
    sem_wait(&canProduceOut);
   doTransferCard2Host(Bout, &C2H_event);
  sem_post(&canConsumeOut);
}
```

In the fragments of code reported in Algorithm 10 is designed the basic synchronization scheme among the producer, consumer, and main threads, for the sake of readability, we do not show the implementation of the ping pong buffer which alternates between the bank pairs HMB0/HBM1 and HBM2/HBM3: when transferring to/from HBM0/HBM1 the kernels are processing images on HBM2/HBM3 and vice-versa.

Algorithm 10: Main thread

```
while(){
         sem_wait(&canConsumeIn);
        doTransferHost2Card(Bin, &H2C_event);
       H2C_event.wait();
       startKernels(krnl, &krnl_event);
      sem_post(&canProduceIn);
     krnl_event.wait();
    sem_wait(&canProduceOut);
   doTransferCard2Host(Bout, &C2H_event);
  sem_post(&canConsumeOut);
}
```

The following Fig.9 shows the graphical report produced by Vitis using the events registered during the actual run on the Xilinx U280 board. As we see, computing kernels are always running and, at the same time, there is always a transfer to/from the external HBM banks (either HBM0/HBM1 or HBM2/HBM3).

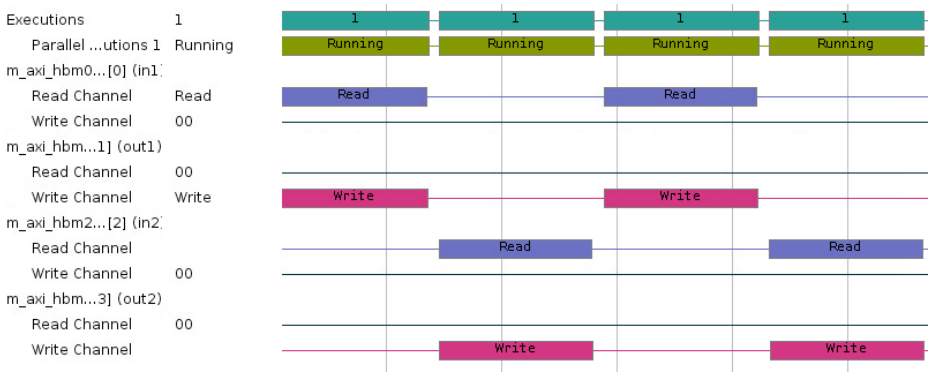


Fig. 9. Overlap of the run of elaboration kernels with I/O on HBM banks (ping pong scheme).

2.6 Programming models

All the image processing/compression library has been designed in C/C++ standard. The only interaction with the HW was defining the width of the streams/memory ports, which can be set through a wizard, the mapping of the I/O buffers on dedicated HBM memory banks (again, done through a wizard), and the definition of the depth for some stream buffers, done through a dedicated pragma. Some optimization pragmas have been spread along with the code, instructing the compiler that certain loops should be pipelined and other loops unrolled, and this would require a certain awareness about the type of architecture we want to obtain (i.e., even working at the C/C++ level, we should not forget that what, and how, you write has a direct impact on the hardware generated).

Let's speak about the feedback we get from the flow to reach our design goal. To check for the functional correctness of the application we are writing, there is a complete Eclipse-based environment that allows debugging our application with the usual tools: breakpoints (several types), memory inspection, access to the debug stack. For the hardware part, we have the automatic instantiation of a multithreaded environment to emulate the concurrency among the different kernels. Once the functional correctness is ensured, we have to compile for the HW (either for HW simulation through the associated HDL simulator or for the actual run on FPGA board). At this level, we can check performance by measuring the time spent to run the accelerated algorithm (we can use the number of clock cycles given by the HDL simulator in HW Emulation). Should we need a more in-depth analysis of the reason why we are not obtaining the expected performance, we found that the analysis of the I/O traffic and of the kernel activation given by the recording of the events during the actual HW run (an example is given in Fig. 8) is useful; sometimes we felt the need of inspecting the HDL simulation to recognize that things were not going as expected (for instance, we were not reading on input streams at full bandwidth), understand why the HW was behaving in that way and go back to the C source code and change it to remove the encountered inefficiency. So, at this level, we found it still quite tricky to remove the inefficiencies and we think that more high-level feedback should be given to guide the optimization process.

Let's analyze this with the case of the 3x3 FIR Filter. Let's suppose that we have implemented the functions *readLine(instream, line),* which reads one line of the input image from the input stream into the local array line, *writeLine(outstream, line),* which writes the local array line into the output stream, and *filterOneLine(il1, il2, il3, ol),* which produces the output line ol from the input lines *il1, il2, il3* through a 3x3 convolution kernel. A simple way to apply the filter to an image is the following Algorithm 11.

The code is scheduled by the Vitis flow as in the Fig.11 (we report the start signals of the functions *S1-S5*, as taken from the HDL simulator of Vitis).

Algorithm 11: filter processing design

```
readLine(inStream,line1);
        writeLine(out_stream, line1);
        readLine(inStream,line2,);
        for (i=2; i<ImgRows; i++)
        {
                S1: readLine(inStream,line3,);
                S2: filterOneLine(line1, line2, line3, lineout);
                S3: writeLine(out_stream, lineout);
                S4: copyLine(line2, line1);
                S5: copyLine(line3, line2);
        }
      writeLine(out_stream, line3, NbParallelInputWords);
```

The Fig.10, the computing kernel *S2* is often inactive. We have to go back to the code and understand why it is scheduled in such a way. If we look at the algorithm, we see that, at each iteration, it receives a new input line and produces a new output line. As the input and output lines cannot be touched during the processing, we should introduce a 4th input line and an additional output line so that, while processing three input lines and writing on the output line *A*, the r*eadLine* can read the 4*th* input line and the *writeLine* can write the previously written line *B*.

Fig.10. Scheduling of functions in the loop body of image filtering code

Furthermore, in order to avoid the *copyLine* operations, we manually unroll the loop by a factor of 4, so that we process (*il1, il2, il3, olA*) while reading *il4* and writing *olB*, then we process (*il2, il3, il4, olB*) while reading *il1* and writing *olA*, then we process (*il3, il4, il1, olA*) while reading *il2* and writing *olB*, then we process (*il4, il1, il2, olB*) while reading *il3* and writing *olA*, then we are again at the original situation and we can close the loop iteration. The main elaboration loop can be organized as in the Algorithm 12 and time diagram shown in Fig.11.

Algorithm 12: Main loop

```
For (i=0; i<LinesToBeProcesssed; i+=4){
                S1: readLine(inStream,line4,);
                S3: filterOneLine (line1, line2, line3, lineoutA);
                S2: writeLine(out_stream, lineoutA);
                S1: readLine (inStream,line1);
                S3: filterOneLine (line2, line3, line4, lineoutB);
                S2: writeLine (out_stream, lineoutB);
                S1: readLine (inStream,line2);
                S3: filterOneLine (line3, line4, line1, lineoutA);
                S2: writeLine (out_stream, lineoutA);
                S1: readLine (inStream,line3);
                S3: filterOneLine (line4, line1, line2, lineoutB);
                S2: writeLine (out_stream, lineoutB);}
```

The previous loop is scheduled by Vitis overlapping, apart from the initial phase that has to be ad hoc managed, the read from the input stream of a new line with the computation of a new output line and the write to the output stream of the just processed line.

Fig. 11. Scheduling of the optimized loop body

To give an idea of the performance achievable through the HLS flow, we defined an image filtering network constituted by the *loadInput*, the cascade of 20 FIR filters, the compression, and the *storeOutput* kernels. Each of these kernels (apart from the compression and the last one, each one producing 75% of the stream size) reads *S=64* Bytes/cycle, resulting in an aggregated throughput of (NFIR_Kernels + 2.5) $\times S \times fck$ [B/s] and a sustained computation speed of NFIR_Kernels $\times$ Nops_FIR $\times$ *fck* [Op/s]. We used NFIR_kernels=20 and *fck*=300MHz; the number of 16 bit operations executed at each clock cycle by a 3x3 pipelined FIR kernel is Nops_FIR=17, so the internal data throughput is 400 GB/s and the sustained speed is 100 GOp/s.

REFERENCE

[*Lampasi,2016*] A. Lampasi at al.: Progress of the Plasma Centerpost for the PROTO-SPHERA Spherical Tokamak, Energies 2016, 9, 508. https://doi.org/10.3390/en9070508

3 Experimental setup design

The PoC of the *bitpacking* data compression and decompression has been designed for developing on the ENEA FPGA LAB in CRESCO Data Centre on premises.

The ENEA CRESCO FPGA LAB (Fig.12) has made available to the TextaROssa a pool of compute nodes equipped as follow:

- n.6 Linux X86_64 nodes with 2 x Intel Xeon Haswell CPU, 128 GB RAM
- n.2 Xilinx U280 + n.2 Xilinx U250
- Sys.Op. Linux Centos 7.4
- Development software tools: Xilinx VITIS & Intel OneAPI

The access to the FPGA Lab resources is available getting an account in ENEAGRID infrastructure following the access rules reported in ENEA CRESCO portal [CRESCO,2021]. The compute nodes equipped with FPGA boards can be access via ssh on front-end node: cresco-in.portici.enea.it and forwarding on to:

cresco-xilinx0/1/2/3/4/5.portici.enea.it

A graphical remote access to compute nodes of the FPGA Lab is available using ENEA F.A.R.O. (Fast Access Remote Objects) using a client ThinLIC on the front-end node: cresco-in-gui.portici.enea.it

For the TextaRossa project are available some Development Operations tools as follow:

A gitlab for source codes and data benchmark repositories [*GITLAB,2021*]

ENEA Staging Storage Sharing system based on *Owncloud* using AFS (Andrew File System), The geographical distributed filesystem of the ENEAGRID infrastructure as backend [*E3S,2021*]

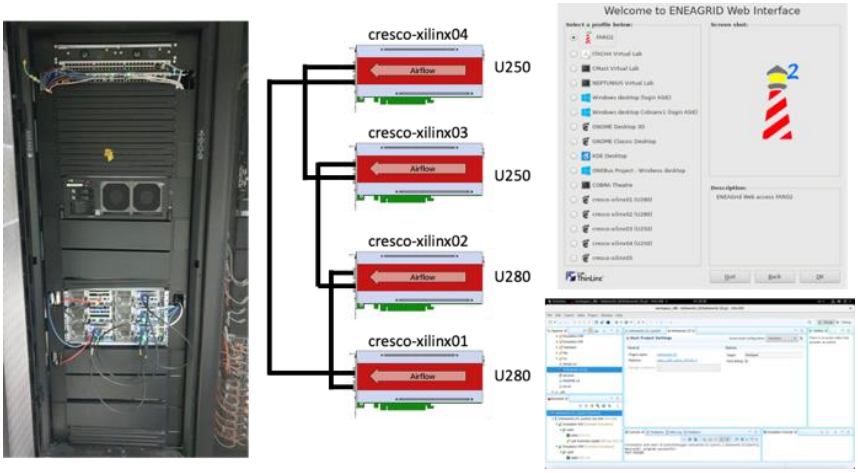


Fig.12: The ENEA FPGA Lab, with F.A.R.O as GUI to develop with VITIS and Intel OneAPI

3.1 Power-Performance metrics

Usually the following four classes of metrics to quantify the power-performance characteristics of a HPC systems [*Feng,2005*].

- *Power*. It is responsible for heat dissipation rate or system operating temperature For a HPC system, power can be defined at various levels of granularity from highest to lowest: HPC system (P_{HPC}), compute node (P_{CN}) and component (P_C). Furthermore the power consumption varies with HPC workload. Since an application uses only a part of the power consumption of the system, we define this part as the power application (P_A), that can be can divided into *idle* and *load* power. The *idle power* is the power consumption under zero workload (i.e., system overhead) and the load power is the increased part of the power consumption when workloads execute on a node. Usually, idle power is a constant while load power varies with time and workload.
- *Energy*. The energy consumed by a HPC system in the time interval $[t_1, t_2]$ is:

$$E_S = \int_{t_1}^{t_2} P_{HPC}\, dt$$

The application Energy is:

$$E_A = \int_0^D P_A\, dt$$

Here *D* is delay which is equivalent to (t_2-t_1) in the Power system equation or TTS (Time-to-solution for the application). Similarly, application energy consumption can be broken into idle part and load part.

- *Performance.* Performance (i.e. reduced TTS) is the critical design constraint in high-performance systems. For fixed workload, speedup can be used to quantify performance comparisons between two alternatives designs or two operating points.
- *Power Performance efficiency*. Sometimes, the performance of HPC systems is improved at the cost of more energy consumption. For example, the number of nodes or operating points used by an application directly affects both energy consumption and TTS; for a fixed problem size on an increasing number of nodes, it is likely that there is some operating point at which point increasing nodes results in largely increased energy consumption with little or no performance gain (i.e. TTS). Therefore, to quantify the power-performance tradeoff of an application on different system configurations, the energy-delay product, $E \cdot D$ and/or energy-delay-square product (ED2P) is used, $E \cdot D^2$ to quantify power-performance efficiency in the context of parallel scalability.

For the power Performance efficiency, there are some specific definitions [*Goz,2019*]. The most obvious one is to compare the TTS to the energy-to-solution (ETS) for a set of runs of both codes and benchmarks. The power performance efficiency can be estimate in terms of Energy Delay Product (EDP). The EDP proposed by Cameron, is a metric to evaluate trade-off between TTS and ETS. The EDP is defined as:

$$EDP = E \times T^w$$

where *E* is the total energy consumed during the run, *T* is TTS and *w* is a parameter to weight performance versus power. Common values of this parameter are *w=1,2,3*. The larger is the *w* exponent the greater the weight we assign to its performance.

3.2 Power analysis design

Power is defined as the amount of electrical energy consumed per unit time, where the unit of power is Joule/second or commonly Watt. There are various techniques to measure and report electric power. For example to measure the instantaneous raw power, average power, peak power, minimum power, root mean square (RMS) power, or a moving average applied to the raw power data. Usually a standard measure in computing systems is the so-called “1-second moving average” that is the form of power analysis thermally relevant for this kind of systems. This averaging provides correlation to real world measurable thermal events. For example, the measure of instantaneous power on millisecond range, useful for other power analysis, detects spikes in power levels for short durations that will not have measurable impacts to the silicon temperature on the heat sink o other thermal solution. Many chipsets provide power sensors with 1 second moving average when report the total compute electric power consumed in watt.

Another common power terms is the Thermal Design Power (TDP). This is the CPU power rating which is typically specified in watts. The TDP ratings refers to the maximum amount of heat generated by the CPU for which the designed cooling system is required to dissipate while running common software. This does not mean that power is strictly limited to the TDP rating. The TDP rating is not the same as “peak possible power” but more like a power rating when running with typical parallel number crunching applications. The TDP specification is also a good baseline number of the wattage power budget needed to run a processor to full performance (100% of utilization). It is possible for processors to consume more than the TDP power specification for a short period of time without it being thermally significant because heat will take same time to propagate, so a short spike in power consumption typically will not violate TDP. To ensure that many CPUs stay within the thermal specifications under such over TDP power consumption, the CPUs have built-in power management hardware which reduces power of the processor by reducing the voltage and/or modulates the processor clock frequency (throttling) until the thermal violation is corrected.

Accurate and timely information regarding power consumption of compute nodes in large scale HPC systems is important in establishing ways to mitigate both the energy consumed and the overall cost.

Monitoring tools enable the measurement of the energy efficiency of a HPC data center focusing on energy modelling, profiling capabilities and upon calibration of models.

There are both hardware and software solutions available today to measure the power consumption of HPC systems while running parallel workloads. One the easiest and most common hardware solutions for measuring total system power is the usage of an AC power meter. This is an instrument that passes power to compute node under load and measures the power consumption in real time. Most of these instruments have ethernet/GPIB/USB interfaces allowing a power data logging by means computer. Another hardware method used to measure system power is the usage of power distribution units (PDUs). These are typically used in data center racks that house HPC systems. The PDU is a smart AC power strip that has an embedded controller that is continually monitoring the total power consumption or load of all compute nodes it supplies with power in a rack including its temperature. These smart PDUs are accessible using a simple web interface or a Simple Network Management Protocol (SNMP) to gather and log the total system power.

On the other hand software tools are able to utilise various data sources such as Baseboard Management Controllers (BMCs) that have various sensors for reporting on the physical hosts. These sensors include measurements for the energy and power consumed of a physical host and are able to report this using the Intelligent Platform Management Interface (IPMI). These sensors are however subject to error and cannot be practically substituted with more accurate Watt meters.

In order to gather sensor data there are two principle sources for monitoring infrastructures to collect data from. The first is reporting data from the operating system, which can utilise special structures such as */proc/* on Linux. The second is to use more specialist hardware such as baseboard management controllers (BMCs) and standardised interfaces.

This can include aspects such as CPU performance counters as well as standardised interfaces such as IPMI. IPMI allows for sensors that are integrated in current generation server hardware to be accessed over a common API. The sensors that have traditionally been used to remotely manage and monitor larger clusters of physical machines and are starting to include power sensing capabilities. The data gathered by these sensors can the be utilised to generate a model, that can calculate the power consumed based on utilisation. Errors in the values reported by the models that drive the energy modeller and Watt meter emulator can occur at two stages. The first is the calibration phase and the second is at operation time.

The calibration phase results in an inaccurate model that does not correctly represent the relationship between load and power consumption. This can occur for several different reasons:

- *Unsynchronized metric update intervals for different metric types*: This could occur when measuring CPU utilisation and power together. For calibration to be accurate it requires the measurements to be perfectly synced or for the utilisation to remain stable during a measurement phase, so that both measurements represent the physical host's true state.
- *Measurement arrival latency (Monitoring infrastructure over-head)*: Differing on the above case, where synchronisation issues may occur, this is caused by the inherent delays in taking a measurement, transferring the value across a network and recording it in the monitoring infrastructure. This effects the detection of the start and end of periods of induced load. This can be mitigated by performing the calibration run locally without the use of a full monitoring infrastructure, such as Zabbix, Ganglia etc. This however will only work during the calibration and will not work during normal operations. Locally monitoring load will however have the side effect of measuring a small amount of load induced by itself.
- *Averaging and time windows of measurement's values*: Measurements arrive with a given polling interval, however measurements such as CPU load also have a time window in which the measurement was taken e.g. over the last minute. This averaging causes errors in the model and requires the CPU utilisation measurement window to be made as small as possible. One alternative is for measurements used in the calibration dataset to only start to be taken after load has been induced for a time that is longer than the length of the averaging period.
- *Update interval* of a sensor's reported value Sensors such as power measurements taken over IPMI update slower than the interval at which the baseboard can be queried. Thus rapid polling of the interface can result in the previously reported value been reported again, without prospect of change. Hence the poll interval should not exceed this update interval. In the case of IPMI power values polling should hence restricted to every 1 seconds.

Therefore this provides the basis of several recommendations which we implement in that should be followed while calibrating an energy model:

- to use metrics that represent the physical host in its most recent state, which we call spot metrics and tend to avoid averaging and representing long periods of time. The sensors base data acquisition of the compute nodes is generally of the millisecond scale, but the firmware of chipset motherboards provides different sampling periods ranging from a minimum of 1 second up to the minutes. Usually the IPMI command interfacing the Data Center Manageability Interface (DCMI) provides electric power measurement of compute node like this:

Linux IPMI Command:
ipmitool -H nodename -U XXXXX -P XXXXX dcmi power reading *with the following output:* *Instantaneous power reading: 89 Watts* *Minimum during sampling period: 87 Watts* *Maximum during sampling period: 89 Watts* *Average power reading over sample period: 88 Watts* *IPMI timestamp: Wed May 18 18:07:25 2022* *Sampling period: 00000001 Seconds.* *Power reading state is: activated*

For example the Lenovo compute nodes, in ENEA FPGA labs, provide an average power reading with a sampling period of 1 second instead the Supermicro compute nodes of CRESCO5F ENEA HPC clusters provide a sampling period of 1 minute. In order to gather electric power measurement of a Supermicro compute node the IPMI command is as follow:

ipmitool raw -H nodename -U XXXXX -P XXXXX 0x30 0xe2 0x00

Unfortunately a complete description of the IPMI tool in raw mode is not available by Supermicro compute node.

- The CPU/GPU/FPGA load should be induced followed by waiting a set period of time for the values to stabilise and then taking measurements. A further addition to this is to detect plateaus in the measured values and only using congruent data points as shown in Fig.1, which can be used as a mechanism to determine how long to wait before accepting measurements as being valid when a compute node is loaded by applications.

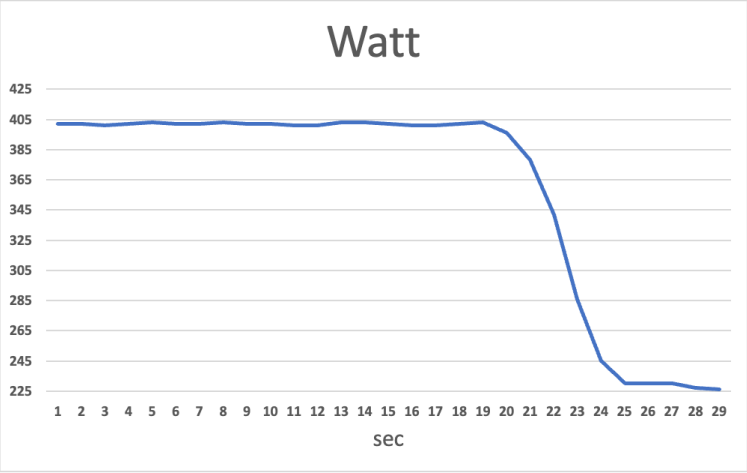


Fig.13: Power monitoring of compute node via IPMI

An alternative technique to collect power consumption induced by applications is to embed in the source code some events probes providing start and end timestamps of the event. each event also reports the identifier of its type (”eventType”), which allows identifying different

types of events for the same application to be analyzed separately. For example a typical json structure is as follow:

Json structure for power monitoring via IPMI

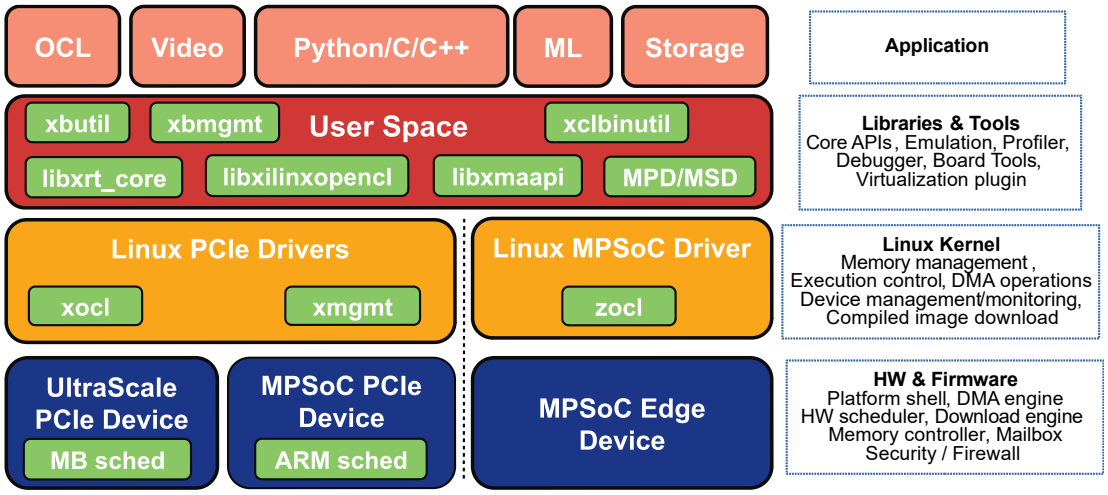
```
{
        "schema version":
        {
                "schema": "JSON",
                "creation_date": "Wed Aug  3 14:37:51 2022 GMT"
        }
        "event":
        {
                "appID": "App 1",
                "eventType": "Event 1",
                "deviceID": "fpga 0000:0b:00.1",
                "starttime:" Wed Aug  3 14:37:51 2022 GMT",
                "endtime:" Wed Aug  3 14:37:55 2022 GMT",
        }
}
```

- to take measurements remotely avoiding intrusive monitoring overloading compute node. Network Time Protocol (NTP) has to be used to sync timestamps between remote and undertest compute node. The sync accuracy in at few msec. The timestamp collected within function kernels in users applications should provide a reliable measure of energy to solution.
- About the granularity in terms of energy consumption of single components hardware it depends on the BMC sensors list that obviously doesn't include the energy consumption of GPU/FPGA processors installed in the node under test.
  The commands like : *nvidia-smi* for GPU NVIDIA as well as *xbutil* for FPGA Xilinx can be run only in intrusive mode consuming CPU power of the node under test.
  In GPU NVIDIA, the NVIDIA System Management Interface (nvidia-smi) is a command line utility, based on top of the NVIDIA Management Library (NVML), intended to aid in the management and monitoring of NVIDIA GPU devices.
  The command provides all runtime metric data of GPU devices installed into the compute node and it works also in virtualized environment based on ESXI and XenServer. It allows the usage for logging with timeout linux command set the queries in a time window. An example of query with 1 second steps on 5 seconds of time window for GPU id 0 is as follow:

```
timeout 5  nvidia-smi -i 0 --query-gpu= timestamp, name, pstate, pcie.link.gen.max, pcie.link.gen.current, temperature.gpu, utilization.gpu, utilization.memory, memory.total,memory. free,memory.used, power.draw --format=csv -l 1
timestamp, name, pstate, pcie.link.gen.max, pcie.link.gen.current, temperature.gpu, utilization.gpu [%], utilization.memory [%], memory.total [MiB], memory.free [MiB], memory.used [MiB], power.draw [W]
2022/08/04 14:39:47.721, NVIDIA A100-PCIE-40GB, P0, 4, 4, 30, 0 %, 0 %, 40960 MiB, 40300 MiB, 53 MiB, 32.82 W
2022/08/04 14:39:48.728, NVIDIA A100-PCIE-40GB, P0, 4, 4, 30, 0 %, 0 %, 40960 MiB, 40300 MiB, 53 MiB, 32.82 W
2022/08/04 14:39:49.733, NVIDIA A100-PCIE-40GB, P0, 4, 4, 30, 0 %, 0 %, 40960 MiB, 40300 MiB, 53 MiB, 32.82 W
2022/08/04 14:39:50.738, NVIDIA A100-PCIE-40GB, P0, 4, 4, 30, 0 %, 0 %, 40960 MiB, 40300 MiB, 53 MiB, 32.92 W
2022/08/04 14:39:51.745, NVIDIA A100-PCIE-40GB, P0, 4, 4, 30, 0 %, 0 %, 40960 MiB, 40300 MiB, 53 MiB, 32.82 W
```

The above command provides several metric data tagged with a timestamps including: temperature in Celsius degree, GPU and memory utilization as percentage, power consumption in Watt.

In FPGA Xilinx Alveo boards the Xilinx Runtime Library (XRT) provides a standardized software interface that facilitates communication between the application code and the accelerated-kernels deployed on the reconfigurable portion of PCIe based Alveo accelerator cards. The stack software of the XRT is as the following picture Fig.14:



XRT Software Stack

Fig.14: ALVEO XRT software stack

The command xbutil examine -d <fpga_device> -r all provides all runtime data of FPGA device, including:

Electrical	***Thermals***
Max Power : 225 Watts	*PCB Top Front : 25 C*
Power : 24.823374 Watts	*PCB Top Rear : 24 C*
Power Warning : false	*FPGA : 29 C*
Power Rails : Voltage Current	*FPGA HBM : 24 C*
12 Volts Auxillary : 12.223 V, 0.857 A	
12 Volts PCI Express : 12.253 V, 1.171 A	
3.3 Volts PCI Express : 3.278 V	
3.3 Volts Auxillary : 3.379 V	
Internal FPGA Vcc : 0.850 V, 11.250 A	
DDR Vpp Bottom : 2.500 V	
DDR Vpp Top : 2.496 V	
5.5 Volts System : 5.458 V	
Vcc 1.2 Volts Top : 1.198 V	
Vcc 1.2 Volts Bottom : 1.199 V	
1.8 Volts Top : 1.795 V	
0.9 Volts Vcc : 0.897 V	
12 Volts SW : 12.166 V	
Mgt Vtt : 1.197 V	

These data can be selected also with *electrical* and *thermals* options of the *xbutil examine* command.

REFERENCE

[*Feng,2005*] Feng, Xizhou & Ge, Rong & Cameron, K.W.. (2005). Power and Energy Profiling of Scientific Applications on Distributed Systems. 34- 34. 10.1109/IPDPS.2005.346.

[*Goz,2019*] Goz, David et al. “Direct N-body application on low-power and energy-efficient parallel architectures.” PARCO (2019). https://doi.org/10.48550/arXiv.1910.14496