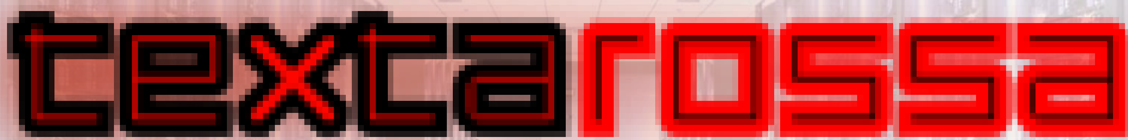
Ref. Ares(2023)2413770 - 03/04/2023

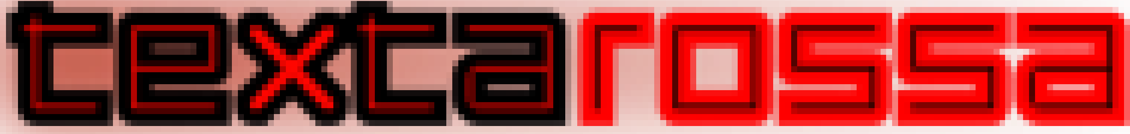
Towards EXtreme scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale



## WP1 Specifications, Co-design & Benchmarking

# D1.4 Benchmarking Design and Planning

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: D1.4 Benchmarking Design and Planning**

**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** | D1.4 |
|---|---|
| **WP No:** | WP1 |
| **WP Leader:** | ENEA |
| **Due date:** | M24 |
| **Delivery date:** | 31/03/2023 |

**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

textarossa

# 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:** | D1.4 | | | | | |
| **Deliverable title:** | Benchmarking Design and Planning | | | | | |
| **Due date:** | M24 | | | | | |
| **Actual submission date:** | 03/04/2023 | | | | | |
| **Editor:** | Xavier Martorell | | | | | |
| **Authors:** | Xavier Martorell, Daniel Jiménez-González, Antonio Filgueras, Carlos Alvarez, Alessandro Lonardo, Cristian Rossi, Francesco Simula, Francesca Lo Cicero, Ottorino Frezza | | | | | |
| **Work package:** | WP1 | | | | | |
| **Dissemination Level:** | Public | | | | | |
| **No. pages:** | 52 | | | | | |
| **Authorized (date):** | 31/03/2023 | | | | | |
| **Responsible person:** | Xavier Martorell and Carlos Álvarez | | | | | |
| **Status:** | Plan | Draft | Working | Final | Submitted | Approved |

**Revision history:**

| Version | Date | Author | Comment |
|---|---|---|---|
| 0.1 | 27/01/2023 | X. Martorell, D. Jiménez, A. Filgueras, C. Álvarez | Draft structure |
| 0.2 | 02/02/2023 | Alessandro Lonardo, Cristian Rossi, Francesco Simula | Power measurements section |
| 0.9 | 14/03/2023 | D. Jiménez, A. Filgueras, C. Álvarez | All remaining sections |
| 1.0 | 24/03/2023 | D. Jiménez, A. Filgueras, C. Álvarez | First integrated and revised version. Candidate release |
| 1.1 | 24/03/2023 | D. Jiménez, A. Filgueras, C. Álvarez | Final formatting check |
| 1.2 | 31/03/2023 | Massimo Celino | Final check and submission on ECAS |

**Quality Control:**

| Checking process | Who | Date |
|---|---|---|
| **Checked by internal reviewer** | Andrea Biagioni | 28/03/2023 |
| **Checked by Task Leader** | Xavier Martorell | 24/03/2023 |
| **Checked by WP Leader** | Francesco Iannone | 26/03/2023 |
| **Checked by Project Coordinator** | Massimo Celino | 31/03/2023 |

## COPYRIGHT

© Copyright by the **TEXTAROSSA** consortium, 2021-2024

This document contains material, which is the copyright of TEXTAROSSA consortium members and the European Commission, and may not be reproduced or copied without permission, except as mandated by the European Commission Grant Agreement No. 956831 for reviewing and dissemination purposes.

## 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

The content of the publication herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

The information contained in this document is provided by the copyright holders "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the members of the TEXTAROSSA collaboration, including the copyright holders, or the European Commission be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of the information contained in this document, even if advised of the possibility of such damage.

## Table of Contents

**Executive Summary** .....6

**Partner Report Activity** .....6

**List of Authors** .....6

**List of Acronyms** .....6

1 Introduction .....9

1.1 Relationship with the objectives of the project .....9

1.2 Organization .....9

2 Hardware Description .....11

2.1 IDV-A and Mitigating Platform (GPU) .....11

2.2 IDV-E and Mitigating Platform (FPGA) .....12

3 Metrics of interest .....13

4 Analisys methodology .....15

4.1 Performance measurements .....15

4.1.1 Performance Measurement on CPUs .....15

4.1.2 Performance Measurement on accelerators (GPUs & FPGAs) .....21

4.2 Power measurements .....25

4.2.1 Power Measurement on CPUs .....25

4.2.2 Power Measurement on GPU .....29

4.2.3 Power Measurement on FPGA .....30

4.3 Accuracy measurements .....38

5 Benchmarking example .....39

5.1 Applications .....39

5.1.1 Matrix multiplication .....39

5.1.2 N- Body .....39

5.1.3 Spectra .....39

5.1.4 Cholesky decomposition .....40

5.2 Results .....40

5.2.1 SMP Results .....40

5.2.2 GPU Platform Results .....42

5.2.3 FPGA Platform Results .....43

6 Conclusions .....46

Appendix A. IDV-E Prototype proof-of-concept .....48

Appendix B. Benchmark codes example .....51

## Executive Summary

This report shows the Benchmarking Design and Planning within the WP1 of the Textarossa project. It provides a reference on how to measure the KPIs of the platform along some benchmark results in the available HPC platforms.

## Partner Report Activity

| | |
|---|---|
| Task 1.2.3<br>TL: INRIA<br>WP6<br>TL: PSNC | This deliverable reports the benchmarking design to obtain the Key Performance Indicators needed by the test activities carried out in WP6 and reported in D6.1, D6.2 and D6.3 |
| Github address | The software developed and the benchmarks carried out during the activity are downloadable at github at the address: |
| Technology | BSC, E4, INFN |
| Technical development | The technical development is performed by BSC, INFN |

## List of Authors



| | |
|---|---|
| BSC | Xavier Martorell, Daniel Jiménez-González, Antonio Filgueras, Carlos Alvarez |
| INFN | Alessandro Lonardo, Cristian Rossi, Francesco Simula, Francesca Lo Cicero, Ottorino Frezza |



## List of Acronyms

| | |
|---|---|
| ABI | Application Binary Interfaces |
| ACP | Acceleration Coherency Port |
| ADC | Analog Digital Converter |
| AI | Artificial Intelligence |
| ALU | Arithmetic Logic Unit |
| 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 |
| CLB | Configurable Logic Block |
| CNN | Convolution Neural Network |
| CP | Common Platform |
| CPU | Central Processing Unit |
| CU | Compute Unit |
| 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 |
| DSL | Domain Specific Language |
| DSP | Digital Signal Processing |
| DTPC | Direct Two-Phase Cooling |
| ECC | Error correction code memory |
| EDP | Energy Delay Product |
| ED2P | Energy Delay Square Product |
| eDRAM | Embedded DRAM (Dynamic Random-Access Memory) |
| EDS | Embedded Design Suite |
| EPAC | EPI Accelerator |
| EPI | European Processor Initiative |
| ETS | Energy-To-Solution |
| FLOP | Ploating Point OPeration |
| FP | Floating Point |
| FPGA | Field Programmable Gate Array |
| FPU | Floating Point Unit |
| FSB | Front Side Bus |
| GPU | Graphics Processing Unit |
| GPGPU | General Purpose Graphics Processing Unit |
| GRM | Global Resource Manager |
| 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 |
| HTC | High Throughput Computing |
| IoT | Internet of Things |
| IOB | Input/Output block |
| IP | Intellectual Property |
| IPMI | Intelligent Platform Management Interface |
| IR | Iterative Refinement |
| KPI | Key Performance Indicator |
| KPN | Kahn Process Network |
| LCM | Last Common Multiple |
| LE | Logic Element |
| LRM | Local Resource Manager |

| MCM | Muti-Chip-Module |
|---|---|
| MD | Molecular Dynamic |
| 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 |
| NVMe | Non-Volatile Memory |
| OAM | OCP Accelerator Module |
| OCP | Open Compute Project |
| QPI | Quick Path Interconnect |
| PCG | Preconditioned Conjugate Gradient |
| PCIe | Peripheral Component Interconnect Express |
| PFLOPS | Peta Floating Point Operations per Second |
| PoC | Proof of Concept |
| PSU | Power Suppy Unit |
| PU | Processing Unit |
| PUE | Power Usage Effectiveness |
| QoS | Quality of Service |
| RAPL | Running Average Power Limit |
| RDMA | Remote Direct Memory Access |
| RISC | Reduced Instruction Set Computer |
| RMS | Resources Management Systems |
| 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 |
| STX | Stencil/Tensor accelerator |
| TCO | Total Cost of Ownership |
| TDP | Thermal Design Power |
| TOPS | Tera Operations per Second |
| TTS | Time-To-Solution |
| ULL | Ultra-Low Latency |
| UVM | Unified Virtual Memory |
| VM | Virtual Machine |
| VPU | Vector Processing Unit |
| W | Watt |

# 1 Introduction

In WP1 the objective is to use a co-design process for developing key technology components able to achieve sizable levels of energy-efficiency in heterogeneous systems for High Computing Performance, AI and HPDA applying using a holistic approach. Usually, a co-design centric process works on the whole stack from underlying HW/SW platforms to the tip of the applications with a process built around identifying specific high-impact applications and providing custom optimization targets.

This deliverable covers the last part of the WP1 objectives: performing benchmarking activities including the design of a benchmarking tool for user applications as well as to perform initial benchmarking of original codes, whereas possible, on reference platforms over the HPC resources available in the project for scalability analysis.

## 1.1 Relationship with the objectives of the project

This deliverable is related to the following project objectives as stated in the DoA:

- Objective 1 - Energy efficiency. This deliverable reports how to measure energy consumption for the IDV platforms. Energy consumption measurement is the first necessary step in order to adapt the runtimes and/or the applications to improve it.
- Objective 2 - Sustained application performance. This deliverable plans how to measure the relevant performance KPIs of the applications. As reported in Section 5 Results, this deliverable also shows some applications performance results obtained with the tools developed in the project that are competitive with the state-of-the-art results for the same platforms.
- Objective 3 - Fine-tuned thermal policies integrated with an innovative cooling technology. As with Objective 1, the energy consumption measurement methods explained in this deliverable are the first step to develop fine-tuned thermal policies.
- Objective 4 - Seamless integration of reconfigurable accelerators. The results obtained for the FPGA platform showed in Section 5 Results are developed with reconfigurable accelerators seamlessly integrated in the system with the OmpSs@FPGA framework. The code of the benchmarks is publicly available and shown in Appendix B for convenience.
- Objective 5 - Development of new IPs. The FPGA platform results showed have been obtained using the hardware IP developed in Task 2.5 and the OmpSs@FPGA framework developed in WP4 across different tasks.
- Objective 6 - Integrated Development Platform. As explained in section 2.2 IDV-E Platform (FPGA) at the moment of finishing this deliverable we have been able to run and test the benchmarks in the final IDV-E Platform. It is important to highlight that although most of the results shown have been obtained using the temporal platform that was available from the beginning of the project, now that the final platform is available the codes and techniques developed in the project can be adapted to the new IDV-E platform in a straightforward way.

## 1.2 Organization

This document is organised as follows. In Section 2, the different hardware platforms used in the project are presented. It also details the hardware architecture being available for testing. Section 3 briefly refers the metrics of interest for the project. Section 4 presents the methodology plan for benchmarking the

applications. In Section 5, there is an example on the results of the benchmarking designed in this deliverable for some example applications. Finally, Section 6 brings some conclusions to the table.

# 2 Hardware Description

In this section the different hardware platforms proposed in the project are described, as well as the current available platforms used to explain the benchmarking design and planning.

## 2.1 IDV-A and Mitigating Platform (GPU)

The mitigating platform used for the experiments is BullSequana XH2000 platform, accessible to Atos partners in several funded projects. In particular only one CRRM blade is used, and then, there is no connection to high-speed interconnect and the only access is a 1Gb/s link of the CPU host.

The CRRM blade is composed of dual AMD EPYC 7402P with 24 cores and 48 threads per CPU. Main memory is laid out as 8 NUMA nodes, with 64GB of memory each. They add up a total of 512GB of main memory. This node also contains 4 nvidia A100 GPU XSM-40GB GPU. They are attached via the nvidia SXM4 socket. This provides PCIe Gen 4 x16 connectivity to the host system as well as 4 nvlink lanes to each of the other GPUs. Each GPU includes 40GB HBM2 local memory. This node is currently accessible via ATOS' Dibona research cluster, located at ATOS' facilities. Figure 2.1.1 shows a high level view of this architecture.

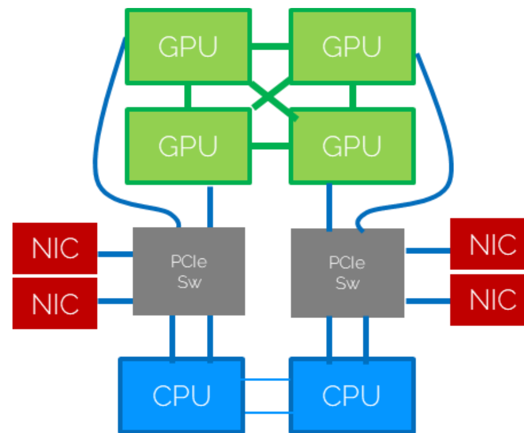



**Figure 2.1.1 GPU blade architecture with PCIe switches**

The architecture of final IDV-A is similar, except that the PCIe switch is embedded in the Infiniband NDR NIC (ConnectX 7 ou CX7), as described in the following figure:

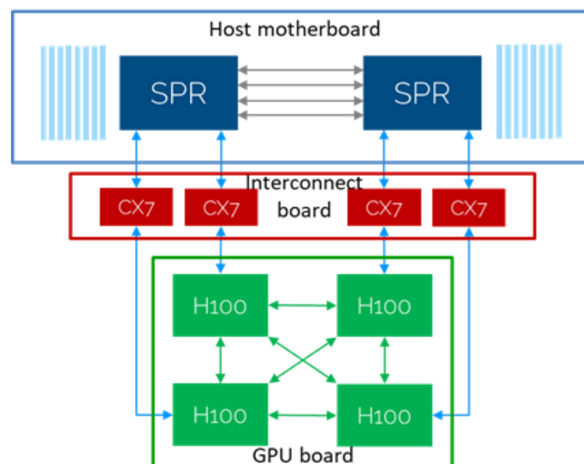



**Figure 2.1.2 GPU blade architecture with embedded PCIe switches**

This architecture change has no impact on the blade performance. Therefore, although the mitigating platform is not exactly the same as the final one, the benchmarking design and planning can be easily ported to the final platform, and we only expect performance improvements due to two main evolutions:

- Nvidia GPU is the next generation Hopper H100, interconnected with Nvlink.
- In the host node, the AMD Rome CPU is replaced with Intel Sapphire Rapids CPU, with 8 DDR5 memory channels @4800 MT/s and 2 x16 PCIe Gen5 slots per socket.

This blade will be hosted on the new BullSequana XH3000 platform. As this blade remains isolated, the node will also be accessed with the 1Gb/s Ethernet link of the host.

## 2.2 IDV-E and Mitigating Platform (FPGA)

The IDV-E platform is composed of a 2U Mt. Collins with 2 Ampere® Altra® Max Series Processor with 128 cores per CPU plus a Xilinx Alveo U280 FPGA. Main memory is distributed among 2 NUMA nodes with 128GB of main memory each. All hardware details are described in D5.2 ARM + FGPA node prototype. The platform is located at E4 premises and it is currently available. However, there has been two issues that have been solved at the moment of finishing and reporting this deliverable:

1. Xilinx doesn't support ARM host processors for its standalone (PCIe connected) family of FPGAs including the Alveo series. The work carried out at the project overcame this issue with a prototype mounted at BSC[1] (described in D4.1 Progress Report on Programming Models and Runtime Systems, section 3.3.1 Proof-of-concept Textarossa IDV-E Test support). This has been solved in the final machine thanks to the collaboration between E4 and BSC partners.
2. There were PCIe communication issues between the ARM processor and the FPGA that had to be debugged to find out which were the hardware configuration problems.

Appendix A describes the proof-of-concept experiments performed on the IDV-E platform. Therefore, as already mentioned, the final IDV-E prototype is now available to perform executions and measurements for applications using FPGAs. However, this deliverable reports the results obtained with a mitigation platform described below due to timing constrains on performing all the experiments again with the functional IDV-E platform using FPGAs. The mitigation platform has very similar characteristics to the final platform so that the experiments presented should be able to be ported to the final IDV-E platform. On the other hand, IDV-E project platform has been used to report performance results for applications only using CPUs.

The mitigation FPGA platform is composed of a Xilinx Alveo U200 (XCU200-FSGD2104) using a custom design flow. This accelerator card is connected to a host system via a PCIe 2.0 x16 link. The host system is a dual Xeon CPU X5680 running at 3.33GHz, with two NUMA nodes, six cores per socket and two threads per core, with 72GB PC3-10600 of main memory.

[1] The specific contents of deliverable 4.1 referred have been included in the Appendix A of this deliverable for convenience.

# 3 Metrics of interest

The metrics of interest of the applications used in the project are diverse as it is the aim of the project to elaborate solutions for a whole set of different problems. In this sense, the metrics of interest of the project can be grouped in three different types of KPIs: computational efficiency KPIs, energy KPIs and accuracy KPIs.

In particular, the metrics we propose are KPIs against the time. This way, Flops oriented applications can measure GFlops/s (probably the most common KPI) while other applications can measure GPairs/s, GEvents/s, Qubits/s etc. A full list of selected KPIs per application can be found in D6.1 Applications and Use cases. For convenience, Table 3.1 reproduces the contents of D6.1 Table 3 listing the individual KPIs of each application. Once the baseline KPIs per second are stablished we propose to measure not absolute values but relative improvements (speedup measured as time improvements achieved to obtain the same KPIs) in order to evaluate the impact of the project in the application performance.

| App name | KPI - computational efficiency | KPI - energy | KPI - accuracy |
|---|---|---|---|
| Smart cities | execution time/speedup on GPU vs. scalability vs. accuracy | Power model on target GPU and on FPGA | Yes |
| Mathlib-CNR | execution time/speedup/strong and weak scalability; number of iterations to a fixed accuracy/time per iteration for iterative solvers | Iterations / W; Dofs / W | Yes (user's parameter dependent) |
| RTM | increase memory bandwidth, then maybe increase Flops | No | No |
| HEP | Events / s | Events / J | No |
| NestGPU | Simulated ms / s | SUPs / J | No |
| RAIDER | Events / s | Events / J | Yes |
| TNM | Qubits / s<br>Gate / s | Qubits / W s<br>Gates / W s | No |
| Mathlib-INRIA | Flops/s \| Interactions/s | Flops / W, Interactions / W | No |
| UrbanAir | iterations/s, simulated time/s | Iterations / W | No |

Table 3.1: Individual KPIs

The KPIs for computational efficiency are:

- Time per iteration, used by iterative solvers where performance can be judged based on how many seconds are needed per each iteration and number of iterations to a user's defined accuracy.
- Simulated time/s (or timesteps/s), used by the solvers which iterates through simulated time, the more timesteps are calculated within one second the better the performance is.
- Interactions/s, used by n-body simulations where the number of interactions between particles is representative of the performance.
- Events/s, used by trigger systems in physics experiments where we refer to "event" as the instantaneous physical situation or occurrence associated with a point in spacetime, characterised in our systems by different information obtained through several physical detectors.

- Qubits/s (and Gate/s), with an equal fixed set of convergence parameters for a quantum simulation with tensor networks method, e.g., fixed bond dimension, the performance can be evaluated looking at what is the size of the system n, in terms of number of qubits, that can be simulated within a second. In some other application, for a given n-qubit system, the performance can be evaluated looking at the number of quantum gates within a second that can be executed.
- FLOP/s, a general performance KPI to indicate how many floating-point operations per second can application achieve.

# 4 Analisys methodology

## 4.1 Performance measurements

As we mentioned, one of the main problems of defining a common framework to measure performance in the context of the project is the wide range of different applications and computing platforms used in the project. In this sense, this section proposes a set of guidelines that can be followed by all the available applications while, at the same time, tries to be flexible enough to 0allow for each application to measure its own KPIs.

As every application relies in a different KPI, measuring the exact KPI is application dependent. However, usually all the KPIs numerators are easily computed from the execution itself. As an example, in a Matrix Multiplication problem, the number of FLOPs computed is always $2xN^3$ where N is the size of the side of the (square) matrix. In an iterative solver, the number of iterations executed is also easily obtained after the solution is found. This fact reduces the problem at measuring the denominator (i.e., time).

In any case, measurements should be reproducible in the sense of:

1) If you want to measure elapsed time, try to run the application within a SLURM queue system that you can reserve a full node to run your application. Otherwise, be sure that you are alone when interactively running with no other applications running at the same time.
2) If your application performs I/O operations and you want to measure the elapsed time dedicated to them, avoid NFS/CIFS file system.
3) Be aware about the energy governors of your OS system. OnDemand or power safe governors may not be the best choices to look for maximum speed up in terms of execution time.
4) Choose representative Input Data for the benchmarking and keep the output results.
5) You may want to use a software for tracking changes.
6) Repeat the measurements and statically analyse them (i.e., T-student).
7) Run the original application in an in-site platform to:
    a) Compile the application with different optimization flags to obtain the best performance that the compiler can achieve (i.e., with gcc, -O1, -O2, -O3, -Ofast, -march=native, etc.).
    b) Obtain the execution time and KPIs of the application for representative Input Data sets
    c) Obtain golden output (if not integrated in the execution of the application)

As already commented above, in addition to the set of different applications there is also a whole set of different platforms where the applications can be executed. In the next subsections, different ways of measuring time performance are detailed for the different targets in the project.

### 4.1.1 Performance Measurement on CPUs

Time measurement on CPUs is an already solved problem. Different solutions exist that can report the time consumed by a CPU while executing a program.

An incomplete list of the most common alternatives follows.

**Shell time command (Linux)**

The easiest and less precise way of measuring time is the Linux time command. Although this method has some disadvantages (the main one being that specific phases inside a program cannot be measured

independently) it also has some strong advantages: it is very easy to use and if the program relies on close-source libraries, may be the only alternative. Figure 4.1.1 shows the output of a very simple program (pi_seq) that computes constant Pi with a Monte-Carlo algorithm that increases the output precision with the number of iterations. The application outputs the number of iterations (in this example this would be the KPI), the Pi constant computed and the time in seconds measured only around the Monte-Carlo kernel. This number can be compared against the real time reported by the Linux time

```
sca0@boada-6:~/new/lab0/openmp$ time ./pi_seq 1000000000
Number pi after 1000000000 iterations = 3.141588211059570
17.276947

real    0m17.280s
user    0m17.270s
sys     0m0.004s
```

**Figure 4.1.1: Shell time command usage example**

The output of Figure 4.1.1 shows the main limitation of the time command. As it can be seen, the "exact" time used (as reported by the more precise in-program measurement) is smaller than the "real" time reported by the command. In applications with large startup overheads (like reading large amounts of data from disk) the difference can hide the improvements done in the actual computational kernel, especially for small datasets used in development.

**GNU time command (/usr/bin/time)**

GNU time command provides more information about the amount of page faults and %CPU used. In addition, you can specify what kind of output format and information you want. Listing 4.1.1 shows two examples of usage of GNU time command. First example keep timing in output.txt and second uses a format to process it later.

```
> /usr/bin/time -o output.txt ./pi_seq 1000000000

Number pi after 1000000000 iterations = 3.141592653589828

Execution time (secs.): 0.685819

> cat output.txt

0.68user 0.00system 0:00.70elapsed 98%CPU (0avgtext+0avgdata 2296maxresident)k

0inputs+8outputs (0major+96minor)pagefaults 0swaps

> /usr/bin/time -f "%e %U %S %P"  -o output.txt ./pi_seq 1000000000

Number pi after 1000000000 iterations = 3.141592653589828

Execution time (secs.): 0.686557

> cat output.txt

0.69 0.68 0.00 98%
```

**Listing 4.1.1: Two examples of time command usage. The first one with timing and the second one formatted for later use.**

**Perf-stat command**

Perf is a powerful profiling tool that include stat functionality. It shows the elapsed execution time of the application (overall) and hardware counter information as cycles, cache misses, branches, branch-misses, etc. The perf-stat tool allows the user to specify the number of runs she/he wants to perform of the application to do a basic statistical analysis. Figure 4.1.2 shows the output of the perf stat when running 3 times a "ls" command in a Linux system. The important thing here is to notice that we can see the main information about cycles, branches and branch-misses, cache misses in different levels and the overall execution time (elapsed time). In addition, the perf tool already performs a possible KPI as IPC (ins per cycle in the Figure). Partial results can be kept with --table option for the number of runs we want to perform (-r option). "-d" option is to show cache memory accesses and misses.

```
Performance counter stats for 'ls' (3 runs):

          4.41 msec task-clock              #    0.631 CPUs utilized            ( +-  3.00% )
             2      context-switches        #  464.008 /sec                     ( +- 16.67% )
             0      cpu-migrations          #    0.000 /sec
            99      page-faults             #   22.968 K/sec                    ( +-  0.58% )
     5,257,885      cycles                  #    1.220 GHz                      ( +-  0.55% )  (84.28%)
     7,322,729      instructions            #    1.39  insn per cycle           ( +-  0.33% )
     1,318,451      branches                #  305.886 M/sec                    ( +-  0.31% )
        32,468      branch-misses           #    2.48% of all branches          ( +-  0.69% )
     1,922,443      L1-dcache-loads         #  446.015 M/sec                    ( +-  0.32% )
        47,296      L1-dcache-load-misses   #    2.47% of all L1-dcache accesses  ( +-  0.31% )
        12,596      LLC-loads               #    2.922 M/sec                    ( +-  2.24% )
           254      LLC-load-misses         #    2.07% of all LL-cache accesses  ( +-116.29% )  (15.72%)

         # Table of individual measurements:
         0.007268 (+0.000278) #
         0.006656 (-0.000333) ##
         0.007045 (+0.000055) #

         # Final result:
         0.006990 +- 0.000179 seconds time elapsed  ( +-  2.56% )
```

**Figure 4.1.2: Perf stat command usage example: perf stat --table -r 3 -d ls**

**Perf profiling**

As mentioned, Perf is profiling tool that can use the hardware counters of the processors to measure CPU time but also cache misses, branch miss predictions, etc. In fact, it can provide this information at user, library and even kernel function level. In addition, if the application is compiled with debug information, it also can provide information at source code (and assembly) line. Overall, it is a powerful tool to detect the hot pots of the code to be optimized, parallelized or accelerated and ported to the GPU or FPGA. In order to know the hardware counters are available at the machine you can run *perf list* command. Be aware that depending on the system you may need sudo permissions or ask the system administrator to set to 1 and 0 the follow Linux files:

```
echo 1 > /proc/sys/kernel/perf_event_paranoid
echo 0 > /proc/sys/kernel/kptr_restrict
```

**Listing 4.1.2 shows an example of usage of perf to profile and visualize the profiling results.**

```
> perf record -e cycles -e cache-misses ./pi.O3.g > /dev/null
[ perf record: Woken up 71 times to write data ]
[ perf record: Captured and wrote 17.739 MB perf.data (387449 samples) ]

> perf report –stdio
# Total Lost Samples: 0
#
# Samples: 250K of event 'cycles'
# Event count (approx.): 41585025326
#
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ...............................
#
   31.70%  pi.O3.g  pi.O3.g          [.] __udivsi3
   27.47%  pi.O3.g  pi.O3.g          [.] SUBTRACT
   27.36%  pi.O3.g  pi.O3.g          [.] DIVIDE
    9.58%  pi.O3.g  pi.O3.g          [.] LONGDIV
    3.54%  pi.O3.g  pi.O3.g          [.] .divsi3_skip_div0_test
    0.08%  pi.O3.g  pi.O3.g          [.] __divsi3
    0.06%  pi.O3.g  libc-2.31.so     [.] memset
    0.02%  pi.O3.g  libc-2.31.so     [.] putchar
    0.01%  pi.O3.g  libc-2.31.so     [.] __vfprintf_internal
    0.01%  pi.O3.g  [kernel.kallsyms]  [k] vfs_write
    0.01%  pi.O3.g  [kernel.kallsyms]  [k] _raw_spin_unlock_irq
    0.01%  pi.O3.g  libc-2.31.so     [.] _IO_file_overflow@@GLIBC_2.4
    0.01%  pi.O3.g  libc-2.31.so     [.] _IO_file_write@@GLIBC_2.4
…
> perf annotate –n –stdio
… // capture of the output of perf-annotate
       0 :   8f8:    ldr     r2, [pc, #200]  ; (9c4 <SUBTRACT+0xd8>)
       0 :   8fa:    add     r2, pc
         :          int j, k;
         :          unsigned q, r, u;
         :          long v;
         :          for( k = N4; k >= 1; k-- )
       1 :   8fc:    ldr     r3, [pc, #200]  ; (9c8 <SUBTRACT+0xdc>)
       0 :   8fe:    ldr     r3, [r2, r3]
       0 :   900:    ldr     r3, [r3, #0]
       0 :   902:    str     r3, [r7, #20]
       2 :   904:    b.n     96a <SUBTRACT+0x7e>
         :          {
         :          if( (x[k] = y[k] - z[k]) < 0 )
    2642 :   906:    ldr     r3, [r7, #20]
    1152 :   908:    ldr     r2, [r7, #8]
     678 :   90a:    add     r3, r2
     724 :   90c:    ldrsb.w r3, [r3]
    1015 :   910:    uxtb    r2, r3
     803 :   912:    ldr     r3, [r7, #20]
    1125 :   914:    ldr     r1, [r7, #4]
     950 :   916:    add     r3, r1
     900 :   918:    ldrsb.w r3, [r3]
    1489 :   91c:    uxtb    r3, r3
    2353 :   91e:    subs    r3, r2, r3
    3685 :   920:    uxtb    r1, r3
    2817 :   922:    ldr     r3, [r7, #20]
    1741 :   924:    ldr     r2, [r7, #12]
    1266 :   926:    add     r3, r2
```

**Listing 4.1.2: Profiling example with perf tool.**

However, sometimes we require to measure the CPU time, cache misses, etc. of a part of the code that does not corresponds to a function or even elapsed time. In this case we should do code instrumentation.

**Linux Timing API**

In order to measure the execution time (elapsed and CPU time) of a part of the code, we may need to instrument the code using, for instance, clock_gettime that provides *ns* precision. Listing 4.1.3 shows an example of timing instrumentation of part of the application.

```
#include <time.h>
...
struct timespec start_wc, start_cpu;  // wall clock and CPU time
if (clock_gettime(CLOCK_MONOTONIC, &start_wc) == -1) exit(1);
if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start_cpu) == -1) exit(1);

// code to measure the elapse and cpu time
...

struct timespec end_wc, end_cpu;
if (clock_gettime(CLOCK_MONOTONIC, &end_wc) == -1) exit(1);
if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end_cpu) == -1) exit(1);
float elapsed_time = (end_wc.tv_sec - start_wc.tv_sec) + (end_wc.tv_nsec - start_wc.tv_nsec)*1e-9;
float cpu_time = (end_cpu.tv_sec - start_cpu.tv_sec) + (end_cpu.tv_nsec - start_cpu.tv_nsec)*1e-9;
```

**Listing 4.1.3: Instrumentation example with clock_gettime to obtain elapsed and CPU time of a part of a code.**

In case we need to measure something that requires access to the hardware counters, PAPI instrumentation can be used.

**PAPI instrumentation**

PAPI (Performance application Programming Interface) allows programmers to access the hardware counters through a library. In order to know which are the hardware counters (and their names) that you can access you should run *papi_avail* command at your machine. Listing 4.1.1 shows the output result of running papi_avail in a Zynq 7000 system (ARM SMP + AMD/Xilinx FPGA).

```
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_TLB_DM  0x80000014  Yes   No   Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  Yes   No   Instruction translation lookaside buffer misses
PAPI_HW_INT  0x80000029  Yes   No   Hardware interrupts
PAPI_BR_MSP  0x8000002e  Yes   No   Conditional branch instructions mispredicted
PAPI_TOT_IIS 0x80000031  Yes   No   Instructions issued
PAPI_TOT_INS 0x80000032  Yes   No   Instructions completed
PAPI_FP_INS  0x80000034  Yes   No   Floating point instructions
PAPI_LD_INS  0x80000035  Yes   No   Load instructions
PAPI_SR_INS  0x80000036  Yes   No   Store instructions
PAPI_BR_INS  0x80000037  Yes   No   Branch instructions
PAPI_VEC_INS 0x80000038  Yes   No   Vector/SIMD instructions (could include integer)
PAPI_TOT_CYC 0x8000003b  Yes   No   Total cycles
PAPI_L1_DCA  0x80000040  Yes   No   Level 1 data cache accesses
```

**Listing 4.1.4: Output rsult of papi_avail.**

As it can be seen in listing 4.1.4, the list of hardware counters that are available (that should be the same accessible by the perf tool) allow the programmer to measure different KPIs and Floating-Point operations per second.

Listing 4.1.5 shows an example of PAPI instrumentation of a code.

```
#include <papi.h>
…

// start PAPI setup
long long counters[NUM_COUNTERS];
int PAPI_events[] = { PAPI_TOT_CYC, PAPI_L1_DCM };

PAPI_library_init(PAPI_VER_CURRENT);
papi_counters = PAPI_num_counters();

if (papi_counters < NUM_COUNTERS) {
   fprintf(stderr,"Error, only %d available counters\n", papi_counters);
   exit(1);
}
// end PAPI setup


// PAPI instrumentation to indicate it should start counting the type of events indicated by PAPI events
status = PAPI_start_counters( PAPI_events, NUM_COUNTERS );
if (status != PAPI_OK) {
   fprintf(stderr,"Error in PAPI_start_counters (code: %d)\n", status);
   exit(1);
}

....
// code to be measured
...

// PAPI instrumentation to indicate it should stop counting and store them into counters
status = PAPI_read_counters( counters, NUM_COUNTERS );
if (status != PAPI_OK) {
   fprintf(stderr,"Error in PAPI_read_counters (code: %d)\n", status);
   exit(1);
}

// Print out the value of the counters counters
printf("\n%lld L1 data cache misses in %lld cycles\n", counters[1], counters[0] );
 …
```

**Listing 4.1.5: Instrumentation example with PAPI to obtain cycles and cache misses.**

**Likwid-perfctr tool**

This tool is a wrapper to provide an end-to-end measurement hiding previous tools. This supports different modes: wrapper (as the name says), stethoscope, timeline (just performing the KIP divided by time), maker API (instrumentation). Additional information can be found at https://github.com/RRZE-HPC/likwid/wiki/likwid-perfctr

## Scalability

When executing and measuring time performance of parallel applications, usually scalability studies are done executing with a different number of threads. There are two usual scenarios to evaluate the scalability of one application:

1. Increase the number of threads with constant problem size: **strong scaling**. In this case the **objective** is to **reduce the execution time**.
2. Increase the number of threads with problem size proportional to the number of threads: **weak scaling**. In this case the **objective** is to **solver larger problems in the same amount of time**.

Assuming the number of threads equal to the number of processors P, then, Figure 4.1.3 shows the relationship between the total problem size, the number of processors and the granularity of work per processor for weak and strong scaling.

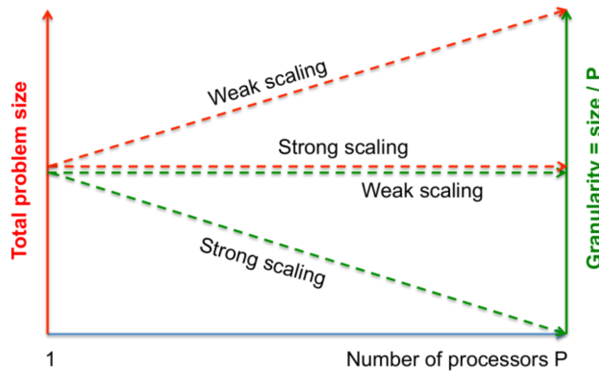



**Figure 4.1.3: Strong vs Weak scaling: Total problem size and Granularity.**

Figure 4.1.4 shows an example of strong and weak scaling for an application (pi computation). For Strong scaling we show speedup and timing analysis (left and center graphs, respectively). As expected, the execution time is reduced when increasing the number of threads. On the other hand, if we compute the speedup achieved in the weak scaling as a ratio of the execution time using one thread and the execution time using P number of threads (each of them with the same problem size), the expected speedup is 1. As notice, the scaling is not perfect and that may be due to parallelization overheads.

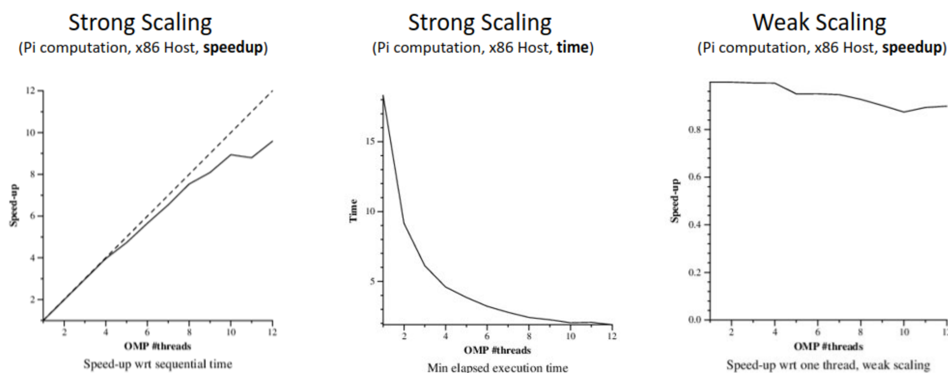



**Figure 4.1.4: Strong vs Weak scaling: Speedup and Time example.**

### 4.1.2 Performance Measurement on accelerators (GPUs & FPGAs)

When measuring time of computations offloaded to accelerators there are two common alternatives:

- Using the CPU time between calls
- Using an accelerator specific method

**Measuring accelerator time from the CPU:**

Although less precise than accelerator specific methods, measuring an accelerator kernel execution time from the CPU is a common practice that delivers precise-enough results.

In the case of the FPGA acceleration, we can do timing instrumentation using clock_gettime and measure elapsed time, as we explained above. It is important to assure that the timing is done under completeness of the acceleration. In the case of OmpSs@FPGA, where acceleration is performed using a task, this is true after a "#pragma omp taskwait". Listing 4.1.6 shows an example of a OmpSs@FPGA program and timing instrumentation of an accelerated task called scale_task. Function scale_task is defined as a task, and in particular for target device FPGA. Any call to this function is transformed at compile time to a runtime library call to create a task with the input and output dependences specified. Indeed, at compile time, the toolchain will create the accelerator for this function using the proprietary tools of the FPGA vendor to generate a bitstream.

In order to execute the program with acceleration, the programmer has to load the bitstream at the FPGA before executing the program. At runtime, this program will create a scale_task task (that should go to the FPGA) and a thread will take care to issue this acceleration to the FPGA accelerator. Taskwait will wait for completeness of the scale_task acceleration to continue. Therefore, to measure the elapsed execution time of this acceleration we only need to instrument our code to from just before calling the function till just after the taskwait.

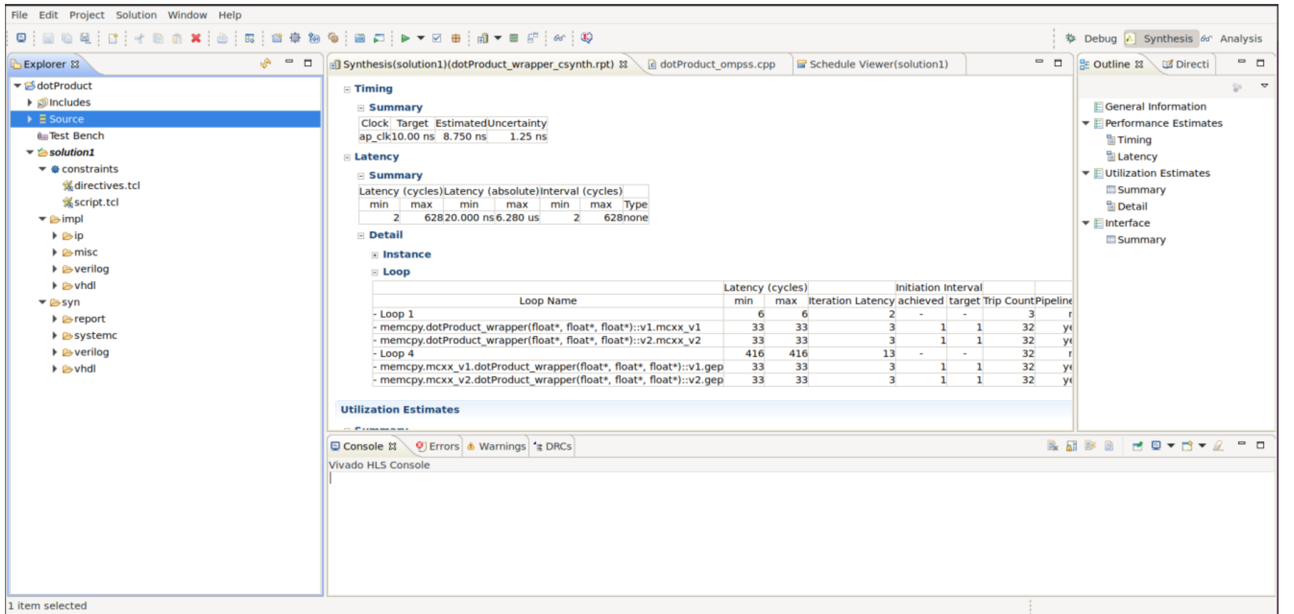
```
const int SIZE=1024;
#pragma omp target device(fpga) copy_deps num_instances(1)
#pragma omp task in([SIZE]c,a) out([SIZE]b)
void scale_task(double *b, double *c, double *a) {
#pragma HLS ARRAY_PARTITION variable=c complete dim=1
#pragma HLS ARRAY_PARTITION variable=b complete dim=1
  double alpha=*a;
  for (int j=0; j < SIZE; j++) b[j] = alpha*c[j];
}

int main (int argc, char *argv[])
{
  // Start timing
  scale_task(B,C,&alpha);
  #pragma omp taskwait
  // End Timing
  ...
}
```

**Listing 4.1.6: Instrumentation example with FPGA acceleration.**

In any case, if the programmer wants to measure the overall execution time and not only the acceleration execution time, she/he also can use any of the methods described above at performance measurements at the CPU, and even perform profiling at the CPU part.

In the case we want to compute any KPI that is related to the number of operations we should consider the CPU equivalent code in order to compute the number of operations and then, divide it by the execution time.

In case we want to know the cycles (and its frequency) of the task accelerator generated the hardware High Level Synthesis (HLS) project can be open and analysed. Figure 4.1.5 shows a general (top) and a detail (bottom) view the automatically generated Vivado HLS project for a dotproduct acceleration. At the general view one can see the global latency (and the estimated frequency) of the dot product acceleration and also the expected latency of copy in and copy out the data to be able to perform the task.



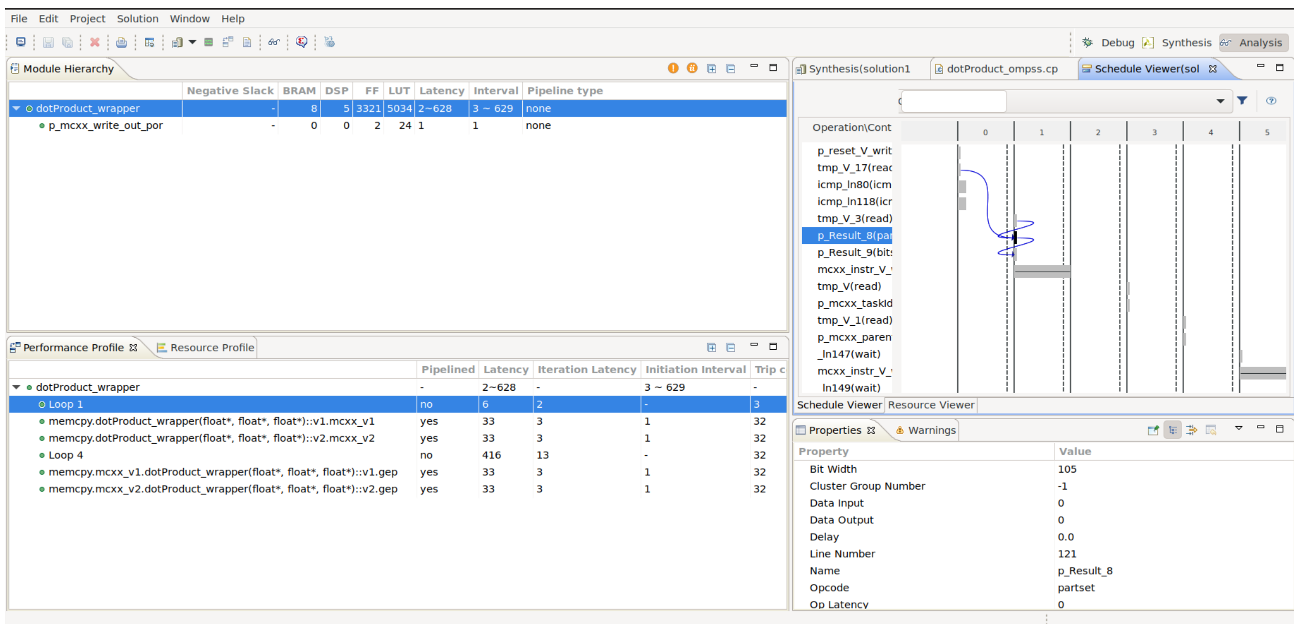




**Figure 4.1.5: General (top) and Detail (bottom) view of the HLS project automatically created.**

In the case of the GPU, and in particular for CUDA programming we can measure the execution time using CUDA events and synchronized with them. Events are enqueued to the device and assure the correct measure of the kernel execution. Listing 4.1.7 shows an example of how to create the events to be able to measure elapsed time for CUDA programs. With this, we can measure execution time of memory transfers from host to device, from device to host and the kernel execution.

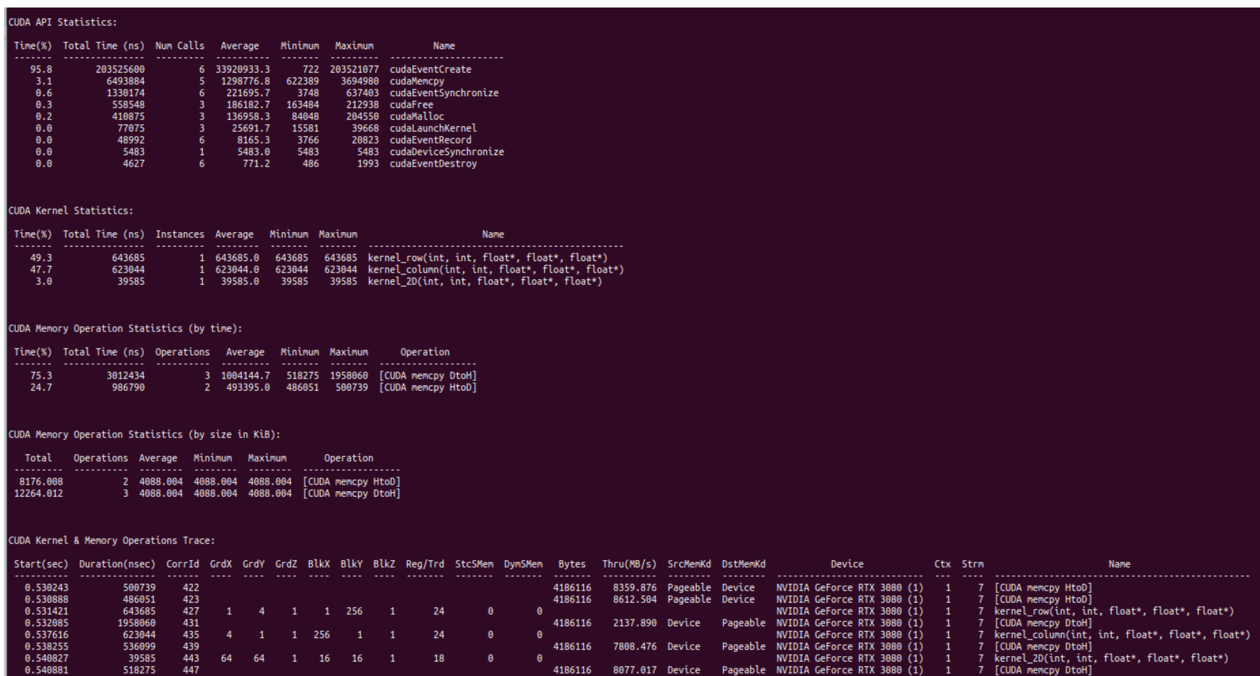
```
cudaEvent_t E0, E1;
…
cudaEventCreate(&E0);
cudaEventCreate(&E1);
…
cudaEventRecord(E0, 0); cudaEventSynchronize(E0);
```

```
...
// Code to be measure that may include kernel calls
//  saxpyP<<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
...
cudaEventRecord(E1, 0); cudaEventSynchronize(E1);
...
cudaEventElapsedTime(&time_measured, E0, E1); // miliseconds
...
```

Listing 4.1.7: Instrumentation example with GPU (CUDA) acceleration.

**Measuring accelerator time with CUDA profiling:**

Nvprof has been the CUDA profiling tool for years. Nowadays, with recent CUDA versions the Nsight Systems command lines help with transition from legacy NVIDIA `nvprof` tool. Figure 4.1.6 shows the output results of a CUDA program when doing profiling with nvprof –print-gpu-trace [program] in legacy systems. On newer systems nsys nvprof --print-gpu-trace [program] must be used to get an API call trace and `nsys profile --stats=true` to display an execution summary. The output shows the API statistics, the kernel execution times, the memory transfers that have been done, and finally, a trace of the memory and kernel execution events, with information of the Grid and Block characteristics of the program.



CUDA API Statistics:

| Time(%) | Total Time (ns) | Num Calls | Average | Minimum | Maximum | Name |
|---|---|---|---|---|---|---|
| 95.8 | 203525600 | 6 | 33920933.3 | 722 | 203521077 | cudaEventCreate |
| 3.1 | 6493884 | 5 | 1298776.8 | 622389 | 3694980 | cudaMemcpy |
| 0.6 | 1330174 | 6 | 221695.7 | 3748 | 637403 | cudaEventSynchronize |
| 0.3 | 558548 | 3 | 186182.7 | 163484 | 212938 | cudaFree |
| 0.2 | 410875 | 3 | 136958.3 | 84048 | 204550 | cudaMalloc |
| 0.0 | 77075 | 3 | 25691.7 | 15581 | 39668 | cudaLaunchKernel |
| 0.0 | 48992 | 6 | 8165.3 | 3766 | 20823 | cudaEventRecord |
| 0.0 | 5483 | 1 | 5483.0 | 5483 | 5483 | cudaDeviceSynchronize |
| 0.0 | 4627 | 6 | 771.2 | 486 | 1993 | cudaEventDestroy |

CUDA Kernel Statistics:

| Time(%) | Total Time (ns) | Instances | Average | Minimum | Maximum | Name |
|---|---|---|---|---|---|---|
| 49.3 | 643685 | 1 | 643685.0 | 643685 | 643685 | kernel_row(int, int, float*, float*, float*) |
| 47.7 | 623044 | 1 | 623044.0 | 623044 | 623044 | kernel_column(int, int, float*, float*, float*) |
| 3.0 | 39585 | 1 | 39585.0 | 39585 | 39585 | kernel_2D(int, int, float*, float*, float*) |

CUDA Memory Operation Statistics (by time):

| Time(%) | Total Time (ns) | Operations | Average | Minimum | Maximum | Operation |
|---|---|---|---|---|---|---|
| 75.3 | 3012434 | 3 | 1004144.7 | 518275 | 1958060 | [CUDA memcpy DtoH] |
| 24.7 | 986790 | 2 | 493395.0 | 486051 | 500739 | [CUDA memcpy HtoD] |

CUDA Memory Operation Statistics (by size in KiB):

| Total | Operations | Average | Minimum | Maximum | Operation |
|---|---|---|---|---|---|
| 8176.008 | 2 | 4088.004 | 4088.004 | 4088.004 | [CUDA memcpy HtoD] |
| 12264.012 | 3 | 4088.004 | 4088.004 | 4088.004 | [CUDA memcpy DtoH] |

CUDA Kernel & Memory Operations Trace:

| Start(sec) | Duration(nsec) | CorrId | GrdX | GrdY | GrdZ | BlkX | BlkY | BlkZ | Reg/Trd | StcSMem | DymSMem | Bytes | Thru(MB/s) | SrcMemKd | DstMemKd | Device | Ctx | Strm | Name |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.530243 | 500739 | 422 | | | | | | | | | | 4186116 | 8359.876 | Pageable | Device | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | [CUDA memcpy HtoD] |
| 0.530888 | 486051 | 423 | | | | | | | | | | 4186116 | 8612.504 | Pageable | Device | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | [CUDA memcpy HtoD] |
| 0.531421 | 643685 | 427 | 1 | 4 | 1 | 1 | 256 | 1 | 24 | 0 | 0 | | | | | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | kernel_row(int, int, float*, float*, float*) |
| 0.532085 | 1958060 | 431 | | | | | | | | | | 4186116 | 2137.890 | Device | Pageable | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | [CUDA memcpy DtoH] |
| 0.537616 | 623044 | 435 | 4 | 1 | 1 | 256 | 1 | 1 | 24 | 0 | 0 | | | | | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | kernel_column(int, int, float*, float*, float*) |
| 0.538255 | 536099 | 439 | | | | | | | | | | 4186116 | 7808.476 | Device | Pageable | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | [CUDA memcpy DtoH] |
| 0.540827 | 39585 | 443 | 64 | 64 | 1 | 16 | 16 | 1 | 18 | 0 | 0 | | | | | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | kernel_2D(int, int, float*, float*, float*) |
| 0.540881 | 518275 | 447 | | | | | | | | | | 4186116 | 8077.017 | Device | Pageable | NVIDIA GeForce RTX 3080 (1) | 1 | 7 | [CUDA memcpy DtoH] |

Figure 4.1.6: Profiling with nvprof a CUDA program.

## 4.2 Power measurements

In this section we describe how to perform power measurements in the different available platforms. When possible different techniques are described in order for the applications to use the one that better fits their specific charactersitics.

### 4.2.1 Power Measurement on CPUs

As described in the Section 2 Hardware Description, Textarossa project will deal with two kinds of CPU architectures: x86_64 (AMD Milan/Rome, Intel Sapphire Rapids) and ARM V8.2 64 bit (AMPERE Altra Max).

**x86_64 Architectures: Running Average Power Limit (RAPL) interface.**

Most modern processors, including Intel processors, provide Running Average Power Limit (RAPL) interfaces for reporting the accumulated energy consumption of various power domains of the CPU chip, attached DRAM and on-chip GPU. The update interval of the RAPL energy counters is approximately one millisecond.

The RAPL energy reporting feature has been available for many generations on Intel SoC products, and energy reporting is standard practice for the industry. Intel processors utilize this energy information for internal SoC management purposes, such as control of SoC power limits in association with Intel® Turbo Boost Technology power limit settings within the SoC.

This RAPL energy data is exposed to the platform via the host-software-accessible model specific registers (MSRs) such as MSR_PKG_Energy_Status and MSR_PP0_Energy_Status. This allows software to use the RAPL energy data for observation, telemetry, and/or inputs to platform-level power or thermal control algorithms [1].

RAPL readings are highly correlated with plug power, promisingly accurate enough and have negligible performance overhead. Experimental results suggest RAPL can be a very useful tool to measure and monitor the energy consumption of servers without deploying any complex power meters [2]. In addition, RAPL supports multiple power domains. The RAPL power domain is a physically meaningful domain (e.g., Processor Package, DRAM etc) for power management. Figure 4.2.1 illustrates the hierarchy of the power domains graphically.

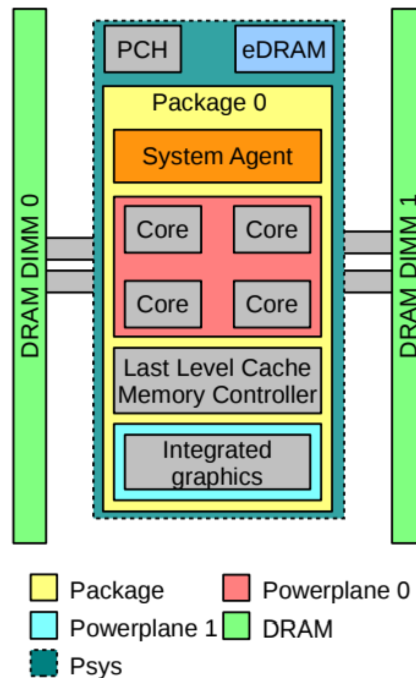



**Figure 4.2.1: Power domains supported by RAPL [3]**

Each power domain informs the energy consumption of the domain, allows to limit the power consumption of that domain over a specified time window, monitors the performance impact of the power limit and provides other useful information, that is, energy measurement units, minimum or maximum power supported by the domain [3].

RAPL provides the following power domains for both measuring and limiting energy consumption:

- Package: Package (PKG) domain measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components (last level caches, memory controller).
- Power Plane 0: Power Plane 0 (PP0) domain measures the energy consumption of all processor cores on the socket. RAPL does not support measuring the power consumption of individual CPU cores.
- Power Plane 1: Power Plane 1 (PP1) domain measures the energy consumption of processor graphics (GPU) on the socket (desktop models only).
- DRAM: DRAM domain measures the energy consumption of random-access memory (RAM) attached to the integrated memory controller. Deviations up to 20% from actual measurements have been reported for this domain, with a strong dependence on the specific processor architecture [4].
- PSys: Intel Skylake has introduced a new RAPL Domain named PSys. It monitors and controls the thermal and power specifications of the entire SoC and it is useful especially when the source of the power consumption is neither the CPU nor the GPU.

As Figure 4.2.1 suggests, PSys includes the power consumption of the package domain, System Agent, PCH, eDRAM and a few more domains on a single socket SoC. For multi-socket server systems, each socket reports its own RAPL values (for example a 2-socket computing system has two separate PKG readings for both the packages, two separate PP0 readings, etc). The support for different power domains varies

according to the processor model, as energy unit used: the Sandy Bridge uses energy units of 15.3 microjoules, whereas Haswell and Skylake uses units of 61 microjoules.

RAPL measurements are accurate, the correlation coefficient between RAPL and plug AC power values has been measured using the Stream benchmark on a Haswell processor, resulting in a value of 0.99 [3].

The RAPL features described above are also available for AMD processors (from family 17h).

Linux supports RAPL since from kernel 3.14, access to RAPL data is possible through several mechanisms, such as reading files under /sys/class/powercap/intel-rapl/intel-rapl:0, using the perf_event interface (e.g., sudo perf stat -a -e "power/energy-cores/" executable) or using raw-access to the underlying MSR registers provided by the msr kernel module.

Several energy profiling tools using the RAPL infrastructure are currently available, we selected likwid-powermeter as reference power measuring tool for CPU tasks. `likwid-powermeter` is part of the Likwid toolsuite [5] of command line applications and a library for performance-oriented programmers. It works for Intel, AMD, ARMv8 and POWER9 processors on the Linux operating system. There is additional support for Nvidia GPUs.

To perform simple end-to-end measurements on a custom application, another lightweight command line tool in the LIKWID toolsuite: `likwid-perfctr`. This tool can be used as a wrapper to your application, in this way the power measuring can be pinned to a precise process to be profiled. A simple example of usage of this tool it is reported below:

```
likwid-perfctr -C 0 -g ENERGY -t 0.1s -O -o out.csv ./executable
```

where, via specific flags, we are requiring a power measuring (-g ENERGY) on an application bound to be executed on a single CPU core (-C 0), with a sampling rate of 100 ms (-t 0.1s) and with a .csv output file (-O -o out.csv).

### AMPERE Altra Max (ARM V8.2 64 bit)

According to the technical documentation provided by the manufacturer, this implementation of the ARM V8.2 64-bit architecture does not provide any RAPL-like facility for fine-grained power measurement.

There are four high power domains for Altra / AltraMax processors:

- PCP power domain for CPU cores and mesh interconnects
- SoC power domain for SoC blocks, memory and PCIe controllers
- RCA power domain for PCIe/CCIX controllers
- DDR4 power domain for memory IOs and DIMMs

The Altra Max Processor Complex (PCP) features include:

- 128 Arm v8.2+ 64-bit CPU cores at up to 3.00 GHz maximum
- 64 KB L1 I-cache, 64 KB L1 D-cache per core
- 1 MB L2 cache per core
- 16 MB System Level Cache (SLC)
- 2x full-width (128b) SIMD
- Coherent Mesh Interconnect (CMI):

Only PCP and SoC power domains are accessible using the Linux HWMON infrastructure, either reading the corresponding `/sys/class/hwmon/hwmon0/*` entries of the filesystem or using the sensors command.

An alternative method is to use the BMC infrastructure, that provides the following power data:

- PCP power domain for CPU cores and mesh interconnects
- SoC power domain for SoC blocks, memory and PCIe controllers
- DDR4 power domain for memory IOs and DIMMs

The IDV-E nodes at E4 (tcnode13 and tcnode14) premises are powered by a Managed PDU that can provide power measurements. Measurements can be performed on the login node only (tlnode1), using the scripts located in the /opt/share/scripts/powerdiscovery directory. Two scripts are available:

```
./getpower.sh <HOSTNAME> <CAPTURE TIME>
```

that provides a timeseries for required node with timestamp [seconds since 1970-01-01 00:00:00 UTC], total power [Watt] and apparent power [VA]. The output of the getpower command is shown in Listings 4.2.1.

```
[alonardo@tlnode01 powerdiscovery]$ ./getpower.sh tcnode14 60
time,watts,va
1670842235,179,191
1670842236,178,190
1670842238,178,190
1670842239,179,192
1670842240,178,191
…
```

**Listing 4.2.1: getpower command output example**

And a second script reporting the instant power:

```
./get_instant_power.sh <HOSTNAME>
```

The output of the get_instant_power command is shown in Listings 4.2.2.

```
alonardo@tlnode01 powerdiscovery]$ ./get_instant_power.sh tcnode14
178 Watts
191 VA
```

**Listing 4.2.2: get_instant_power command output example**

A finer grained power measurement is available using the sensors command, that report I/O and CPU power for each of the two processors. The output of the sensors command is shown in Listings 4.2.3.

```
[alonardo@tcnode14 ~]$ sensors
apm_xgene-isa-0000
Adapter: ISA adapter
SoC Temperature:  +38.0 C
CPU power:        11.76 W
IO power:         17.03 W

i350bb-pci-20400
Adapter: PCI adapter
loc1:         +50.0 C  (high = +120.0 C, crit = +110.0 C)
```

```
apm_xgene-isa-0000
Adapter: ISA adapter
SoC Temperature:  +38.0 C
CPU power:        13.08 W
IO power:         20.03 W
```

**Listing 4.2.3: sensors command output example**

### 4.2.2 Power Measurement on GPU

In this section we consider only NVIDIA GPU in the context of the TEXTAROSSA project.

#### NVML Library

The NVIDIA Management Library (NVML) is a C-based programmatic interface for monitoring and managing various states within NVIDIA GPU devices. It is the underlying library for the NVIDIA-supported nvidia-smi tool. NVML is thread-safe, allowing simultaneous NVML calls from multiple threads. Figure 4.2.2 shows the NVIDIA vGPU Software server interfaces for GPU management.

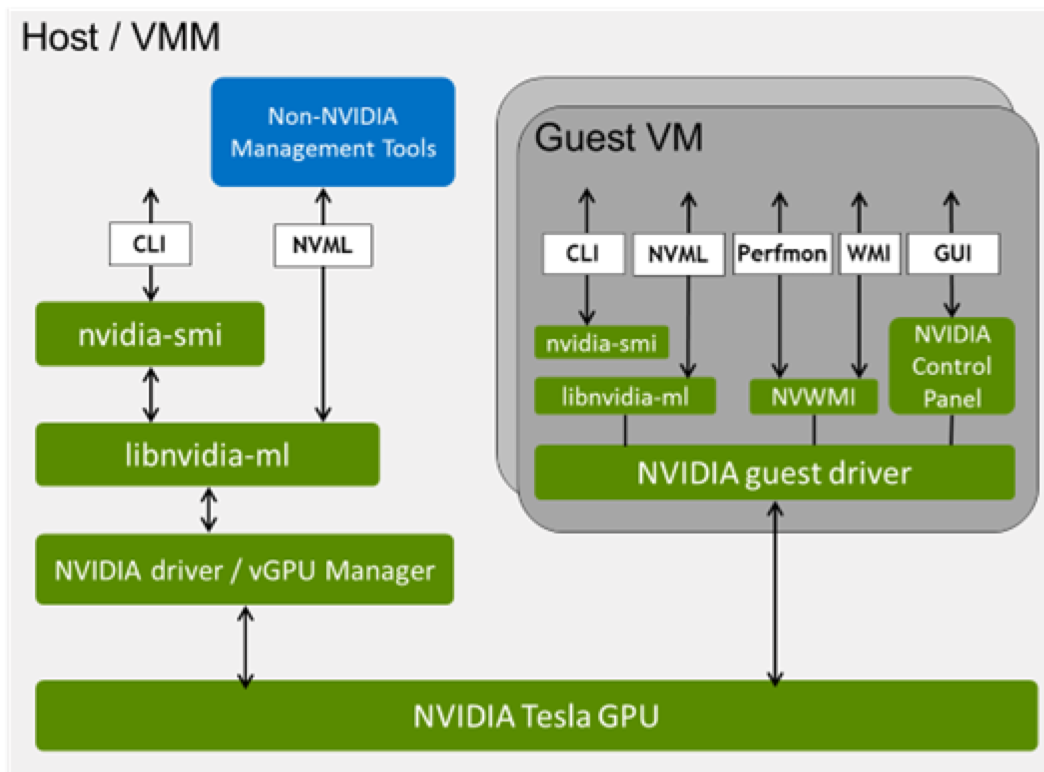



**Figure 4.2.2: NVIDIA vGPU Software server interfaces for GPU management [5]**

NVML is delivered in the NVIDIA vGPU software Management SDK, which enables third party applications to monitor and control NVIDIA physical and virtual GPUS that are running on virtualization hosts. INFN is experimenting with a simple tool, called "GPowerU" (available here: https://github.com/crrossi/GPowerU), able to measure the power consumption of a CUDA kernel in specific points of the device code. Figure 4.2.3 shows the output of the tool. Since the NVML APIs can be

called only by the host side, the idea behind the tool is to send a «message» to the CPU from the GPU to take the power value at specific locations of the CUDA kernel in during the execution (blue points in Figure 4.2.3).

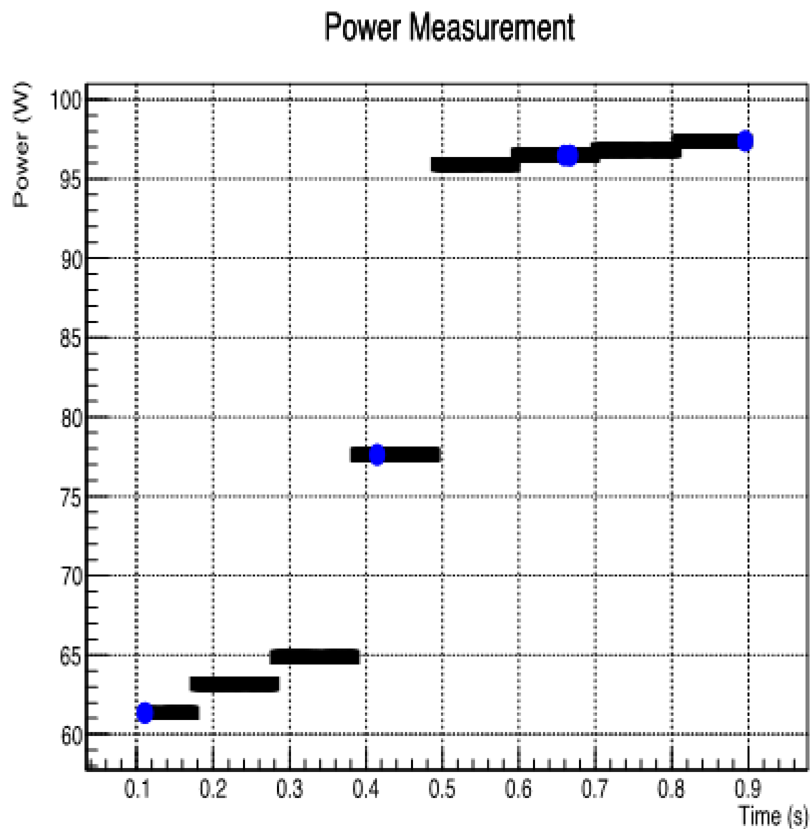



**Figure 4.2.3: Plot with NVIDIA GPU power measurements from INFN**

In addition to this, it is also possible to obtain (through another CPU thread) the overall profile of the power consumption (represented by the black points in the graph in Figure 4.2.3).

However, the power value obtained through the NVML APIs is updated every ~20 ms. Thus, this sampling interval is not suited for a precise evaluation of the power consumption profile of a CUDA kernel with a short execution time. And even if some solution to this power monitoring problem does exist (it is possible to do a loop monitoring starting the data taking with a variable delay), this issue cannot be easily avoided in the case of the power measure in specific kernel checkpoints, since the required GPU-CPU communication unavoidable latency is O(10 us).

### 4.2.3 Power Measurement on FPGA

Referring to FPGAs' power monitoring, two different initiatives have been developed. POLIMI developed a methodology to deploy online power monitors into generic hardware design, capable of periodic power estimate.
They evaluated 2 possibilities for implementing power monitoring:

- Software power monitor: applications can provide online power monitoring if platform RTL description is not accessible, at the cost of a non-negligible performance overhead, low accuracy and limited temporal resolution for the power estimate
- Hardware power monitor: dedicated hardware delivers highly accurate power estimates at high temporal resolution and without performance overhead at the cost of modifying the RTL description of the computing platform.

In this context, POLIMI, UNIPI and INFN have started working together to characterize the power consumption of the IPs developed in TEXTAROSSA project. More information about this methodology can be found in Deliverable 4.4 Power modeling tool suite [6].

The second initiative carried out by BSC integrates FPGA power monitoring into the OmpSs@FPGA environment and it's available in its GitHub public page [12]. This power monitoring tool measures OmpSs@FPGA design power usage in real time while being the first step into managing power usage by the runtime. Its implementation is described in more detail in the OmpSs@FPGA Power Monitoring subsection that follows.

Finally, Xilinx power monitoring tools could be used to obtain an estimation of the power consumed by a design. Although the precision of the estimation is not as high as in the previous methods, it is a good first indicator of the design efficiency and can be used to drive the accelerator design or even to improve its efficiency using Xilinx built-in tools. This approach is described in the next Xilinx Power Monitoring Tools subsection.

### OmpSs@FPGA Power Monitoring

OmpSs@FPGA provides a set of software tools and hardware infrastructure to allow users to get real time power usage and temperature statistics. More information can also be found on D4.4 Power modeling tool suite.

In the hardware side, a Card Management Subsystem is integrated along with OmpSs shell static logic and user accelerators. This module communicates with a satellite controller in the FPGA card and makes power readings available via a homogeneous memory mapped interface. Communication with satellite controller is done via UART and GPIO interfaces, which may be different across different FPGA boards.

A diagram of the power related hardware infrastructure is shown in Figure 4.2.4.

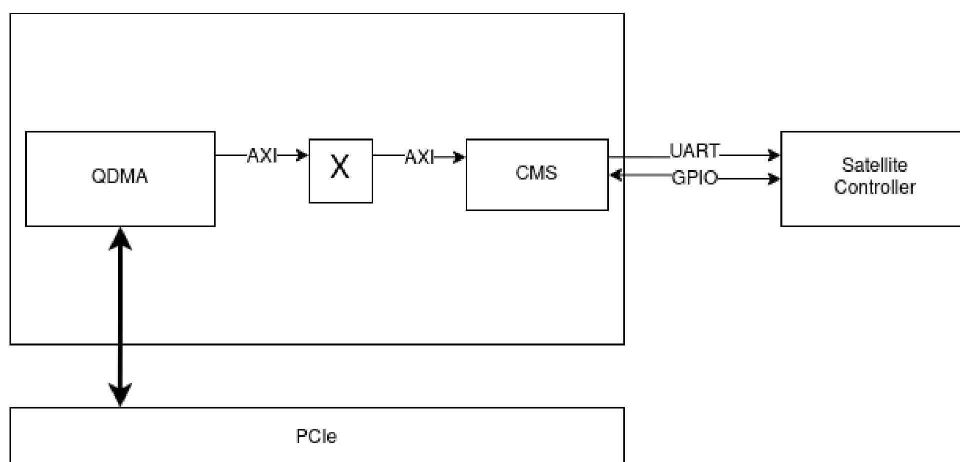



**Figure 4.2.4: FPGA power related hardware infrastructure diagram**

Software power monitoring tools access CMS power statistics by accessing a known memory mapped I/O region inside the PCIe management BAR.

Concurrently to user requests, CMS module polls the satellite controller using UART and GPIO interfaces in order to keep statistics updated. Finally, the satellite controller directly interfaces power delivery components and sensors upon request.

Software wise, we developed a set of applications and libraries that allow users to programmatically query energy measurements either by using an API in the target application or by getting real time power measurements using an independent application.

Listing 4.2.4 shows a sample output from the standalone application.

```
time,12V_PEX_iv,12V_PEX_ii,12V_PEX_cip,12V_PEX_av,12V_PEX_ai,12V_PEX_cap
,3V3_PEX_iv,3V3_PEX_ii,3V3_PEX_cip,3V3_PEX_av,3V3_PEX_ai,3V3_PEX_cap,12V
_AUX_iv,12V_AUX_ii,12V_AUX_cip,12V_AUX_av,12V_AUX_ai,12V_AUX_cap,3V3_AUX
_iv,3V3_AUX_ii,3V3_AUX_cip,3V3_AUX_av,3V3_AUX_ai,3V3_AUX_cap,cap_total,c
ip_total,12V_PEX_ap,12V_PEX_ip,3V3_PEX_ap,3V3_PEX_ip,VCCINT_ap,VCCIN_ip
1678277530.387455,12147,1538,18.682087,12172,1471,17.905012,3364,0,0.000
000,3369,0,0.000000,12176,1427,17.375153,12199,1432,17.468967,3330,0,0.0
00000,3322,0,0.000000,35.373978,36.057240,0,0,0,0,0,0
1678277530.637653,12172,1345,16.371340,12170,1467,17.853390,3361,0,0.000
000,3366,0,0.000000,12211,1497,18.279867,12201,1451,17.703651,3331,0,0.0
00000,3339,0,0.000000,35.557041,34.651207,0,0,0,0,0,0
1678277530.887843,12180,1388,16.905840,12174,1451,17.664474,3366,0,0.000
000,3367,0,0.000000,12207,1449,17.687943,12203,1451,17.706553,3329,0,0.0
00000,3312,0,0.000000,35.371025,34.593781,0,0,0,0,0,0
...
```

**Listing 4.2.4: Sample power output from a standalone application**

This contains the raw readings from all power related sensors in an easy to parse format, which allows data to be easily analyzed and shown using off-the-shelf data analysis tools. This includes instantaneous, immediate and peak voltage, current and power for the different power delivery rails. Note that some of them are 0 as they may not be used in current board, but current infrastructure supports reding power information from different boards.

Figure 4.2.5 shows a plot of the different power supply rails during the execution of an application executing a small matrix multiplication design (only 6 256x256 accelerators performing each 64 multiplications per cycle).

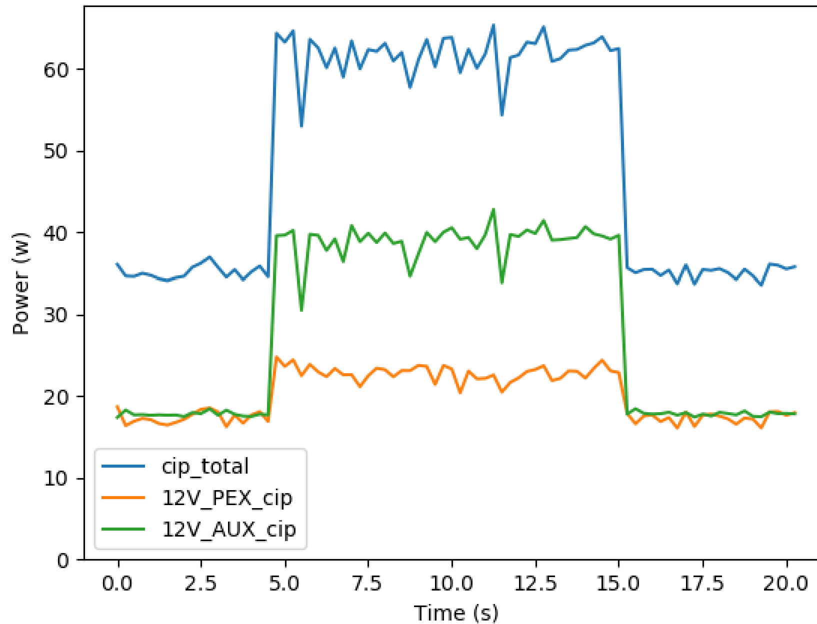



**Figure 4.2.5: Plot from different power supply rails from Alveo U200 board**

It shows instantaneous power over time. More precisely, it shows 12_PEX, and 12_AUX, which are the two power inputs of the board being tested, which is an Alveo u200 accelerator card.

Application execution can clearly be seen. Around second 5, a matrix multiplication starts. This increases power consumption due to the increase of switching activity in the FPGA needed to perform computations.

The same data shown by the application output shown in Listing 4.2.4 and Figure 4.2.4 is accessible through a C API so users can carry custom measurements tailored to a particular application. This allows users to read power related data in strategic places or reset average and peak counters and read them after the application has completed a particular phase.

Listing 4.2.5 specifies the API defined to use power monitoring tools.

`cms_start_power_monitor(uintptr_t cms_address)`

Sets up communications with Card Management Subsystem and resets it to a known state. `cms_address` is the address in which CMS register file is mapped inside PCIe BAR.

`cms_stop_power_monitor()`

Stops CMS and frees all resources associated with power measurement.

`cms_reset_power_monitor()`

Resets average and peak voltage, redings.

`cms_power_monitor_read_values(power_info_t *info)`

```
Reads all power related hardware counters.
```

Listing 4.2.5: OmpSs@FPGA power monitoring API

A `power_info_t` struct is passed by reference. It contains instant and average voltage, current, and power for each of the power inputs as well as total power.

The different rails are: 12V_AUX, 12V_PEX, 3V3_AUX, 3V3_PEX

The structure is defined as shown in Listing 4.2.6.

```
typedef struct power_info_t {

    // Total Power = 12V_AUX Power + 12V_
    float computed_instant_power_total;
    float computed_average_power_total;

    float computed_instant_power_12V_AUX;
    float computed_average_power_12V_AUX;
    uint32_t instant_voltage_12V_AUX;
    uint32_t instant_current_12V_AUX;
    uint32_t average_voltage_12V_AUX;
    uint32_t average_current_12V_AUX;
    …
}
```

Listing 4.2.6: OmpSs@FPGA power information API data structure

In addition to power measurements provided by OmpSs@FPGA, Xilinx power estimation tools can be used to estimate design power in early design stages.

### Detailed example use case

Energy measurement methodology proposes two different use cases or workflows. One of them being a power monitor style application and the other one being an API so a given application can get detailed power statistics.

Power monitor dumps power statistics, this includes power, current and voltage for every power delivery rail in the card, and various health statistics such as temperatures of the various components of the card (FPGA, Memory, QSFP transceivers) and fan speed. This allows easy setup of a system monitoring daemon, which can be a useful log and analyze statistics of an FPGA cluster or implement health monitoring and alert tools. Also provides a user to easily monitor health and energy usage when device is idle and no application is running and analyze how executing a workload affects overall statistics.

Power monitor usage is summarized as shown in listing 4.2.7:

```
power_monitor CMS_address [update_interval]
```

Listing 4.2.7: Power monitor usage

`CMS_address` is the address in which the power monitoring subsystem is mapped. In future releases, this will be automatically determined without user intervention. Update_interval specifies the polling interval between measurements. Note that the internal management subsystem updates statistics every 140ms

approximately [cms manual ref], therefore, specifying an interval below this may result in repeated readings. This is in line with behavior shown by other power and sensor monitoring tools such as NVidia's `nvml` or Linux's sensors.

Once launched, the application outputs readings for all available sensors in a machine-readable format so that further processing and analysis can be carried out by a user.

Listing 4.2.8 shows a complete usage example of the power monitor tool.

```
$ power_monitor 0x40000 0.250
Starting power monitor
device open /sys/bus/pci/devices//0000:02:00.0/resource2
mapping memory
Software profile: 0x0
Enabling CMS...
Using _baseAddr=40000
Power monitor started
time,12V_PEX_iv,12V_PEX_ii,12V_PEX_cip,12V_PEX_av,12V_PEX_ai,12V_PEX_cap,3V3_PEX
_iv,3V3_PEX_ii,3V3_PEX_cip,3V3_PEX_av,3V3_PEX_ai,3V3_PEX_cap,12V_AUX_iv,12V_AUX_
ii,12V_AUX_cip,12V_AUX_av,12V_AUX_ai,12V_AUX_cap,3V3_AUX_iv,3V3_AUX_ii,3V3_AUX_c
ip,3V3_AUX_av,3V3_AUX_ai,3V3_AUX_cap,cap_total,cip_total,12V_PEX_ap,12V_PEX_ip,3
V3_PEX_ap,3V3_PEX_ip,VCCINT_ap,VCCIN_ip
1679577462.404118,12168,1451,17.655767,12157,1473,17.907261,3362,0,0.000000,3363
,0,0.000000,12218,1778,21.723604,12212,1764,21.541967,3324,0,0.000000,3329,0,0.0
00000,39.449226,39.379372,0,0,0,0,0,0
1679577462.654288,12178,1508,18.364424,12167,1495,18.189665,3366,0,0.000000,3362
,0,0.000000,12197,1760,21.466721,12209,1767,21.573303,3321,0,0.000000,3326,0,0.0
00000,39.762970,39.831146,0,0,0,0,0,0
1679577462.904479,12206,1487,18.150322,12179,1494,18.195427,3362,0,0.000000,3363
,0,0.000000,12206,1771,21.616825,12208,1768,21.583744,3324,0,0.000000,3338,0,0.0
00000,39.779171,39.767147,0,0,0,0,0,0
1679577463.154675,12191,1482,18.067062,12181,1473,17.942614,3359,0,0.000000,3362
,0,0.000000,12207,1752,21.386663,12210,1759,21.477390,3327,0,0.000000,3334,0,0.0
00000,39.420006,39.453728,0,0,0,0,0,0
```

**Listing 4.2.8: Power monitor usage example**

Also, a simple plotting tool has been developed along with the power monitor one in order to visualize output data. This tool allows a user to plot readings from a selected sensor list.

Embedded usage instructions are shown un Listing 4.2.9.

```
usage: plot_power [-h] input_file [fields [fields ...]]

Plot power measurements

positional arguments:
  input_file  Power readings file to plot
  fields      Fields to plot

optional arguments:
  -h, --help  show this help message and exit
```

**Listing 4.2.9: Plot usage instructions**

Listing 4.2.10 shows a usage example of the plotting tool. It shows how to plot aggregated inmediate (`cip_total`) and average (`cap_total`) from `matmul_power_data.csv` file.

```
plot_power matmul_power_data.csv cip_total cap_total
```

Listing 4.2.10: Plot tool usage example

Resulting plot is shown in Figure 4.2.6, which shows results for command specified in Listing 4.1.10.

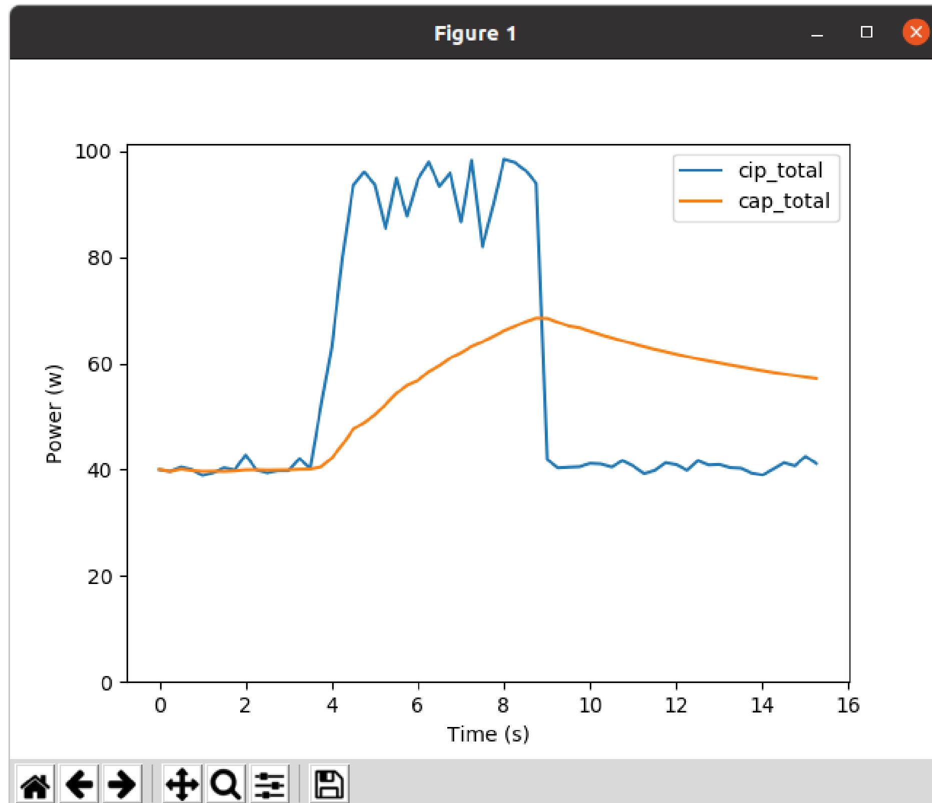



Figure 4.2.6: Power inmediate and average power plot example

Along with the power monitoring tool described above, a C API has been developed in order to programmatically access power statistics and obtain fine grained power statistics. API calls are described in Listing 4.2.5.

One of the use cases of the API is to query average power consumed by an accelerated part of an application. Listing 4.2.11 shows an example that illustrates this use case.

```
power_info_t powerInfo;
cms_start_power_monitor(0x40000);
cms_reset_power_monitor();

application_kernel();

cms_power_monitor_read_values(&powerInfo);
printf("Average power %f\n", powerInfo.computed_average_power_total);
```

Listing 4.2.11: Example usage of the power monitoring API

In this example, the power monitoring subsystem is initialized, then, average values are reset. After that, the application kernel is executed, and power statistics are read and printed. power_info_t data structure

contains data for all available statistics. Therefore, this flow remains the same regardles of the value to be measured.

### Xilinx Power Monitoring Tools

In addition to this, Xilinx®provides a suite of software tools capable of evaluating power supply requirements of the device throughout each stage of the design cycle. Some of the tools are standalone while others are integrated into the implementation software, to align with the environment and information available to you at each stage.

The available tools are:

- **Xilinx Power Estimator (XPE):** is a power estimation tool typically used in the pre-design and pre-implementation phases of a project. The XPE interface lets you specify design resource usage, activity rates, I/O loading, and many other factors which XPE then combines with the device models to calculate the estimated power distribution. XPE is also commonly used later in the design cycle during implementation and power closure to, for example, evaluate power implications of engineering change orders (ECO)
- **Vivado Power Analysis:** this feature performs power analysis through the stages of: post-synthesis, post-placement, and post-routing. It is most accurate at post-route because it can read the exact logic and routing resources from the implemented design
- **Vivado Power Optimization:** Vivado® design tools offer a variety of power optimizations to minimize dynamic power consumption by up to 30% in your design. These optimizations use the equivalent techniques of a complex ASIC clock gating to minimize switching activity without affecting the design functionality

Also, Xilinx Runtime library (XRT) Linux kernel driver xclmgmt binds to management physical function and handles the access to in-band sensors (temperature, voltage, current, power etc.). In fact, it is possible to enable profiling and capturing event trace data during the execution of a custom application, consuming additional device resources to track the host and kernel execution steps. However, this process requires modifying the host application and HLS kernels code, and, most important, it requires configuring the **xrt.ini** File to capture data at runtime.

In detail, specifying the power profiling using the *power_profile* option in the **xrt.ini** file, it is possible to generate the **power_profile_<device>.csv** report [7].

As an example, listing 4.2.12 reports the content of one power profile report, where all the data fields are reported (timestamp is in ms, currents in mA and voltages in mV).

```
Target device: xilinx_u200_gen3x16_xdma_base_1

timestamp,12v_aux_curr,12v_aux_vol,12v_pex_curr,12v_pex_vol,vccint_curr,vcci
nt_vol,3v3_pex_curr,3v3_pex_vol,cage_temp0,cage_temp1,cage_temp2,cage_temp3,
dimm_temp0,dimm_temp1,dimm_temp2,dimm_temp3,fan_temp,fpga_temp,hbm_temp,se98
_temp0,se98_temp1,se98_temp2,vccint_temp,fan_rpm

141.921,1055,12260,1175,12142,10644,851,0,3334,0,0,0,0,26,29,31,28,35,36,0,3
4,28,34,39,1083,

162.441,1055,12260,1175,12142,10644,851,0,3334,0,0,0,0,26,29,31,28,35,36,0,3
4,28,34,39,1083,
```

```
…
```

**Listing 4.2.12: Xilinx power profile contents**

The vitis_analyzer tools can be used to plot power measurements (in Watts) for 3 different available domains (Internal, 12V PCIe, 12V Auxiliary) reading the profile reported data as shown in Figure 4.2.7. To assess power consumption in µW given the above-described information, the following expression is evaluated:

```
P = (12v_aux_vol*12v_aux_curr+12v_pex_vol*12v_pex_curr+3v3_pex_vol*3v3_pex_curr)
```

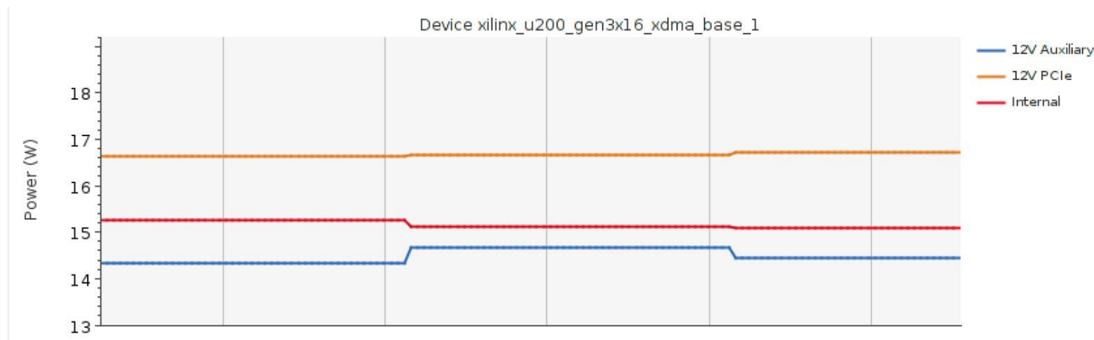



**Figure 4.2.7: Xilinx power plot measurements**

# 4.3 Accuracy measurements

Accuracy measurements are one of the Key Performance Indicators selected in D6.1 Evaluation Plan. Accuracy measurements can refer either to accuracy in detection and classification for the target application (i.e. accuracy = number of the correct predictions divided by total number of predictions) vs. computational complexity and vs. used arithmetic; or accuracy in iterative linear solvers (i.e., number of correct digits in the solution, as required by users). In some applications of the latter case when using mixed precision, the accuracy in the method doesn't affect the accuracy of the final solution but the number of iterations needed to reach this final solution.

As a highly application-dependent measurement, we think that it is not practical to provide a common framework to measure accuracy influence on each application. In order to obtain accuracy and mixed precision results, however, it would be worth to define different accuracies that are useful for the program specific domain and measure the other different KPIs (i.e., performance and power KPIs) and relate them to a specific final accuracy obtained. D6.2 Initial Application Benchmarks and Results will report for each application the specific results obtained.

# 5 Benchmarking example

To evaluate how our benchmarking design works out we have performed the evaluation over 4 different applications in the different platforms available for the project. This section described the results obtained.

## 5.1 Applications

This section describes briefly the applications used to show the usefulness of this deliverable. Additional descriptions can be found elsewhere in the literature [7, 8].

### 5.1.1 Matrix multiplication

Matrix multiplication is a well-known embarrassingly parallel application. The application computes C = C + A × B, being A, B and C matrices of size N × N . We have chosen a matrix multiplication as it is both well-known application and serves as a building block for a wide range of applications. In order to accelerate execution, matrices are divided into square blocks of size BS × BS. This block multiplication is implemented as a task that will be offloaded to an accelerator (either in SMP, GPU or FPGA). Therefore, the multiplication of a large matrix results in several block multiplications. Multiplications in this block accelerator are parallelized to perform multiple parallel multiplications and additions per cycle. The KPI of the matrix multiplication algorithm is (billion) floating point operations per second or GFlops/s. The number of operations performed being $n$ the length of the square matrix side is defined by the following expression: $2n^3$.

### 5.1.2 N- Body

The N-body simulation computes how a group of particles with different masses interact with each other due to gravitational forces over a period of time. Algorithm input is a set of particles, each of one consisting of an initial position, mass and initial velocity. Position and velocity are 3-dimensional single precision floating point vectors, while mass is a scalar value. The output of the algorithm is the set of particles with their positions updated due to gravitational interactions after a given amount of time steps. Each time step consists of two different phases. First, force acting on each particle is calculated. This implies calculating the forces for each pair of particles, which has a cost proportional to $n^2$ being n the number of particles in the system. Then, the position and velocity of each particle is updated using the calculated forces during the time interval that each time step represents using the Euler method. This phase has a cost proportional to n, where n is the number of particles. The KPI of the N-Body algorithm is the number of particle pairs computed in a given unit of time, also referred as billions of pairs per seconf ot GPairs/s. Given a problem consisting of a simulation of $p$ particles during $s$ steps, the number of performed pair computations is defined by the following expression. $p^2 \cdot s$

### 5.1.3 Spectra

The Spectra application [8] computes a histogram of electronic weights between particles versus distance for a given set of particles. To do so, it needs to compute the distance between each pair of particles and then add their electronic weight to the histogram. The histogram is afterwards used to compute the X-ray spectra of the physical material being analysed. The input of the algorithm is a set of particles consisting in 3-dimensional particle position and electric charge stored as single precision floating point numbers. The output is a histogram of the electronic weight according to the distance of the particles with a given number of buckets. As with the N-Body algorithm, the KPI of the Spectra algorithm is the number of particle pairs computed per unit of time or GPairs/s, although the main reason is that a Particle-to-Particle operation

(Pair) doesn't involve exactly the same operations as in the N-Body case. Given a problem consisting of $p$ particles, the total amount of operations computed is defined by the following expression: $\frac{p(p-1)}{2}$

### 5.1.4 Cholesky decomposition

This benchmark performs the Cholesky decomposition of a real Hermian positive definite matrix A into a lower triangular matrix L. Multiplying L by its transpose, results in the original matrix A = L × L$^T$ . In the same fashion as the matrix multiplication kernel, the input matrix A is distributed in square blocks of size BS × BS elements. This application is composed of four different kernels: gemm, trsm, syrk, and potrf. The gemm kernel is in fact a matrix-matrix multiplication. trsm and syrk have a data access pattern like the gemm. Potrf kernel in fact performs the sequential cholesky decomposition over the block. In the case of the Choleskiy decomposition, the KPI is number of floating point operations per second or GFlops/s. Given a problem with matrix sidel of length $n$ , the number of operations needed to compute a solution is defined in the following expression: $\frac{n^3}{3}$. Due to the different behaviour of the algorithm a smaller number of GFlops/s than in the case of the Matrix Multiplication algorithm is expected.

## 5.2 Results

### 5.2.1 SMP Results

**AMD (IDV-A)**

Table 5.2.1 Shows the IDV-A SMP performance across applications described in section 5.1. For each application KPI, average power and performance per watt are shown. All these applications perform computations using exclusively main system processor, no accelerators are used in this case. Also, it is worth noting that all available cores are used in these measurements. Execution time is computed using linux timing API, as described in section 4.1.1. Consumed power is measured using `likwid` in order to query RAPL interface, as described in section 4.2.1.

| Application | Performance | Average power (W) | Performance per watt |
|---|---|---|---|
| Matrix Multiplication (single precision) | 1547 GFlops/s | 331 | 4.6 GFlops/W |
| Matrix Multiplication (double precision) | 924 GFlops/s | 353 | 2.6 GFlops/W |
| N-body | 9.04 GPairs/s | 424 | 0.021 GPairs/W |
| Spectra | 1.7 GPairs/s | 253 | 0.0067 GPairs/W |
| Cholesky | 684 GFlops/s | 168 | 4.0 GFlops/W |

Table 5.2.1: IDV-A SMP Performance & Power measurements

Figure 5.2.1 shows a strong scalability study of the Matrix Multiplication application on the AMD node of Dibona. It can be seen that the performance improvements plateau at approximately 24 threads. This may be caused by two main reasons. First of all, this CPU runs two threads per core. However, performance is not expected to scale with the number of threads, and is expected to scale with the number of cores. This is due to the CPU having a fixed number of computational resources per core (not per thread) and different threads running in the same core end up sharing computational resources. This may help in certain

workloads, but in this example, as we can efficiently use all computational resources in each core, performance is not expected to grow once we use all the cores, even if more threads are used.

Furthermore, memory layout can impact performance as the system is composed of different NUMA nodes, and accessing data that is not directly attached to a given set of cores is slower. In this particular CPU, each socket is divided into four different NUMA nodes. This may explain why performance starts to grow at a slower than ideal pace well before all available cores are used. This may be caused due to problem data being initialized by a single core, which affects how data is physically laid out across physical memory.

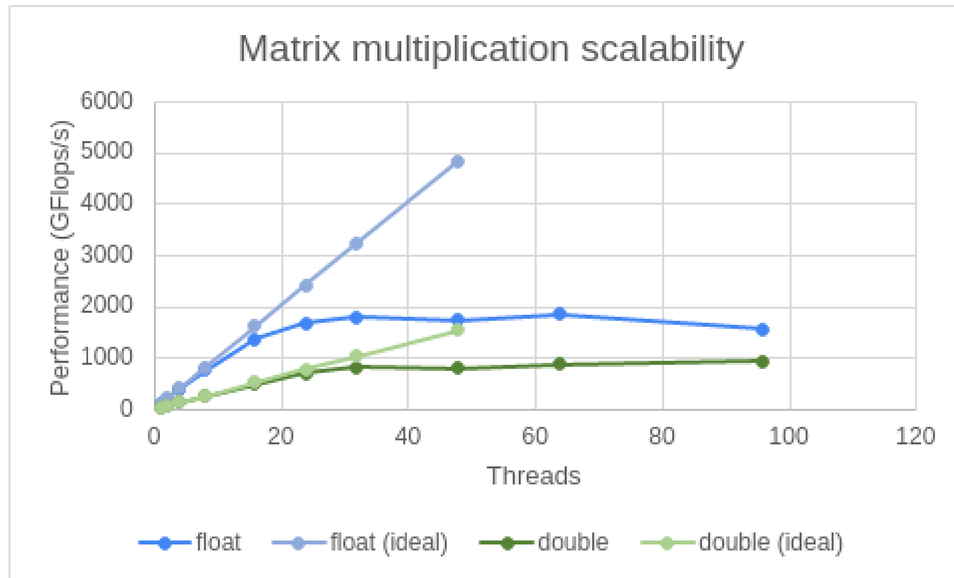



**Figure 5.2.1: AMD Processor Matrix Multiplication Strong Scalability Measurements**

## ARM (IDV-E)

Table 5.2.2 shows performance and power consumption for benchmarks described in section 5.1. They are run using CPU of the IDV-E prototype. Execution time is measured by using Linux timing API, described in section 4.1.1, in the same fashion as how time is measured in the case of the AMD CPU. However, power measurements are done by reading power consumption through sysfs sensors interface (`/sys/class/hwmon/hwmon*`). This is needed because the Ampere Altra Max CPU does not support RAPL interface. As we can see results for dense Algebra problems (Matrix Multiplication and Cholesky) are fairly good but all-to-all problems (N-Body and Spectra) obtain very poor performance when compared against the other computing alternatives.

| Application | Performance | Average power (W) | Performance per watt |
|---|---|---|---|
| Matrix Multiplication (single precision) | 1895.3 GFlops/s | 272.77 | 6.95 GFlops/W |
| Matrix Multiplication (double precision) | 500.02 GFlops/s | 237.52 | 2.11 GFlops/W |
| N-body | 1.88 GPairs/s | 150.32 | 0.0125 GPairs/W |
| Spectra | 2.78 GPairs/s | 148.32 | 0.01875 GPairs/W |
| Cholesky | 1106.7 GFlops/s | 212.68 | 5.20 GFlops/W |

Table 5.2.2: IDV-E ARM SMP performance & Power measurements.

Figure 5.2.2 shows a strong scalability study using the matrix multiplication benchmark, running in the Arm CPU of the IDV-E prototype. In this case performance improvements plateau at around 128 cores, which is one full socket. This behaviour is in line with the one seen for the AMD processor in previous section. In this case, however, there is no multithreading and therefore, performance should ideally scale linearly with the number of threads used until all available threads are used. In this case performance seems to be constrained on one side by available bandwidth, which may be limiting performance scalability before two sockets and cores from different NUMA nodes are used. On the other side, when using both sockets, we may be running causing all cores from one of the socket to access data from the neighbour NUMA node, which will have a negative impact in performance. It is also worth noting that difference in performance between single and double precision floating point is much larger than the difference shown in the AMD processor.

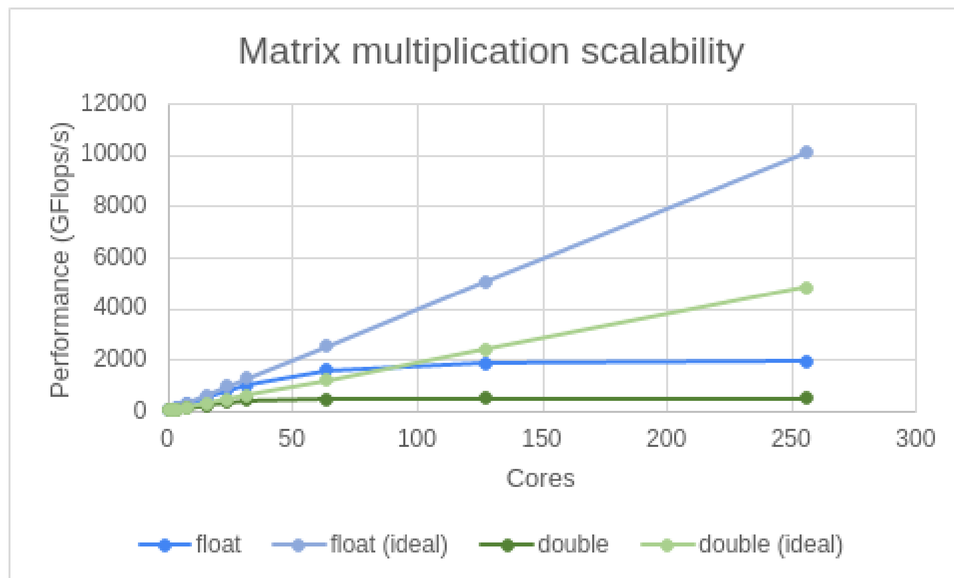



Figure 5.2.2: IDV-E ARM Processor Matrix Multiplication Strong Scalability Measurements

## 5.2.2 GPU Platform Results

Table 5.2.3 shows the performance and power results obtained when executing a simple CUDA porting of the applications described over the Dibona node. The results report for each application it's performance, average power and performance per Watt KPIs.

| Application | Performance | Average Power (W) | Performace per Watt |
|---|---|---|---|
| Matrix Multiplication | 18710 GFlops/s | 266 | 70 GFlops/W |
| N-Body | 195 GPairs/s | 194 | 1.0 GPair/W |
| Spectra | 543 GPairs/s | 313 | 1.7 Gpair/W |
| Cholesky | 17095 GFlops/s | 301 | 57 GFlops/W |

Table 5.2.3: CUDA Implementations Performance & Power Measurements

Listing 5.2.1 shows a profiling summary corresponding to the matrix multiplication execution shown in table 5.2.3. It shows how much time is spent in all API calls, cuda kernel statistics and and data movements between host and device.

```
CUDA API Statistics:

Time (%)  Total Time (ns)  Num Calls   Avg (ns)       Med (ns)       Min (ns)     Max (ns)     StdDev (ns)              Name
--------  ---------------  ---------  ------------  -------------  -----------  -----------  -------------  -----------------------
    42,0      903.565.896          8  112.945.737,0    1.699.253,0       88.750  554.518.201  214.897.970,0  cudaFree
    34,0      729.759.659          4  182.439.914,0  169.130.801,0  166.329.230  225.168.827   28.524.523,0  cudaMemcpy
    22,0      479.584.999          5   95.916.999,0        4.130,0        2.690  479.566.769  214.466.740,0  cudaDeviceSynchronize
0,0        7.736.237          6    1.289.372,0    1.033.202,0        4.770    3.758.478    1.455.238,0  cudaMalloc
     0,0          230.920        756         305,0          280,0          160        3.370          187,0  cuGetProcAddress
     0,0           50.590          1      50.590,0       50.590,0       50.590       50.590            0,0  cudaLaunchKernel
     0,0           23.361         18       1.297,0          710,0          440        8.101        1.756,0  cudaEventDestroy
     0,0           21.490         18       1.193,0          510,0          470       12.260        2.763,0  cudaEventCreateWithFla
     0,0            3.751          2       1.875,0        1.875,0        1.800        1.951          106,0  cuInit
     0,0            1.750          3         583,0          180,0          170        1.400          707,0  cuModuleGetLoadingMode

CUDA Kernel Statistics:

Time (%)  Total Time (ns)  Instances    Avg (ns)       Med (ns)       Min (ns)     Max (ns)    StdDev (ns)        Name
--------  ---------------  ---------  ------------  -------------  -----------  -----------  -----------  ------------------------
   100,0      479.566.167          1  479.566.167,0  479.566.167,0  479.566.167  479.566.167          0,0  cutlass::Kernel<…> (...)

CUDA Memory Operation Statistics (by size):

Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
----------  -----  --------  --------  --------  --------  -----------  ------------------
6442,451        3  2147,484  2147,484  2147,484  2147,484  0,000        [CUDA memcpy HtoD]
2147,484        1  2147,484  2147,484  2147,484  2147,484  0,000        [CUDA memcpy DtoH]
```

**Listing 5.2.1: GPU profiling summary output**

Listing 5.2.2 shows a summarized trace of the operations that are carried during the application. It includes various statistics for data movements as well as kernel executions performed during application execution.

```
CUDA Kernel & Memory Operations Trace:

  Start (ns)    Duration (ns)  CorrId  GrdX   GrdY  GrdZ  BlkX  BlkY  BlkZ  Reg/Trd  StcSMem (MB)  DymSMem (MB)  Bytes (MB)  Throughput
(MBps)  SrcMemKd  DstMemKd           Device            Ctx  Strm                                  Name

-------------  -------------  ------  -----  ----  ----  ----  ----  ----  -------  ------------  ------------  ----------  ---------
--------  --------  --------  ------------------------  ---  ----  ---------------------------------------------------------------
-----------------

3.549.319.778     167.851.982    2.035                                                                            2147,484    10737,418
Pageable  Device    NVIDIA A100-SXM4-40GB (0)    1     7  [CUDA memcpy HtoD]

3.717.254.992     166.098.772    2.036                                                                            2147,484    12884,902
Pageable  Device    NVIDIA A100-SXM4-40GB (0)    1     7  [CUDA memcpy HtoD]

3.883.435.076     165.906.805    2.037                                                                            2147,484    12884,902
Pageable  Device    NVIDIA A100-SXM4-40GB (0)    1     7  [CUDA memcpy HtoD]

4.100.804.282       478.604.564     2.041    2.048      32       1    128        1        1         126    0,000               0,066
NVIDIA A100-SXM4-40GB (0)    1     7  void cutlass::Kernel<cutlass_80_tensorop_d884gemm_64x64_16x4_nn_align1>(T1::Params)

4.587.263.345     225.384.638    2.044                                                                             2147,484    8589,935
Device     Pageable  NVIDIA A100-SXM4-40GB (0)    1     7  [CUDA memcpy DtoH]
```

**Listing 5.2.2: Cuda execution trace**

### 5.2.3 FPGA Platform Results

In this section we report the results obtained with the method described in this deliverable with the FPGA platform. Part of the results reported here have been used to submit a research article that has been accepted at the 31st IEEE International Symposium on Field-Programmable Custom Computing Machines [9] and is currently pending publication. Even though the purpose of this section is not to show results that are competitive with the state-of-the-art, but rather the viability of the measurement methods, we want to highlight that the results reported for this section improve said state-of-the-art.

#### Performance Results

Table 5.2.4 shows the baseline results of the applications described in section 5.1 in the FPGA platform. Each result is measured in the application's corresponding KPI. It is worth to highlight that the baseline results reported are obtained using the software developed inside the Textarossa project and, to the best of our knowledge, are in line (Matrix Multiplication) or better (all the remaining applications) than the best results reported for these applications over the analysed platform in the available scientific literature.

| Application | Blocking Size | Frequency (MHz) | Performance |
|---|---|---|---|
| Matrix Multiplication | 384 | 300 | 353 GFlops/s |
| N-Body | 2048 | 300 | 37 GPairs/s |
| Spectra | 2232 | 300 | 51 GPairs/s |
| Cholesky | 256 | 300 | 242 GFlops/s |

**Table 5.2.4: Baseline Implementations FPGA Performance Measurements**

Figure 5.2.3 shows the performance results obtained in the FPGA platform when executing the selected benchmarks. In order to better show a real evaluation result, the different implementation features developed in Tasks 4.2 Task-based Models and 4.6 SW integration & Optimization are compared between them (as reported in D4.1 TEXTAROSSA Progress Report on Programming Models and Runtime Systems, Section 3.3.3 Performance Improvements). The baseline column reflects the basic OmpSs@FPGA implementation of the benchmark. The Placement column reports the performance improvement obtained by managing the accelerators placement inside the FPGA area. Over that, the Priorities column reports the performance obtained by setting memory access priorities between the accelerators and the Interleave column adds to the previous features the distribution of the data among all the memories of the FPGA.

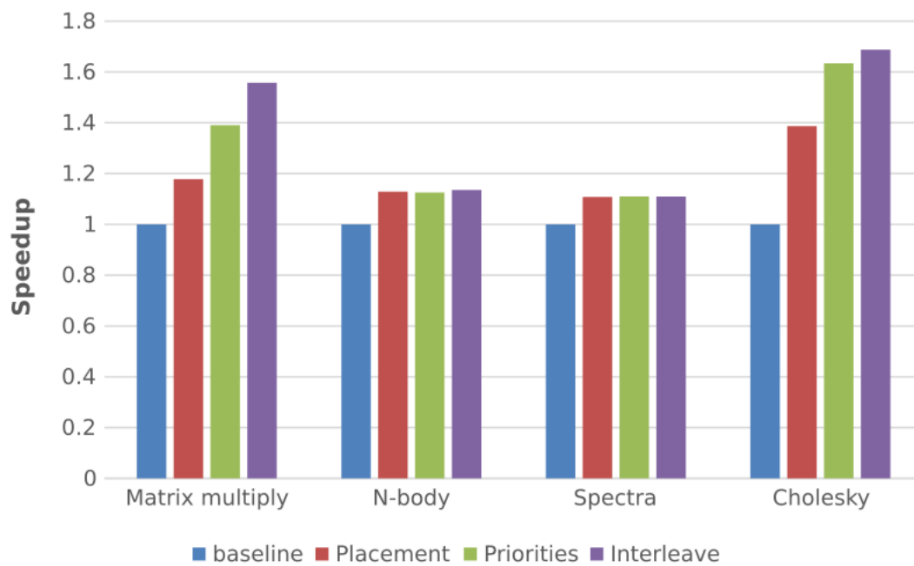



**Figure 5.2.3: FPGA Performance Measurements**

## Power Results

Table 5.2.5 shows the power results obtained in the IDV-E mitigation prototype using the measurement system integrated in the OmpSs@FPGA framework developed for the project as described in section 4.2.3. As it can be observed in Table 5.2.5, FPGA performance per Watt is competitive with a processor in dense linear algebra problems (Matrix Multiplication and Cholesky) and much higher for all-to-all problems (N-Body and Spectra) which is a common pattern in molecular dynamic applications.

If we compare performance per Watt with GPGPU accelerators (as reported in section 5.2.2), we can see that GPGPUs have higher performance per Watt. However, we can attribute this difference to two main factors. On the one hand the FPGA board used here (Alveo U200) is based in a 16 nm technology while the GPU reported in section 5.2.2 is based on a 7nm technology. That is approximately 5 years of technology difference. On the other hand, the numbers showed here refer to the consumption of the

accelerator alone. In a production environment, the accelerators should be scaled and connected to other components (like regular CPUs) in order to work and/or communicate between them. In this context, FPGAs hold an intrinsic advantage over GPUs as they can operate and communicate in a standalone model [13] while GPUs cannot. This implies that scaling the FPGA system is going to have significant performance gains and consumption reductions over scaling the GPU system.

| Application | Performance | Average Power (W) | Performace per Watt |
|---|---|---|---|
| Matrix Multiplication | 472.8 GFlops/s | 95.3 | 5.0 GFlops/W |
| N-Body | 37.7 GPairs/s | 105.1 | 0.36 GPair/W |
| Spectra | 51.5 GPairs/s | 90.6 | 0.57 Gpair/W |
| Cholesky | 319.4 GFlops/s | 85.9 | 3.7 GFlops/W |

**Table 5.2.5: OmpSs@FPGA Power Measurements**

### Results with different Accuracies

Figure 5.2.4 shows the performance obtained when using the Matrix Multiplication kernel with different accuracies and the same OmpSs@FPGA features as described in Figure 5.2.3.

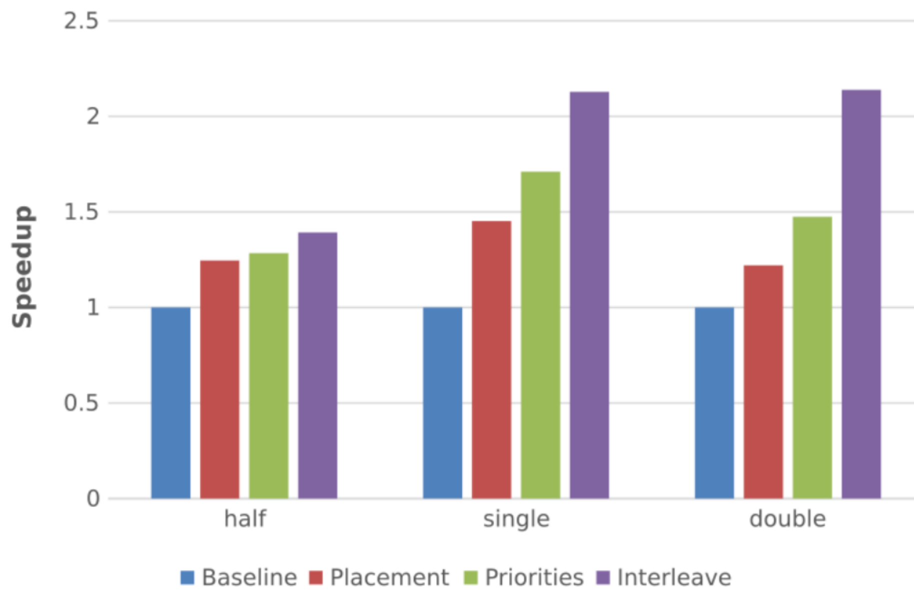



**Figure 5.2.4: Matrix Multiplication with Different Accuracies Performance Measurements**

# 6 Conclusions

This deliverable presents a complete and comprehensive design for benchmarking the project applications over the IDV platforms. Although due to supply chain issues the final platforms were not fully available in time to be presented in this report, alternate mitigation platforms of similar characteristics have been selected and used to elaborate the benchmarking plan and test its reliability and feasibility.

The benchmarking design presented includes a complete set of measurement techniques that can be used for all the project selected KPIs (as presented in D6.1 Applications and Use cases) for Performance, Energy and Accuracy across both project IDV platforms. Also, an example of how some HPC applications could be benchmarked has been carried out.

The results of the example benchmarking are also showed in this report. The example presents results from the project developments that are currently accepted for conference publication in the 31st IEEE International Symposium on Field-Programmable Custom Computing Machines. In addition, these first example tests also show some architectural aspects of the IDV systems that will help in exploiting their available resources. This allows us to conclude that the methodology elaborated could be followed by the project applications to obtain the project final results in the last year.

# References

[1] Intel Guidance: Running Average Power Limit https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html. Last retrieved January 2023.

[2] Khan, Kashif & Hirki, Mikael & Niemi, Tapio & Nurminen, Jukka & Ou, Zhonghong. (2018). RAPL in Action: Experiences in Using RAPL for Power Measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS). 3. 10.1145/3177754.

[3] Intel Corporation 2015. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3, System Programming Guide. Intel Corporation.

[4] LIKWID Performance Tools: https://hpc.fau.de/research/tools/likwid/. Last retrieved February 2023.

[5] Management SDK User Guide: NVIDIA Virtual GPU Software Documentation. https://docs.nvidia.com/grid/15.0/grid-management-sdk-user-guide/index.html. Last retrieved February 2023.

[6] GPowerU GPU Power Measurement Tool: https://github.com/crrossi/GPowerU/tree/main/GPowerU

[7] J. M. de Haro, J. Bosch, A. Filgueras, M. Vidal, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguadé, and J. Labarta, "Ompss@fpga framework for high performance fpga computing," IEEE Transactions on Computers, vol. 70, no. 12, pp. 2029–2042, 2021.

[8] González, C.; Balocco, S.; Bosch, J.; de Haro, J.M.; Paolini, M.; Filgueras, A.; Álvarez, C.; Pons, R. High Performance Computing PP-Distance Algorithms to Generate X-ray Spectra from 3D Models. Int. J. Mol. Sci. 2022, 23, 11408. https://doi.org/10.3390/ijms231911408

[9] Antonio Filgueras, Miquel Vidal, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell. Improving Performance of HPC Kernels on FPGAs Using High-Level Resource Management. The 31st IEEE International Symposium on Field-Programmable Custom Computing Machines. *Accepted, publication pending*. 2023

[10] EuroEXA project. https://euroexa.eu/. Last retrieved, September 2022.

[11] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, Jesús Labarta: Application Acceleration on FPGAs with OmpSs@FPGA. FPT 2018: 70-77

[12] OmpSs@FPGA Github page. https://github.com/bsc-pm-ompss-at-fpga. Last retrieved, March 2023.

[13] Juan Miguel De Haro Ruiz, Rubén Cano, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, François Abel, Burkhard Ringlein, Beat Weiss: OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing. IPDPS 2022: 828-838

# Appendix A. IDV-E Prototype proof-of-concept

In order to support OmpSs@FPGA for the Textarossa IDV-E, support for running in ARM processors is required. OmpSs and OmpSs-2 already support ARM architecture. In fact, both OmpSs support a wide range of processor architectures, including x86-64, ARM, or RISC-V among many others. However, ARM is not officially supported by Xilinx as host processor (apart from Zynq Family with shared memory between programmable logic and the SMP with ARM cores). This means that device drivers are not guaranteed to work on an ARM CPU as a host to be connected to Xilinx FPGA board.

Before testing on actual hardware, a proof-of-concept test was done to check the feasibility of building system software in the target platform.

On one hand, we have tested the **custom flow**, which relies on QDMA device drivers and user space low level access to PCIe resources. At the beginning, this first test setup consisted of a virtualized ARMv8 system on an intel host. Then, PCIe resources were forwarded from the host to the guest operating system in order to provide PCIe low level access to the host system. As soon as this test was successful and a minimal set of features was verified, we carried on testing to an actual ARMv8 based system to perform functionality tests as well as early performance tests.

Issues found in the virtualized environment were also found to be in line with experiments carried out on actual hardware. Even though device drivers should make use of Linux PCIe subsystem, which abstracts the device driver from the underlying architecture, this seems not to be completely true for Xilinx's QDMA device drivers. In particular, device drivers contain x86-64 inline assembly, which is not legal in ARM architecture as shown in the excerpt of code of listing A.1.

```
asm volatile("rdtscp" : "=a" (low), "=d" (high));
return low | ((u64)high) << 32;
```

**Listing A.1: Erroneous driver code**

However, the erroneous code can be safely deleted, as it is only used for internal driver statistics that do not interfere with driver functionality. After these issues are resolved, we were able to verify that all software works as expected for the OmpSs flow, which does not use Xilinx's Vitis flow and XRT runtime libraries. Listing A.2 shows the bitstream information uploaded in the FPGA connected to the ARM host. This can be read from the ARM host command line and provides information about the frequency of the accelerators, number of accelerators, the accelerators names, etc.

```
Bitstream info version:   9
Number of acc:      8
Base freq:    156 MHz
AIT version: 5.24
Wrapper version     12
Features:
0x184
[ ] Instrumentation
[ ] Hardware counter
[x] Performance interconnect
[x] Extended HW runtime
```

```
[x] SOM
[ ] POM
xtasks accelerator config:
type   #ins   name   freq
0000000006708694863 001    matmulFPGA                     300
0000000004353056269 007    matmulBlock                    300
ait command line:
main.py   --name=matmul   -b=alveo_u200   -c=300   --hwruntime=som   --
interconnect_opt=performance --interconnect_regslice all --verbose --
slr_slices=all            --floorplanning_constr=all               --
placement_file=u200_placement_7x256.json --wrapper_version=12

Hardware runtime VLNV:
bsc:ompss:smartompssmanager:4.7
```

Listing A.2: Accelerator information

Also, some performance tests have been carried out to ensure that there are no unexpected performance differences when comparing with x86-64.

On the other hand, Xilinx's **Vitis flow** is based on XRT for runtime support, and device drivers. This is the expected path from the point of view of Vitis programming flow. For XRT we also found some incompatibilities. Xilinx states that ARM is supported, but this is only true for the ZynqMP device drivers. These are different from the PCIe ones, as the FPGA in ZynqMP devices is not attached via PCIe. In fact, *zocl* drivers fail to natively build as they expect to be cross compiled using a petalinux kernel tree. In this case, incompatibilities have been found only in the build system itself. If the target system is an ARM CPU, it's assumed that we're cross compiling for a Zynq device. Once the build system is modified to allow native ARM builds, all device drivers and runtime libraries, build and work as expected.

In the same fashion as the QDMA software, built-in tests have been executed to verify the correct behavior of system software as shown in listing A.3.

```
Compiled   kernel   =   xilinx_u200_xdma_201830_2/test/verify.xclbin
Original            string            =             [b'\x00\x00\...']
Original            string            =             [b'\x00\x00\...']
Issue               kernel                start                requests
Now   wait   for   the   kernels   to   finish   using   xrtRunWait()
Get the output data produced by the 2 kernel runs from the device
Result          string          =          [b'Hello             World']
Result          string          =          [b'Hello             World']
PASSED TEST
```

Listing A.3: Vitis flow test output

### System verification results

In this project we have worked over the results of our previous European projects to further develop the OmpSs@FPGA ecosystem [11]. In the EuroEXA project [10] we developed the OmpSs@FPGA support for CRDB. CRDB was a heterogeneous system composed of a ZynqMP SoC and a Virtex UltraScale XCVU9P. In Textarossa project we have designed and developed new extensions of OmpSs toolchain to support the new IDV-E platform. Indeed, we have improved our previous performance results [7] that are now used as the baseline.

On the one hand, we have done some sanity checks on the new IDV-E platform to verify that the setup is working properly. Some performance tests have also been performed in order to check that reasonable performance is obtained and we're not running into any software compatibility issue in our setup.

Figure A.1 shows performance (in GFLOPS, the higher the better) across different test environments and accelerator configurations. It shows performance across 3 different systems:

- An intel-based Xeon X5680 CPU with an Alveo u200 FPGA
- An ARM-based Cavium CN8890 with an Alveo u200 FPGA
- An ARM-based CRDB containing a ZynqMP SoC and a Virtex UltraScale XCVU9P

For each test environment, different configurations are tested. One with 4 small 64x64 matrix multiplication accelerators (4x64 in the Figure) and a larger one with 7 256x256 accelerators (7x256 in the Figure). For each accelerator configuration, task creation and spawn can be done by the main CPU (cHost in the Figure) or by the FPGA accelerators (cFPGA in the Figure). Figure does not show results for the large configuration and the CRDB machine since it doesn't fit in the FPGA.

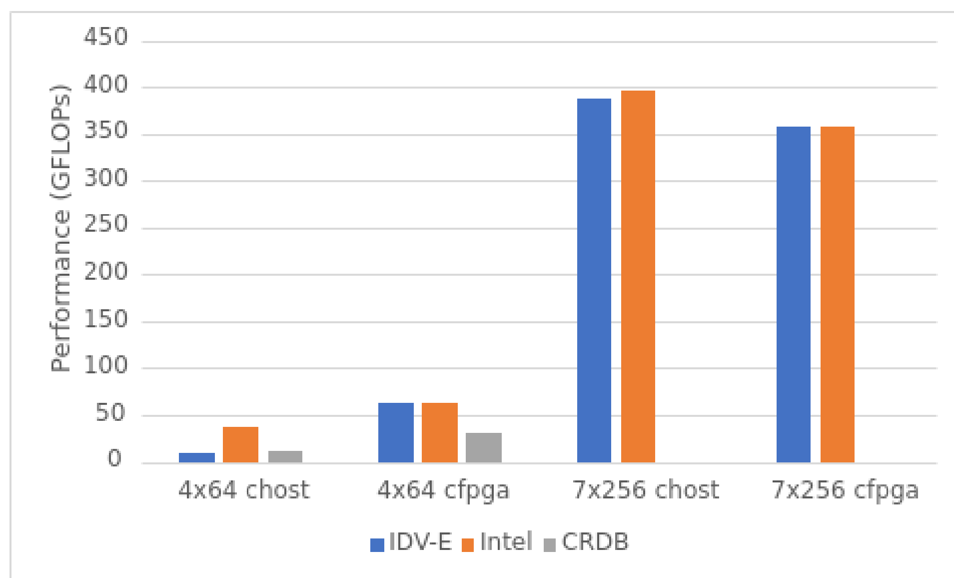



**Figure A.1: Performance comparison between ARM and Intel systems**

For small accelerators, where the cost of creating tasks on main CPU is the limiting factor, IDV prototype performance is similar to CRDB. This is expected as the single core performance of both CPUs is similar. When tasks are created from inside the FPGA CRDB shows less performance. This is due to this machine having slower memory attached to the FPGA. For larger accelerators, SMP performance is not relevant anymore as performance is limited by the computing resources of the accelerators themselves. There's no test data for CRDB in this configuration because it would not fit available FPGA resources in the system.

As a result of this first test, we can conclude that the first objective of the project has been accomplished with the OmpSs@FPGA framework properly working over the IDV-E platform.

# Appendix B. Benchmark codes example

All the benchmarks shown in this deliverable (except Spectra [8] that doesn't belong to BSC) can be obtained in the OmpSs@FPGA githup page [12]. Here we show a snippet of the Matrix Multiplication FPGA code to show the seamless integration of the reconfigurable accelerator code in the main program code. Listing B.1 shows the accelerator code that executes a Matrix Multiplication Block in the FPGA.

```
#pragma omp target device(fpga) num_instances(MBLOCK_NUM_ACCS)
#pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
void matmulBlock(const elem_t *a, const elem_t *b, elem_t *c)
{
  #pragma HLS INLINE // off
  #pragma HLS array_partition variable=a cyclic factor=MBLOCK_FPGA_PWIDTH/64
  #pragma HLS array_partition variable=b cyclic factor=BSIZE/(MBLOCK_II*2)
  #pragma HLS array_partition variable=c cyclic factor=BSIZE/MBLOCK_II
#ifdef USE_URAM
  #pragma HLS RESOURCE variable=b core=XPM_MEMORY uram
  #pragma HLS RESOURCE variable=a core=XPM_MEMORY uram
#endif

  for (int k = 0; k < BSIZE; ++k) {
    for (int i = 0; i < BSIZE; ++i) {
      #pragma HLS pipeline II=MBLOCK_II
      for (int j = 0; j < BSIZE; ++j) {
        c[i*BSIZE + j] += a[i*BSIZE + k] * b[k*BSIZE + j];
      }
    }
  }
}
```

**Listing B.1: FPGA Matrix Multiplication Block Accelerator Code**

Listing B.2 shows the main code using the accelerator from the SMP host. As it can be seen from Listing B.2, the invocation of the accelerator is as simple as calling the function annotated with the device(fpga) clause. The OmpSs@FPGA framework takes care of the data transfers between the SMP and the accelerator memory and transforms both the host and accelerator code so they can communicate across the platform communication channel [7]. Although the OmpSs@FPGA system was already developed for other platforms, in the context of Textarossa the framework has been extended to address the IDV-E platform, being, to the best of our knowledge the first system able to operate in a system composed of an ARM host processor and a PCIe connected FPGA. At the moment of writing this deliverable the framework has been tested in the IDV-E final platform in E4 (just the first operational tests) allowing seamless integration of the accelerators in the platform.

```
void matmulSMP(const elem_t *a, const elem_t *b, elem_t *c, const unsigned int msize) {
  const unsigned int b2size = BSIZE*BSIZE;
  for (unsigned int i = 0; i < msize/BSIZE; i++) {
```

```
      for (unsigned int k = 0; k < msize/BSIZE; k++) {
        unsigned int const ai = k*b2size + i*BSIZE*msize;
        for (unsigned int j = 0; j < msize/BSIZE; j++) {
          unsigned int const bi = j*b2size + k*BSIZE*msize;
          unsigned int const ci = j*b2size + i*BSIZE*msize;
          matmulBlock(a + ai, b + bi, c + ci);
        }
      }
    }
}
```

**Listing B.2: Main code invoking the FPGA Matrix Multiplication Block Accelerator Code**