eman ta zabal zazu

Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

INFORMATIKA
FAKULTATEA
FACULTAD
DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Benchmarking the performance and energy consumption of the AVX512 and VNNI instruction sets

Author

Jon Arriaran Cancho

2022

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

Benchmarking the performance and energy consumption of the AVX512 and VNNI instruction sets

Author

Jon Arriaran Cancho

Director(s)

Jose Antonio Pascual

Contents

Appendices

List of Figures

Abstract

This project is focused on measuring the execution time, the energy consumption and the performance of the new instruction set introduced by Intel in the Cascade Lake series of processors which are called Vector Neural Network Instructions (VNNI). These instructions are part of the AVX512 instruction set, and they are specifically designed to accelerate deep learning codes. To analyse the performance of these instructions, a set of benchmarks will have to be designed and developed. In addition, the impact of using these instructions inside HPC containers will be also evaluated because HPC clusters are the natural place to use these high-end architectures.

1. CHAPTER

Introduction

The birth of this project was inspired by the most recent Intel Xeon Cascade Lake series processors, which were released with the possibility of executing VNNI instructions applying the already available AVX-512 instruction set. A similar behaviour of the VNNI instruction set has been executed only on GPUs until nowadays, so the performance and efficiency these instructions could reach on a processor, it is, at least, something unknown and worth studying.

AVX-512 is a set of CPU instructions that affects storage, compute and network functionalities. The number 512 in the name of the set refers to the size, in bits, of the register file. The registers define how much data can be operated within an instruction at a time. Its predecessor, AVX2, could only compute with a 256 bits register file. So AVX-512 duplicates AVX2 on the number of floating points per seconds (FLOPS) per clock it can reach. In other words, AVX-512 is able to process twice the number of data elements that AVX2 is able to process. It can also accelerate performance for workloads and use cases, such as scientific simulations, AI, 3D modelling, cryptography…

The fact that the considerable AVX-512 instruction set performance upgrade can be used alongside VNNI instructions, brings the processors the possibility to may compete with GPUs. The reason why GPUs are used to execute Vector Neural Network Instructions similar behaviours, is because Neural Networks need the computational power and performance that GPUs offers. Due to, GPUs are capable of executing simple instructions with lots of numbers at one time.

A Neural Network [7] consist of a computational learning system that uses a network of functions to understand and translate a data input in one form into the desired form. The concept of that feature was inspired by human biology neurons, which understands inputs from human senses and bring the brain an output from them. While input and outputs were calculated on different cycles of the instruction execution, the way of execution itself evolves and improves, decreasing the time to obtain the next output from an input by increasing the performance of the method to get it.

VNNI uses this execution method for executing some types of complex calculations, as well as image classification, speech recognition, language translation, object detection and Artificial Intelligence. The third generation [2] of VNNI originally uses last VPDDPBUSD 8-bit instruction type and BFLOAT16 floating-point format, allowing to reach more efficient and faster results compared to their own older versions.

The main goal of the project is to create and develop a set of benchmarks which will use these VNNI instructions with AVX-512 set in many ways. So, it could be possible the evaluation and comparison of the performance and energy efficiency they could obtain. The set of benchmarks will be executed on a cluster deployed at the Informatics Faculty of the UPV/EHU in Donostia/San Sebastián. The physical description and the way of how this cluster is managed is going to be described later. The evaluation will be made on different scenarios inside this cluster, consisting of many modes of executing the same main benchmark, in a different amount of cores.

Once the base evaluations were done, the idea is to continue evaluating their performance with other technologies such as RAPL and inside Singularity HPC containers, for instance, complementing with them the previously made evaluations. Finally, better and more complete conclusions of the power consumption, execution time and frequency performance of the VNNI instruction set will be drawn.

2. CHAPTER

The aims of the project

As previously stated, this project main goal consist of testing the most recent VNNI instruction set with AVX-512 [1] instruction set on the Intel Xeon Cascade Lake series processors. The main idea is to first execute this test on a cluster using these VNNI on different ways and configurations. From those executions, an output and conclusions of the result will be got and taken, comparing the results obtained with the outcome of other configurations.

The program that will implement the usage of VNNI instructions have to stress the processors, as we need to put the processors of the cluster on their execution performance limit, to see how they respond with different loads. Therefore, the first aim of the project is going to be creating a set of benchmarks which demands a lot of loads to the processor.

The program will be written in C language, because the C programming language is one of the few languages that implements the possibility of using the AVX-512 instruction set. This instruction set can be imported to the program with *immintrin.h* [4] header, mostly known as Intel Intrinsics. This package contains all the instructions needed for vectorization, as the ones used on AVX-512. The VNNI instructions, which are also included in the same package, are four in particular. The program will use these one at a time, plus an additional mixed mode of the four, along with another modes of common AVX-512 instructions. Benchmarks result will contain the execution time of the selected mode and the frequency of the processors during that test, saving all the results on a *.out* file type. However, it is necessary to format these outputs, so we could manage them in a better way.

The second aim of this project entails adding the possibility of capturing the energy consumption as the execution of the benchmark goes. To achieve this, *RAPL* [10] technology will be used. This technology is implemented on C language and gets the energy consumption of the processors while it is executing any other thing. The implementation of this feature implies little changes on the Benchmark, as we need to get the new energy consumption data.

Finally, the project will also be executed inside *Singularity* [11] HPC containers [3], which are supposed to be prepared to execute stressful programs. *Singularity* technology brings the potential of using high loads programs without affecting the performance, and that performance is what is going to be analysed. To sum up, these are the three aims this project it is based on:

1. Create the benchmark for executing VNNI and AVX512 instructions.
2. Add to the benchmark the energy measure and analysis of the executions.
3. Analysis of the performance of executing the benchmark inside Singularity container.

Once these aims are reached and completed, the results will be analysed and compared, getting a final conclusion of the performance that AVX-512 instruction set and VNNI instructions could get.

3. CHAPTER

Project management

A planning on a project, through its different stages, is an essential step. A good planing is capable of identify the risks in the project and helps on the managing the most valuable resource in a large scale project of this sort, that is, time. The objective of the planing is to expose every aspect of the project of significance that must be committed in due time with deadlines in check and to find how to efficiently distribute the tasks in that regard, so that potential delays can be prevented, and the project can be successfully completed in time.

The management work will be divided in three main categories, namely: Project management, project development and, as the last category, documentation of the project. These three main phases cover the major aspects and the crucial decisions, as well as how difficulties have been confronted through the project entire process. The modules of each phase constitute its focus points that have to be dealt with for the most adequate transition and evolution towards milestones and goals.

3.1 Description of the phases and their features

The different phases of the project will be explained here to show which has been the dynamic of the project, how each part alone provides useful guidance and, all in all, a watchful observation about how the entire process works as a whole.

3.1.1 Management phase

The principal focus in this phase is to estimate the cost of different tasks and how the project could potentially evolve across its duration. Since this is an early phase of the project, estimating the time, resources and potential risks is of huge importance to be able to track afterwards if the project is going according to plan or there are some (major) deviations with regard to how it was planned in the beginning.

After choosing the specific tasks that have to be carried out, the most volatile part consists on placing them in the correct moment and for the right amount of time. Additionally, the project has to be tracked periodically to ensure that the milestones set for the project are being met or if there is need of some readjustments that allow to better handle tasks for the best possible outcome.

There are three main modules that define the management of the project to take into account:

- **Planning**: The main points of the project have been covered, estimating their possible duration and the resources needed for their realization, as well as searching for when these objectives can be fulfilled and the order in which they are completed. The result is a set of activities ordered and placed across the time with their respective milestones and with a risk management plan to be ready for risks with prior knowledge on how to avoid them. Finally, the scope of the project is determined.

- **Tracking**: In order to check whether the objectives are being completed under the given time and to counter the present risks that can cause unexpected delays, the project is analysed during its progression. Finding new risks, modifying the milestones and creating new objectives or replacing older ones is the main purpose.

- **Communication**: This parts joins the previous two modules and an assessment is conducted to evaluate how is the project progressing and to identify which tasks are going according to plan and which others are not. All of this is explained in detail to the directors of the project so that they can be up-to-date and informed of any kind of alterations in the plans. These issues are handled in periodical meetings when certain milestones have been finished or when a new risk has emerged. In the end, the tasks until the next meeting are decided, which are usually intentions for a short term.

3.1.2 Development phase

The project is heavily oriented towards testing and analysing the VNNI commands performances in the new Intel Xeon Cascade Lake processors. Most of the development phase consist on create some programs to execute and test these commands. The development will be done on different programming languages, using each one on the best situations. C language will be use on the development of the Benchmark, and Python for the formatting and analysing of the results got by the Benchmark. The main problems of the development phase will be the proper developing of the programs, and the proper execution of them. It can be from programming issues due to the compilation of the program, to the warranty of that our results are not influenced by the load of the cluster.

One all of this type of issues are studied, the programs will be developed, with a previous study of what we want to get as a result, to finally compare and draw conclusions about them.

3.1.3 Documentation of the project

The last part involves the report and final conclusions of the project, where the most relevant knowledge and information is gathered about the research conducted, both for the theoretical part and for the practical application. Since bench-marking requires a good background, the document relies on the analysis of amount of result and provides charts that further help to view easily the conclusions and results.

The amount of executions it has to be done through this phase is huge, so it is necessary to keep a good planing of how storage and manage them.

3.2 Estimations

After covering the three phases that compose the project, the initial estimation of time and the finally needed amount of time to complete the tasks will be displayed. In bold, the estimation for a phase is registered, while below, for each phase, each specific task of the project is broken down belonging to that phase. The estimated time and the final time of each task are summed for finding out how much time each phase has required in table on Figure 3.1.

	Estimated time	Final time
Management phase	65	67
Planning	30	25
Tracking	20	30
Communication	15	12
Development phase	55	84
Benchmark Development	10	8
Result formatting program development	15	25
Chart generator program development	15	25
RAPL implementation	5	20
Singularity Implementation	10	6
Documentation phase	220	210
Benchmark explanation and documentation	30	35
Result formatting program explanation and documentation	20	25
Frequency analysis	30	25
Execution time analysis	30	25
Energy Consumption analysis	50	40
Final conclusion discussion	40	45
Find out possible future work	20	15
Total amount of time	340	361

Figure 3.1: Estimation of tasks and their final required time.

3.3 Deviations

The deviations occurred have altered the initial planning, requiring some modifications in the task and even including a new task that wasn't intended in the first place. Some other minor deviations are also addressed.

The major deviation corresponds to getting and formatting correctly the results obtained by the program developed in the first aim. Due to the addition of the other technologies, the raw results of the benchmark add more lines of values to read, so the first formatting program version does not create the results in the correct way. This program will be changed first with the addition of RAPL to measure energy and power, and also changed for a second time when it was going to be launched by *Singularity*. The other tasks of the project were done as expected and planned, taking more time than the expected on some tasks of the documentation phase.

4. CHAPTER

Preliminaries

On this chapter, a little introduction to different technologies and some basic knowledge will be given. This is intended to could situate and understand the deep explanations of all the part that confront the main investigation of this project better. The introductions will be given respecting the order of their usage during the development of the project. First, for instance, the cluster where the project will be located is going to be introduced, because the chapter where this information is necessary is the next to this one.

4.1 Introduction to the Project Deployment Cluster

This project, as mentioned in the introduction chapter, will be carried out by executing a benchmark program, getting results in many ways and modes. This benchmark must be executed on a controlled and trusted environment, so the results are as reliable as possible for their final analysis. It is also necessary suitable processing power, because the benchmark that is going to be executed, generates many processing loads for its correct running.

On top of that, the executor processors must include the possibility of using the AVX-512 and VNNI instruction set. For all of these reasons, *Priscilla* cluster is the perfect choice to could generate and execute the benchmark on the best conditions. On this section, a physical and hardware explanation of this *Priscilla* cluster is going to be described.

Priscilla cluster is located on the Informatics Faculty in Donostia/San Sebastián, and owned by the *Intelligent Systems Group*[1] at the University of the Basque Country. The cluster is deployed on the first floor of the faculty, next to other type and independent amount of clusters, and all of them are refrigerated and equipped with the best security system. *Priscilla* is built as a cluster. Every node is numbered and each of them have, at least, two processors of the same type.

The only nodes of the cluster capable of executing the instruction sets the project is based around are the ones numbered from node 50 to 53 and the node number 150. On the next Figure 4.1 a physical description of the node 150, the only one from the nodes mentioned which includes GPUs, is shown:

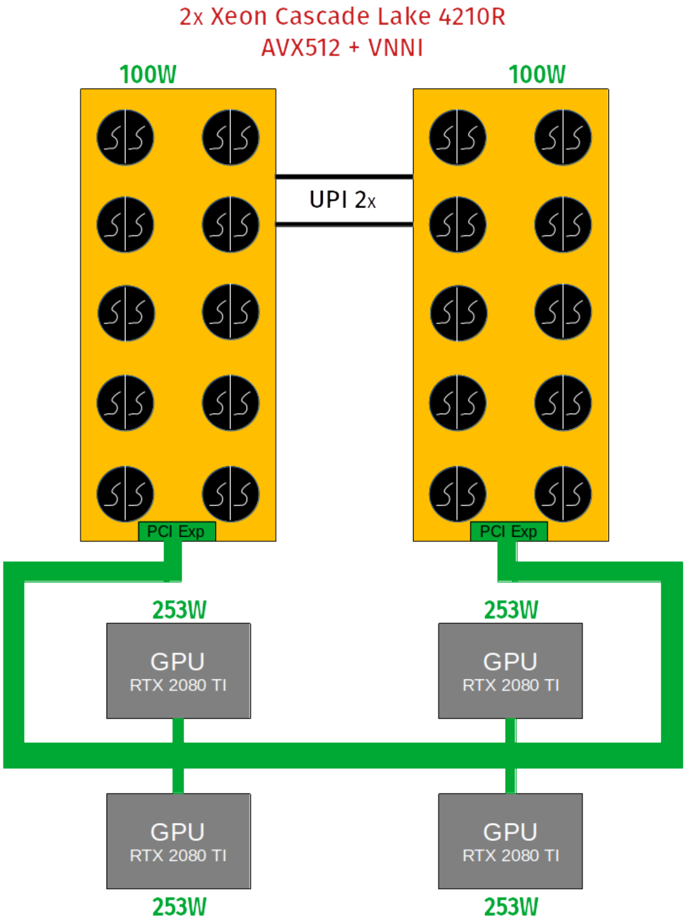

Figure 4.1: Physical Description of node 150

[1] http://www.sc.ehu.es/ccwbayes/

As you can see on the image, every node has two Intel Xeon Cascade Lake series 4210R processors, which have the aptitude to execute the already mentioned VNNI instructions also using AVX-512 instruction set. These two processors count with 10 processor cores (extendable to 20 by activating hyper-threading) and are connected to each other using a pair of UPI links.

In addition, on this, and only on node 150, four RTX 2080 TI GPUs are connected by PCI Express, and working simultaneously with NVLinks. As previously said, nodes between 50 and 53 are exactly the same as the node 150, but with the omission of the GPUs. All nodes also have 512 GB of DDR4-3200 RAM memory and one TB of SSD memory.

There are more nodes than these mentioned ones on the cluster, but they are uncapable of running VNNI. Consequently, they are going to be kept out of this explanation. The whole cluster will be managed by Slurm Workload Manager, which will be introduced in the section below.

4.2 Slurm Workload Manager introduction

Slurm Workload Manager is an open source, highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm does not require any kernel modifications for operating, and it has three main functionalities.

First of all, it allocates exclusive and/or non-exclusive access to the resources of the cluster that users can approach for a period of time, so they can perform whatever they want to execute on the cluster. It also provides a framework for starting, executing and monitoring work on the set of allocated nodes. And as a final functionality, it arbitrates contention for resources by managing a queue of pending.

All of these functions can be complemented with optional extra plugins, which are used for accounting, advanced reservation, gang scheduling, topology optimized resource, resource limits and other likes add-ons. Accounting and advanced reservation plugins are going to be used on this project.

Slurm [12] is based on a centralized manager, **slurmctld**, to monitor resources and work. Each cluster (node) has a **slurmd** daemon, which can be compared to a remote shell: it waits for work, executes that work, return status, and wait for next work. These daemons provide fault-tolerant hierarchical communications.

Users have some tools to execute and manage all the job and events they want to carry out. This user tools include **srun** to initiate jobs, **scancel** to terminate queued or running jobs, **sinfo** to report system status, **squeue** to report the status of the jobs and **sacct** to get information about jobs that are running or have completed.

But this commands will not be used for launching the main job, just for visualizing the actual state of the jobs. Instead, Slurm also provides the possibility to run a job from a **sbatch** function, which allows users to queue a job from a *.sbatch* file, where users can configure a job to run on the way they want.

4.3 Introduction to RAPL program

RAPL energy measurement is a technology aimed to read and measure the energy consumption of any program execution on Linux. RAPL can read Linux kernel results on three different ways to extract the energy information from there.

The first way is to directly read the files under */sys/class/powercap/intel-rapl/intel-rapl:0* using the Linux kernels powercap [9] interface, which provides a consistent interface between the kernel and the user space that allows power capping drivers to expose the settings to user space in a uniform way. This requires no special permissions, and it has been available since Linux 3.13.

The second way is to use the *perf event* interface [8], which requires root privileges, and it is executed as *sudo perf stat -a -e "power/energy-cores/" /bin/ls*. This tool was introduced on Linux 3.14, and it is available on all other newer versions. It is also used to read the performance of an event by the creation of a description file for each executed event or event group. The last way is to use raw-access to the underlying MSRs [6] under */dev/msr*. This file contains all the information relevant to the CPU, but it is a protected file, so a root privileges are necessary to read the information.

RAPL is going to be used with the powercap interface, as it is not necessary to provide privileges. The concept is to read and measure energy performance during the execution of our tests, so the obtained results could be compared to see what configuration is more worthy on energy consumption later. The programs produce results like in the following Figure 4.2:

```
Power units = 0.125W
Energy units = 0.00001526J
Time units = 0.00097656s
Package thermal spec: 130.000W
Package minimum power: 51.000W
Package maximum power: 200.000W
Package maximum time window: 0.046s
Package energy before: 48460.887909J
PowerPlane0 (core) for core 0 energy before: 36127.280838J
DRAM energy before: 0.000000J
Sleeping 1 second
Package energy after: 48468.194504  (7.306595J consumed)
PowerPlane0 (core) for core 0 energy after: 36128.297287  (1.016449J consumed)
DRAM energy after: 0.000000  (0.000000J consumed)
```

Figure 4.2: RAPL example

RAPL could also be used with PAPI, an addition program for reading RAPL values in different ways, so it can generate and plot some charts. But self generated charts will be used as explained on the appendix Chapter A, so this tool is not going to be used alongside the project.

4.4 Singularity containers introduction

Singularity is a container solution created by necessity for scientific and application driven workloads. It improves the very well-known containers' usability nowadays, giving better performance on the scientific and computation (HPC) community. A container consist of grouping and isolating applications which are going to be executed on a specific operative system.

Containers also could be seen as a machine virtualization, which contains the operative system and all applications want to be included on it. With that, it is possible to group different operative system for executing and use any application, because they connect and use the container, which includes all needed to execute. Being a controlled environment, a cluster could be installed on it, so any host who connects to the cluster, no mattering which operative system is using, can use what it is inside the container.

The only bad side of the containers is that their use reduces the performance when talking about executing so demanding programs, as the ones used on scientific and computation fields. Here is when Singularity appears, to provide us the possibility of executing a demanding program, such our VNNI instruction, without the downgrade on their performance.

Singularity also is able to save a container as an image file. This file contains the entire container and can be copied, shared, archived and more, giving to Singularity such a good mobility of compute. Image file mobility reasons also gives a very good reproducibility. As other container use cases, like Docker, it is necessary to install first all the needed into the container. From operative system to every package and application needed to perform what it is wanted on it, but this is something that is going to be explained later on this document, side by side, with the explanation of what is what it is going to be configured.

5. CHAPTER

Zagreus Benchmark Development

The purpose of this chapter is to explain deeply all the development carried on the creation of the main program will be used on the project. The idea is to explain all the parts that confront the program, explaining why and how that way of development is taken on each part one by one. As it is necessary for a correct understanding of a program, partially some figures will be implemented along with the explanation of the moment, representing the exact part of the code is referring to as line numbers. First of all, an explanation about the development of the program will be given, continuing with an explanation about how it will be executed, separated on their respective sections.

5.1 Zagreus development and explanation

The chosen name for the benchmark it is going to execute on the described cluster is *Zagreus*. This program will execute the repeatedly referred VNNI instruction set with AVX-512, but on this section, a deeper explanation and description of the program will be done step by step. The program was made on C language, because the C programming language is one of the few languages capable of using the instructions and importing the needed packages, together with C++ and assembly languages. The whole program is separated on different C files, giving each file one specific role in the program.

5.1.1 *main.c* and *globals.h* files

The main C file of *Zagreus* is named *main.c*, and it is the heart of our program, where the other parts of the program are managed. These are the necessary libraries for the main file of the program (all the references to a specific part of some codes are shown with the same code lines as in the program):

```
#include <stdio.h>
#include <stdlib.h>
#include <immintrin.h>
#include <time.h>
#include "avx512_vnni.h"
#include "avx512.h"
```

Figure 5.1: *main.c* libraries

As we can see on the Figure 5.1 above, *stdio.h* and *stdlib.h* libraries are the first two imported on the program. These are two of the most important header files used in C programming. *stdio.h* is the header file for Standard Input Output and contains the usually used *printf()* and *scanf()* declarations. Meanwhile, *stdlib.h* header file contains declaration of *malloc()*, *free()*, *rand()* and *atoi()*. Furthermore, *time.h* header file contains some declarations for getting the actual time of the device during the execution of the program. However, *immnitrin.h* is the most important header file of all of them. That is because it is use in other C files too and contains the declarations for could execute AVX-512 and VNNI instruction sets.

In addition to this already created headers, another created headers will be imported to the main program. These *avx512_vnni.h* and *avx512.h* headers, will contain declarations of functions created on the other C files of the program, those that execute a specific role of the program. As it will be shown and described later in this section, these example of header contains declarations that are going to be used on the program. Once necessary libraries are successfully imported on the main file, some final global variables will be created to use on the execution, being below listed variables the parameters who describe the behaviour of *Zagreus* benchmark:

- **mode:** this variable is used for selecting the main execution mode of the program. There are only two program behaviours. First of them, it is aimed to execute only AVX-512 base instructions just for setting a ground of information, for later could compare it with AVX-512 and VNNI combined instructions performance. The second behaviour is aimed to execute those VNNI instructions with AVX-512. Therewith, the parameter is saved as *int* type number, where:

    - [mode = 1] equals to AVX-512
    - [mode = 2] equals to AVX-512 + VNNI.

- **command_num:** this variable is used for describing the quantity of instructions will be executed. This value will be multiplied by 1,000,000, just for simplifying the way at the time of passing parameters. For example, if 500 number is given as the parameter input, 500 will be stored as *int* type number but will be multiplied and used as 500,000,000 later.

- **m512_size:** AVX-512 consist of instructions that use a number type sized by 512 bytes. This number could be filled in different ways, but in the project only four types will be used to see if different sizes could take effect and change the performance of the benchmark. This is going to be only used on VNNI instruction set, due to there are the only instructions that use *m512* number type. This parameter will save the chosen size for the instance of the execution, creating a 512 byte size number as in the following examples:

    - [m512_size = 8] 64 numbers of 8 byte size to create a *m512* number type.
    - [m512_size = 16] 32 numbers of 16 byte size to create a *m512* number type.
    - [m512_size = 32] 16 numbers of 32 byte size to create a *m512* number type.
    - [m512_size = 64] 8 numbers of 64 byte size to create a *m512* number type.

- **exec_mode:** in addition to the behaviours of the *Zagreus* benchmark, the program has also the possibility to chose between five different modes of executing the main program behaviour. This parameter saves which of the five execution modes will be executed for the chosen main behaviour. These are the ten configurations altogether:

  For mode 1 behaviour (AVX-512)

  - [exec_mode = 1] only a mul type command.
  - [exec_mode = 2] mul and add type commands.
  - [exec_mode = 3] mul, add and reduce type commands.
  - [exec_mode = 4] 3mul, 2add and reduce type commands.
  - [exec_mode = 5] 3mul, 3add, 2div and reduce commands.

  For mode 2 behaviour (AVX-512 + VNNI)

  - [exec_mode = 1] *_mm512_dpbusd_epi32()* instruction execution.
  - [exec_mode = 2] *_mm512_dpbusds_epi32()* instruction execution.
  - [exec_mode = 3] *_mm512_dpwssd_epi32()* instruction execution.
  - [exec_mode = 4] *_mm512_dpwssds_epi32()* instruction execution.
  - [exec_mode = 5] on each iteration of *command_num* one of the previous four instruction will be executed randomly.

These parameters will be saved as global variables, which could be used on the other files of the program by a helper header called *globals.h*. On this header, these parameters will be described as *extern int* type and just including the header on the other files the variables will be accessible to use. All the executions and instructions functionalities will be explained deeper later on their respective C file development descriptions. But this is all in all, the parameter managing done by *main.c* program, implemented on the program with the next lines of Figure 5.2:

```
        //Get parameters
        //---------------------------------------------------------------
        mode = atoi(argv[1]);
        command_num = atoi(argv[2]);
        m512_size = atoi(argv[3]);
        exec_mode = atoi(argv[4]);
        //---------------------------------------------------------------

        //Detection of errors on arguments
        //---------------------------------------------------------------
        if (mode <= 0 || mode > 3) {
            fprintf(stderr, "Error on mode selection \n"
                            "Argument has to be between 1-3 \n");
            return 1;
        }
        else if (command_num <= 0) {
            fprintf(stderr, "Error on command_num selection \n"
                            "Argument has to be higher than 0 \n");
            return 1;
        }
        else if (m512_size <= 0) {
            fprintf(stderr, "Error on vectors sizes selection \n"
                            "Argument has to be higher than 0 \n");
            return 1;
        }
        else if (exec_mode <= 0 || exec_mode > 5) {
            fprintf(stderr, "Error on execution mode selection \n"
                            "Argument has to be between 0 and 5 \n");
            return 1;
        }
        //---------------------------------------------------------------
```

Figure 5.2: *main.c* parameter managing

After the parameters are read and formatted correctly, the *main.c* file executes the correct configuration, looking to the *mode* parameter to execute the correct behaviour in the execution. On this part of the program, the current time will be taken for once the execution is done, take the time again and print the execution time by calculating the different between the two measurements. As you can see on the below Figure 5.3, the execution of the behaviour itself is managed by two functions described on the *execute_avx512()* and *execute_avx512_vnni()* external C files:

```
        clock_gettime (CLOCK_REALTIME, &t0);
        if (mode == 1) { execute_avx512();}
        if (mode == 2) { execute_avx512_vnni();}
        clock_gettime (CLOCK_REALTIME, &t1);
        tex = (t1.tv_sec - t0.tv_sec) + (t1.tv_nsec - t0.tv_nsec) / (double)1e9;
        printf("%.2fs \n", tex);

```

Figure 5.3: *main.c* behaviour execution

5.1.2 *avx512.c* and *avx512.h* files

This part of the program is aimed to execute the AVX-512 instructions, and it is based on a single function named *execute_avx512()*. The header program *avx512.h* just references to *execute_avx512()*, so the inclusion of the header on other files of the program will allow using the function. The description of this *execute_avx512()* is stored on the C type file. As previously stated in the 5.1.1 Subsection where the parameters were explained, this behaviour of the benchmark has five execution modes. Before execute any of these modes, some allocations and initializations have to be done.

```
    int i, j;
    int reduce;
    float *a, *b;
    __m512 va, vb, vmul, vadd, vdiv;

    //Reserve memory for helpful vectors
    a = (float *) aligned_alloc (64, 64*sizeof (float));
    b = (float *) aligned_alloc (64, 64*sizeof (float));

    //Initialize vectors with random numbers
    srand (1);
    for (int j = 0; j < 64; j++) {
        a[j] = (rand() % 10);
        b[j] = (rand() % 10);
    }
```

Figure 5.4: *avx512.c* private variables initializations

As it is shown on the code part of Figure 5.4 above, two named *a* and *b float list* variables are reserved on the memory, using *aligned_alloc()* function included on *stdlib.h* header. After the reservation of these variables, they will be initialized by *rand()* instruction, so there are never two same executions, getting as a consequence more reliable results.

Once all of this is done, the execution mode will be executed, divided on two *for* functions. The first of them will be performed the same time as the *command_num* parameter, and on each cycle of this loop, the execution mode will be done 1,000,000 times. This is done on this way because of the limits of an *int* number type, due to the result of the multiplication of the parameter and 1,000,000 exceeds the maximum size of an *int*. The next Figure 5.5 describes the different execution modes:

```
        switch(exec_mode) {
        //Execution mode 1, only a mul command
        case 1:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    va = _mm512_load_ps(&a[0]);
                    vb = _mm512_load_ps(&b[0]);
                    vmul = _mm512_mul_ps(va, vb);
                }
            }
            break;

        //Execution mode 2, mul and add command
        case 2:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    vadd = _mm512_setzero_ps();
                    va = _mm512_load_ps(&a[0]);
                    vb = _mm512_load_ps(&b[0]);
                    vmul = _mm512_mul_ps(va, vb);
                    vadd = _mm512_add_ps(vadd, vmul);
                }
            }
            break;

        //Execution mode 3, mul, add and reduce command
        case 3:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    vadd = _mm512_setzero_ps();
                    va = _mm512_load_ps(&a[0]);
                    vb = _mm512_load_ps(&b[0]);
                    vmul = _mm512_mul_ps(va, vb);
                    vadd = _mm512_add_ps(vadd, vmul);
                    reduce += _mm512_reduce_add_ps(vadd);
                }
            }
            break;
```

Figure 5.5: *avx512.c* 1, 2, 3 execution mode descriptions

There are only five functions used on these execution modes. As you can see, the more you increase the number of the execution mode, more increases the total load of it. The first thing done on each execution is to charge the previously reserved randomly initialized vectors on *_m512* type variables, using *_mm512_load_ps()* or *_mm512_setzero_ps()* for initializing the helper variables.

After this first steps, different amount of multiplications, sums, divisions and reductions were done by using *_mm512_mul_ps()*, *_mm512_add_ps()*, *_mm512_div_ps()* and *_mm512_reduce_add_ps()* respectively. The same happens with execution modes 4 and 5 described on the Figure 5.6:

```
        //Execution mode 4, 3mul, 2add and reduce command
        case 4:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    vadd = _mm512_setzero_ps();
                    va = _mm512_load_ps(&a[0]);
                    vb = _mm512_load_ps(&b[0]);
                    vmul = _mm512_mul_ps(va, vb);
                    vmul = _mm512_mul_ps(va, vmul);
                    vadd = _mm512_add_ps(vadd, vmul);
                    vmul = _mm512_mul_ps(vadd, vmul);
                    vadd = _mm512_add_ps(va, vmul);
                    reduce += _mm512_reduce_add_ps(vadd);
                }
            }
            break;

        //Execution mode 5, 3mul, 3add, 2div and reduce command
        case 5:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    vadd = _mm512_setzero_ps();
                    va = _mm512_load_ps(&a[0]);
                    vb = _mm512_load_ps(&b[0]);
                    vmul = _mm512_mul_ps(va, vb);
                    vmul = _mm512_mul_ps(va, vmul);
                    vadd = _mm512_add_ps(vadd, vmul);
                    vmul = _mm512_mul_ps(vadd, vmul);
                    vadd = _mm512_add_ps(va, vmul);
                    vdiv = _mm512_div_ps(vadd, vb);
                    vdiv = _mm512_div_ps(vdiv, va);
                    vadd = _mm512_add_ps(vdiv, vmul);
                    reduce += _mm512_reduce_add_ps(vadd);
                }
            }
            break;
    }

    //Free used memory
    free(a);
    free(b);
```

Figure 5.6: *avx512.c* 4 and 5 execution mode descriptions

_mm512_mul_ps(): This instruction performs a multiplication between two *_m512* number types.

_mm512_add_ps(): This instruction performs a sum between two *_m512* number types.

_mm512_div_ps(): This instruction performs a division between two *_m512* number types.

_mm512_reduce_add_ps(): performs a reduction function for a *_m512* number type.

Once the execution is done, the reserved memory is released by *free()* function and the program will continue its execution on the *main.c* file, ending the benchmark as previously explained, taking the time after execute this part of the code, and calculating the spent time of the execution.

5.1.3 *avx512_vnni.c* and *avx512_vnni.h* files

On the same way that *avx512.c* and *avx512.h* files work, these two files describes a function that is used on *main.c* file. On this case, the function is called *execute_avx512_vnni()* and it has a similar functionality as *avx512_vnni.c*, being possible to use it by importing the header file. On the C type file, the first lines correspond to the description and initialization of the variables will be use on the next parts of the function.

```
    int i, j;
    srand(1);
    __m512i src, result, A, B;
    int a = (rand() % 10) + 1;
    int b = (rand() % 10) + 1;
    int c = (rand() % 10) + 1;

    uint64_t arr_a[512 / m512_size];
    uint64_t arr_b[512 / m512_size];
    uint64_t arr_src[512 / m512_size];
    for (int j = 0; j < 512 / m512_size; j++) {
        switch (m512_size) {
            case 64:
                arr_a[j] = (uint64_t) a;
                arr_b[j] = (uint64_t) b;
                arr_src[j] = (uint64_t) c;
                break;

            case 32:
                arr_a[j] = (uint32_t) a;
                arr_b[j] = (uint32_t) b;
                arr_src[j] = (uint32_t) c;
                break;

            case 16:
                arr_a[j] = (uint16_t) a;
                arr_b[j] = (uint16_t) b;
                arr_src[j] = (uint16_t) c;
                break;

            case 8:
                arr_a[j] = (uint8_t) a;
                arr_b[j] = (uint8_t) b;
                arr_src[j] = (uint8_t) c;
                break;
        }
    }

    A = _mm512_loadu_si512((__m512i * ) & arr_a);
    B = _mm512_loadu_si512((__m512i * ) & arr_b);
    src = _mm512_loadu_si512((__m512i * ) & arr_src);
```

Figure 5.7: *avx512_vnni.c* variables description

As you can see on the previous Figure 5.7, three *_m512i* number type arrays will be initialized as the base of our function. For this, some random numbers will be generated and resized with parsing, depending on the *_m512_size* selected and passed as parameter. After the resizing, the main *_m512i* variables will be initialized by the use of the *_mm512_loadu_si512*.

Once the initializations are done, the execution of the instruction will be tested starts. As in the *avx512.c* file, the behaviour of the VNNI instruction testing has also five execution modes, selected by the *exec_mode* parameter, which are going to be explained following the next two Figures 5.8 and 5.9.

```
        switch (exec_mode) {
        case 1:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    result = _mm512_dpbusd_epi32(src, A, B);
                }
            }
            break;

        case 2:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    result = _mm512_dpbusds_epi32(src, A, B);
                }
            }
            break;

        case 3:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    result = _mm512_dpwssd_epi32(src, A, B);
                }
            }
            break;

        case 4:
            for (i = 0; i < command_num; i++) {
                for (j = 0; j < 1000000; j++) {
                    result = _mm512_dpwssds_epi32(src, A, B);
                }
            }
            break;
```

Figure 5.8: *avx512_vnni.c* 1 to 4 execution modes description

On each execution mode from 1 to 4, a specific VNNI instruction will be executed and tested, *command_num* parameter described amount of times by a for loop function. However, on the last fifth execution mode, each cycle of the *command_num* for function loop, randomly one of the previous four function will be selected and executed.

```
case 5:
    for (i = 0; i < command_num; i++) {
        for (j = 0; j < 1000000; j++) {
            int command;
            command = rand() % 4;
            switch (command) {
                case 0:
                    result = _mm512_dpbusd_epi32(src, A, B);
                    break;

                case 1:
                    result = _mm512_dpbusds_epi32(src, A, B);
                    break;

                case 2:
                    result = _mm512_dpwssd_epi32(src, A, B);
                    break;

                case 3:
                    result = _mm512_dpwssds_epi32(src, A, B);
                    break;
            }
        }
    }
    break;
```

Figure 5.9: *avx512_vnni.c* 5 execution mode description

_mm512_dpbusd_epi32(src, a, b): this VNNI function multiply groups of 4 adjacent pairs of unsigned 8-bit integers in a with corresponding signed 8-bit integers in b, producing 4 intermediate signed 16-bit results. Sum these 4 results with the corresponding 32-bit integer in src, and store the packed 32-bit results in dst.

```
FOR j := 0 to 15
    tmp1.word := Signed(ZeroExtend16(a.byte[4*j]) * SignExtend16(b.byte[4*j]))
    tmp2.word := Signed(ZeroExtend16(a.byte[4*j+1]) * SignExtend16(b.byte[4*j+1]))
    tmp3.word := Signed(ZeroExtend16(a.byte[4*j+2]) * SignExtend16(b.byte[4*j+2]))
    tmp4.word := Signed(ZeroExtend16(a.byte[4*j+3]) * SignExtend16(b.byte[4*j+3]))
    dst.dword[j] := src.dword[j] + tmp1 + tmp2 + tmp3 + tmp4
ENDFOR
dst[MAX:512] := 0
```

Figure 5.10: *_mm512_dpbusd_epi32(src, a, b)* instruction description

_mm512_dpbusds_epi32(src, a, b): this VNNI function multiply groups of 4 adjacent pairs of unsigned 8-bit integers in a with corresponding signed 8-bit integers in b, producing 4 intermediate signed 16-bit results. Sum these 4 results with the corresponding 32-bit integer in src using signed saturation, and store the packed 32-bit results in dst.

```
FOR j := 0 to 15
    tmp1.word := Signed(ZeroExtend16(a.byte[4*j]) * SignExtend16(b.byte[4*j]))
    tmp2.word := Signed(ZeroExtend16(a.byte[4*j+1]) * SignExtend16(b.byte[4*j+1]))
    tmp3.word := Signed(ZeroExtend16(a.byte[4*j+2]) * SignExtend16(b.byte[4*j+2]))
    tmp4.word := Signed(ZeroExtend16(a.byte[4*j+3]) * SignExtend16(b.byte[4*j+3]))
    dst.dword[j] := Saturate32(src.dword[j] + tmp1 + tmp2 + tmp3 + tmp4)
ENDFOR
dst[MAX:512] := 0
```

Figure 5.11: *_mm512_dpbusds_epi32(src, a, b)* instruction description

_mm512_dpwssd_epi32(src, a, b): this VNNI function multiply groups of 2 adjacent pairs of signed 16-bit integers in a with corresponding 16-bit integers in b, producing 2 intermediate signed 32-bit results. Sum these 2 results with the corresponding 32-bit integer in src, and store the packed 32-bit results in dst.

```
FOR j := 0 to 15
    tmp1.dword := SignExtend32(a.word[2*j]) * SignExtend32(b.word[2*j])
    tmp2.dword := SignExtend32(a.word[2*j+1]) * SignExtend32(b.word[2*j+1])
    dst.dword[j] := src.dword[j] + tmp1 + tmp2
ENDFOR
dst[MAX:512] := 0
```

Figure 5.12: *_mm512_dpwssd_epi32(src, a, b)* instruction description

_mm512_dpwssds_epi32(src, a, b): this VNNI function multiply groups of 2 adjacent pairs of signed 16-bit integers in a with corresponding 16-bit integers in b, producing 2 intermediate signed 32-bit results. Sum these 2 results with the corresponding 32-bit integer in src using signed saturation, and store the packed 32-bit results in dst.

```
FOR j := 0 to 15
    tmp1.dword := SignExtend32(a.word[2*j]) * SignExtend32(b.word[2*j])
    tmp2.dword := SignExtend32(a.word[2*j+1]) * SignExtend32(b.word[2*j+1])
    dst.dword[j] := Saturate32(src.dword[j] + tmp1 + tmp2)
ENDFOR
dst[MAX:512] := 0
```

Figure 5.13: *_mm512_dpwssds_epi32(src, a, b)* instruction description

In this case, it is not necessary to free any reserved memory, due to has not been done any reservation previously. After executing the requested execution mode the requested times amount, the function will end and the benchmark will continue on the *main.c* file, taking and calculating the execution time of it.

5.1.4 *Zagreus* makefile

Zagreus benchmark compilation will be done using a makefile. This allows us to could compile the program in an easier and more comfortable way. As you can see on the next Figure 5.14 lines, the description of this makefile is something so simple, being the *make* command the way of building the program and *make clean* as the way of deleting the unnecessary files.

```
all: main

colors.o: colors.c colors.h
    gcc -c colors.c

avx512_vnni.o: avx512_vnni.c avx512_vnni.h
    gcc -c avx512_vnni.c -mavx512f -march=cascadelake

avx512.o: avx512.c avx512.h
    gcc -c avx512.c -mavx512f

main.o: main.c avx512.h
    gcc -c main.c -mavx512f

main: main.o avx512.o avx512_vnni.o colors.o globals.h
    gcc -o main main.o avx512.o avx512_vnni.o colors.o -mavx512f -g

clean:
    rm -f main *.o
```

Figure 5.14: *makefile* file description

It is very important to add the *-mavx512f* and *-march=cascadelake* packages on *gcc* instruction, due to without these packages the cluster will not be able to know that VNNI and AVX512 instructions will be executed on the program, and an error will be shown at the time of compiling the *Zagreus* benchmark.

5.2 Zagreus benchmark execution description

As mentioned, the way to execute some program on the *Priscilla* cluster is managed by *Slurm Workload Manager*. So the next scripts were created to automatize the execution of the benchmark for the best usability of the program. The automatization of the execution consist of two parts. The first one, passes the parameters to generate the *.sbatch* file on the way it is desired and execute it. The second script, takes the parameters passed by the first script and generates the correct *.sbatch* file. But they will be better explained with the below Figure 5.15:

```
#!/bin/sh
for i in {1..20}
do
    ./launcher_info $i $1 > slurm.sbatch
    sbatch slurm.sbatch
done
```

Figure 5.15: *launcher* script description

launcher is the name for the first script described before. It relates to a single for loop, which increments the *i* variable from 1 to 20 values. This variable represents the amount of cores will be used on the execution of the full benchmark. So in total, twenty executions of the benchmark will be done, each of them with a different amount of cores on a specific cluster node. The node will be selected by the execution of this script, and passed as parameters to the second script next to the amount of cores on the first line of the loop. Following, once the *.sbatch* file was generated, it will be executed by a *sbatch slurm.sbatch* command.

For example, an execution of the *Zagreus* benchmark for getting the first results of it, will be done by executing a command like: *./launcher 53*. On this example we are generating 20 jobs, each of them with different amount of core usage on the node number 53 of the *Priscilla* cluster, and the results will be saved on a *.out* files. This will be explained together with the next script explanation on Figure 5.16.

```
#!/bin/sh
dolar='$'
cat <<File
#!/bin/bash
#SBATCH --job-name=zagreus
#SBATCH --output=zagreus_node$2_$1_cores.out
#SBATCH --ntasks=$(( 2*$1 ))
#SBATCH --threads-per-core=1
#SBATCH --ntasks-per-core=1
#SBATCH --mem-per-cpu=2G
#SBATCH --partition=AVX
#SBATCH --nodelist=node$2
#SBATCH --exclusive
```

Figure 5.16: *launcher_info* script description (SBATCH part)

The first look of this *launcher_info* script seems more convoluted than the *launcher* one. It consists of two parts, the first describes and manage Slurm configuration, and the second one describes the commands will execute on the Slurm execution.

As you can see on the Figure 5.16, Slurm configuration is given by some labels at the start of the *.sbatch* file, so following *#SBATCH* string some labels could be passed. The name of the execution and the name of the output file, where the results will be prompted, are *–job-name* and *–output*. In our case, the output file name will be different depending on the node where it is executing the program and the amount of cores it is using for the execution.

Continuing, the configuration of the resources will be used by the program it is configured. The labels for that are *–ntask*, which defines how many cores of the program will be used on the execution, and *–mem-per-cpu*, a label that describes so much memory of the CPU will be allocated for the execution of the program. *–ntask* is supplemented by *–threads-per-core* and *–ntask*-per-core, this labels define how many threads will be used per core, in case that our CPU cores have more than one, and how much ntask could be executed by a single core. Setting last one to a single nstask per core, it is insured to use one core of the CPU per each ntask configured before on the *–ntask* label.

In addition, the *–partition* one describes which Slurm partition of the cluster is going to be used on the execution of the program. These partitions collect a group of nodes on them, so if is wanted to execute something on X numbered node, this labels has to contain a partition where this X node is included. The different partition list could be deployed by executing *sinfo* command on the cluster.

–nodelist describes on which node the configured program will be executed. As explained just before, the node must be on the partition description. As the last, but not less important label, *–exclusive* label will be added. This configuration reserves all the cores of the node where the program will be executed, ignoring if they will be used or not. This will be done to prevent any deviations on the results of the benchmark.

Imagine that the program is being executed by only two cores out of twenty available on the selected node. The remaining eighteen cores will be free for executing on them any other program, and this could change and decrease the total power of the node, altering the results of the benchmark, as it could show a decreased performance due to the simultaneous executions and communications on the whole node. Allocating all the cores of the node, it is insured that only the necessary executions will be done during the *Zagreus* execution.

```
        echo "#AVX mode"
        for j in {1..5}
        do
                echo "#${dolar}j mode"
                for i in 500 1000 2000 5000
                do
                        echo "#${dolar}i times"
                    for k in ${dolar}(seq 1 ${dolar}CORE)
                do
                            ./Zagreus/main 1 ${dolar}i 16 ${dolar}j &
                        done
                        wait
                        echo "#Frecuency"
                        lscpu | grep -e "CPU MHz:"
                done
        done
        echo "#AVX+VNNI mode"
        for j in 8 16 32 64
        do
                echo "#${dolar}j size"
                for k in {1..5}
                do
                        echo "#${dolar}k mode"
                        for i in 500 1000 2000 5000
                        do
                                echo "#${dolar}i times"
                                for l in ${dolar}(seq 1 ${dolar}CORE)
                                do
                                        ./Zagreus/main 2 ${dolar}i ${dolar}j ${dolar}k &
                                done
                                wait
                                echo "#Frecuency"
                            lscpu | grep -e "CPU MHz:"
                       done
                done
        done
        File
```

Figure 5.17: *launcher_info* script description (BASH part)

On the Figure 5.17, it is shown the part of the code from the *launcher_info* script that describe which commands will be executed on the cluster. This is a simple BASH script where both behaviours of the *Zagreus* benchmark will be executed, each of them using all the configurations possible of the selected behaviour.

Zagreus AVX512 testing behaviour will be launched on the first place, and the parameters given to the benchmark will be incremented on the next order: *command_num* and then *exec_mode*. This means that the configuration will start with *exec_mode = 1*, and it will be executed for the 500, 1000, 2000 and 5000 values of *exec_mode*, being like that for all the *exec_mode* possibilities.

It is important to launch as many as benchmark tests as cores of the node we are using, and add at the end of the execution command a *&* symbol, so every node waits to all of them have finished continuing with the next configuration.

The second part of the code references to the second behaviour of *Zagreus* benchmark, the one that uses the VNNI instructions with *AVX512* instructions. It works on the same way of the previous behaviour, executing all the *command_num* possible values for each *exec_mode* possible values. But, in addition, it executes every *exec_mode* possible values for each *m512_size* values, which are, 8, 16, 32 and 64.

In total, a hundred of configurations of the *Zagreus* benchmark will be executed and saved for every node core amount, being two thousands the configurations have to be executed for a whole test of the benchmark. Doing what has been shown up to now, the test is only able to get the execution time of the benchmark, because of that, and for complementing the obtained results, a Linux *lscpu* command will be executed after every cycle of the whole test.

This *lscpu* command returns valuable information of the current status of the CPU. But it will be used on the script just for getting the frequency value of the CPU after executing the Benchmark, and complementing with that, the Slurm script result getting, for, in a future, format, analyse and take conclusions of them.

6. CHAPTER

Energy analysis of the executions

After the first aim is completed correctly, the project is ready to implement a technology overhead to the achieved until now, complementing the entire functionality with another type of result. In this case, RAPL will be the technology chosen for getting the energy consumption of the executions of the *Zagreus* benchmark, as it was introduced in Section 4.3. Besides, this chapter will bring all the information, collected on different sections, about the changes done to the actual *Zagreus* in such a way to obtain this new type of measures and RAPL implementation.

6.1 What is wanted to get

The main reason to implement this RAPL program is measuring some energy. The used energy is going to be obtained by measuring the quantity of Joules spent on an execution of the benchmark. On the same way, the power used on executions will be measured by Watts. All of this data will be taken from the CPU status during the execution.

The idea, after getting this energy and power information, is to generate some charts of them, so it simplifies the way of viewing the new measure results. For that it is necessary to change the program that generates this charts, and thereby, the way of getting this new information from the raw results created by *Zagreus*. In addition, it is necessary to do little changes in the way of executing the benchmark with *Slurm*, but all of this will be explained in the next sections.

6.2 How RAPL works

The version[1] of RAPL it is going to be used on the program simplifies the utility and usage of the energy measures, and it is implemented on a *bash* language. It works reading the information of the CPU stored on a specific directory of the computer it is executing, all of this is detailed on its explanation Section 4.3.

So, the functionality of this *bash* script consist on reading, by a simple *cat* command of Linux, on different points of the execution, the information save on the files. After this, the information taken will be gathered and presented as an output by a *printf* command. On the case of the usage it will be given in this project, only two of the printed final results will be useful for it, but it gives to much more additional, and not less useful measures.

The name of the *bash* script is **rapl_logger.sh** and it is so easy to use. It executes by an execution command of Linux, and it has to be followed by the program that is wanted to be measured, reaching to having to execute something similar to this *./rapl_logger <your-app> <params-of-your-app>*. It is supposed that RAPL works without having to set special read permissions of the files it reads, but some troubles happened when the program was tried on the project. Finally, some permissions were given to RAPL, looking for simplifying its implementation.

6.3 Creation of RAPL launcher scripts

As it is needed to change the mode of executing the benchmark for implementing RAPL and measure consumed energy and power, some changes has to be done on the scripts that launches this benchmark. Similar to the original script that launches the benchmark, that was explained in a previous Section 5.2. Copying the functionality of the previous script, a new script called *launcher_rapl* will be implemented, and it will launch another new script called *launcher_rapl_info* that contains the details of generating the *.sbatch* file for using RAPL, shown after the next Figure 6.1 as Figure 6.2.

[1] https://github.com/ulopeznovoa

```
#!/bin/sh
 for i in {1..20}
 do
     ./launcher_rapl_info $i $1 > slurm.sbatch
     sbatch slurm.sbatch
 done
```

Figure 6.1: *launcher_rapl* script description

```
for k in ${dolar}(seq 1 ${dolar}CORE)
do
    ./Zagreus/rapl_logger.sh ./Zagreus/main 1 ${dolar}i 16 ${dolar}j &
done
```

```
for l in ${dolar}(seq 1 ${dolar}CORE)
do
    ./Zagreus/rapl_logger.sh ./Zagreus/main 2 ${dolar}i ${dolar}j ${dolar}k &
done
```

Figure 6.2: *launcher_rapl_info* script description

This time the changes are so simple on the *launcher_rapl_info*, so it is only shown the lines of the script that changes in comparison to Figure 5.17. It just modifies the way of executing *Zagreus* benchmark to put it along RAPL functionalities.

6.4 Changes done to *Zagreus* to implement RAPL

Some changes have to be done also on how the results are taken for the generation of the *.res* formatted result files. In addition, and with these new formatted results that will contain the energy and power data measured by RAPL, two new type of charts will be generated. The charts will show the results of each test type for every core amount configuration of the *Zagreus* execution, as it does until now for frequency, but with the addition of a comparison between each execution mode implemented on VNNI behaviour.

These will be done to compare the performance difference between all the four VNNI instructions, due to, each execution mode from 1 to 4, contains a test of each instruction, as it was introduced in Figure 5.8 and its explanations.

Just the same way that the frequency is taken until now, and as it was described on Figures A.3 and A.6, some variables will be defined and filled for energy and power measures.

The variables will save the data of two behaviours of the benchmark, but on the VNNI behaviour also the differences between sizes and execution modes will be collected for its comparison. The names these variables will take goes from each execution configuration it is wanted to collect, as it is abridged on the next Figure 6.3.

```
    energy_num = 0

    energy_min_avx = 100000000
    energy_max_avx = 0
    energy_count_avx = 0
    energy_summatory_avx = 0

    energy_min_size8 = 100000000
    energy_max_size8 = 0
    energy_count_size8 = 0
    energy_summatory_size8 = 0

    energy_min_mode1 = 100000000
    energy_max_mode1 = 0
    energy_count_mode1 = 0
    energy_summatory_mode1 = 0
```

```
    power_num = 0

    power_min_avx = 100000000
    power_max_avx = 0
    power_count_avx = 0
    power_summatory_avx = 0

    power_min_size8 = 100000000
    power_max_size8 = 0
    power_count_size8 = 0
    power_summatory_size8 = 0

    power_min_mode1 = 100000000
    power_max_mode1 = 0
    power_count_mode1 = 0
    power_summatory_mode1 = 0
```

Figure 6.3: Variables initialization example for energy consumption data reading

The results will be read from the *.out* raw result's container file, and it will be formatted as a *.res* file, following the formula, one more time, it is used on the frequency case. The result will be written as it can be shown on the next Figure 6.4, showing only examples of AVX and size8 and mode1 of VNNI behaviour.

```
**************************
Energy During Test

Min energy avx: 0.10 kJ
Max energy avx: 2370.19 kJ
Average energy avx: 539.50 kJ
Min energy size8: 0.07 kJ
Max energy size8: 3414.15 kJ
Average energy size8: 380.86 kJ
Min energy mode1: 0.07 kJ
Max energy mode1: 1310.99 kJ
Average energy mode1: 147.74 kJ
**************************
```

```
**************************
Power During Test

Min power avx: 0.04 kW
Max power avx: 56.94 kW
Average power avx: 11.63 kW
Min power size8: 0.05 kW
Max power size8: 247.82 kW
Average power size8: 27.58 kW
Min power mode1: 0.05 kW
Max power mode1: 247.82 kW
Average power mode1: 30.18 kW
**************************
```

Figure 6.4: Energy and power result examples

7. CHAPTER

Analysis of the results

On this chapter, the different conclusions of the project will be presented and explained deeply. The different conclusion will be based on the comparison of the different test were done until now. They will look to get the best option and way of executing the VNNI instruction set on a CPU with AVX512 instruction set, and look how efficiently they can operate. It is also wanted to give a general view of the energy consumption of them, along with the analysis of the *Singularity* program usability, always emphasizing in the performance obtained with all the chances of using this VNNI instructions.

The conclusions will be given one by one separating different aimed comparisons on sections, being each section a comparison and little conclusion of what is being analysed. The comparisons will be gone from the behaviours, execution modes and sizes to a specific VNNI instructions, complementing all of them with the amount of usage cores, used energy, obtained frequency and time of the execution. For the last, a final conclusion will be done on the next chapter, providing an explanation that summarizes what it is reached and learned on the project.

The conclusions are based in all the results taken in the executions, but only few charts of the complete results were shown. The complete results will be available on the next GitHub [1] link, together with the complete files used on the project.

[1] https://github.com/G2Jezrien/AVX512-and-VNNI-instruction-set-performance-and-energy-consumption-benchmarking

7.1 AVX512 and VNNI comparison

AVX512 are more common and used instructions than VNNI ones nowadays, but the last Intel CPUs integrates the possibility to use them together, combining their functionality. AVX512 instructions are able to do easy mathematical operations, such as sums, divisions and reductions with a huge amount of numbers. VNNI instructions instead are able to execute some complex mathematical operations, as it was explained before in Section 5.1.3.

It was thought that executing this VNNI complex instructions will decrease the performance that the usage of AVX512 instructions usually have. That is why this comparison was done, looking for if the performance difference between the base and more usually used AVX512 and VNNI+AVX512 instructions is something to be considered. The next Figure 7.1 will help to explain the conclusion taken about their performance, as in all the generated charts, three quadratic equations will be shown (Poly max, min and avg) along with another three lines of the same colour for raw results values (Raw max, min and avg). Red colour describes the maximum, green colour the minimum, blue colour the average reached values.

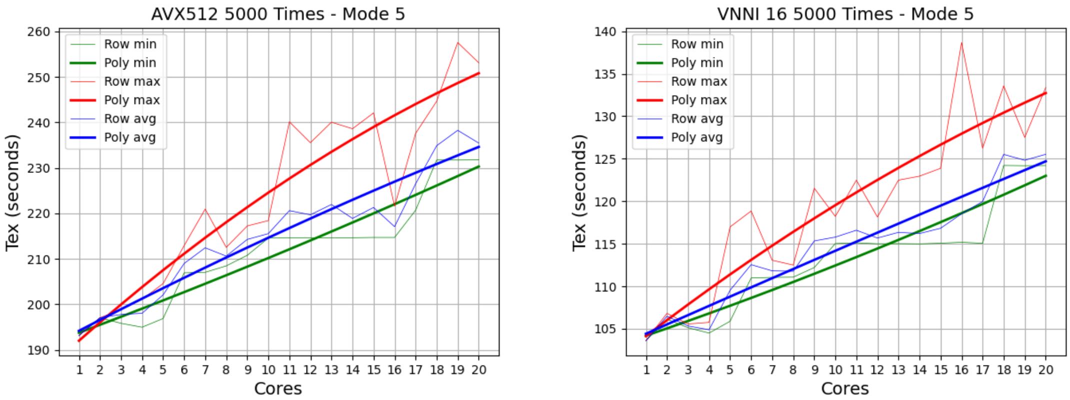

Figure 7.1: AVX512 vs AVX512+VNNI execution time charts

As it can see, the charts are nearly identical, even in the raw values of them. In conclusion, the usage of the complex operations that carries VNNI instruction sets, does not make any considerable changes on the performance they could reach in comparison to more usual and less complex AVX512 instructions. The previous measures were done looking at the performance based on the execution time, but what can be said about the frequency values on each of them. For being in the same conditions of the last comparison, the charts that equals to the results viewed on the Figure 7.1 will be shown and analysed, as Figure 7.2.

In terms of frequency, less value is equal to less performance, or in other words, less frequency involves more time to execute something. This time, the results and conclusions taken about the Figures 7.2 is something unexpected, being base AVX512 instructions which get the lower values. This means that in terms of frequency, VNNI+AVX512 instructions get the best performance.

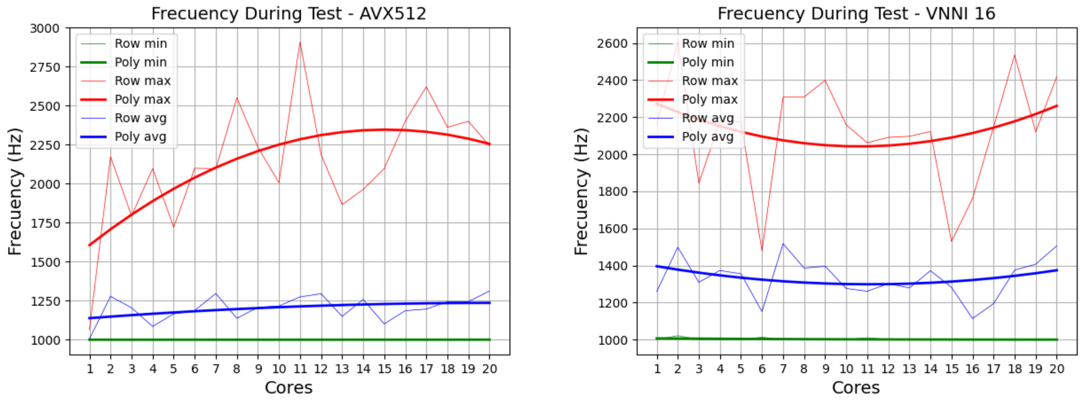

Figure 7.2: AVX512 vs AVX512+VNNI frequency charts

Continuing with the analysis of the instructions sets, let's take a look at the energy they consume on their respective executions. The more logical answer will be that the consumed energy will be alongside with the frequency performance they have, due to lower the frequency is, more is the CPU working, so more energy will consume. This analysis will be made based on the next Figures 7.3 and 7.4.

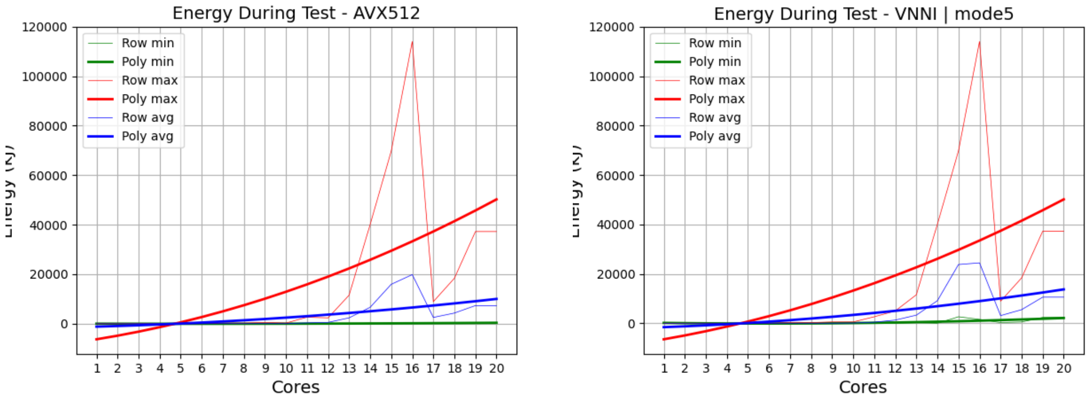

Figure 7.3: AVX512 vs AVX512+VNNI energy charts

Comparing the consumed energy in both behaviours of the benchmark, it is seen that the average obtained kJ consumption is higher on the VNNI instructions execution. The difference between AVX512 and VNNI is not too big, but it is something considerable, being the exact difference close to 5000 kJ on the most differentiable cases. In other words, AVX512 is more or less 33% more efficient in relation to the consumed energy during the *Zagreus* executions measured in Joules.

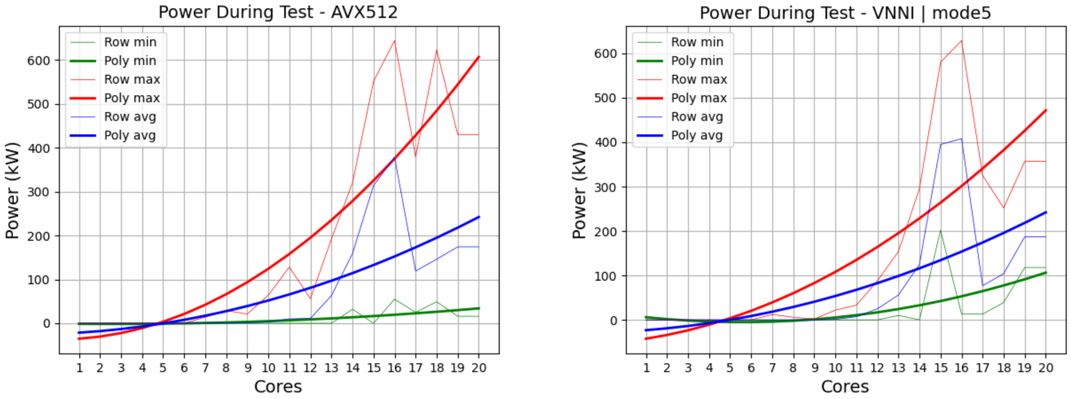

Figure 7.4: AVX512 vs AVX512+VNNI power charts

In the case of the measured power consumption, the average values in both of the behaviours have not almost any differences between them, as it can be seen on Figure 7.4. Nevertheless, the minimum used power is always higher on the VNNI execution cases, coming up with the conclusion that the minimal power necessary to use these instructions is higher. As the average energy consumption has to be more or less equal on both behaviours, the maximum power consumption is expected to be higher on the AVX512 executions. Maximum used power reaches 22% higher kW on the most differentiable cases of the AVX512 behaviour, fulfilling what it is expected.

A special mention to something that happens on all the obtained results. As you can see in the charts explained over this section, the value of the results for the usage of cores from 14 to 17 approximately is higher in nearly all the analyses done. As said, this happens for all the behaviours and their respective execution modes of *Zagreus*. It seems that both AVX512 and VNNI have some troubles at the moment of using this quantity of cores, maybe the explanation for this is that the communications needed to use more than 15 cores are best exploited using 18, 19 or 20 cores.

7.2 VNNI instructions comparison

7.2.1 Size configuration analysis

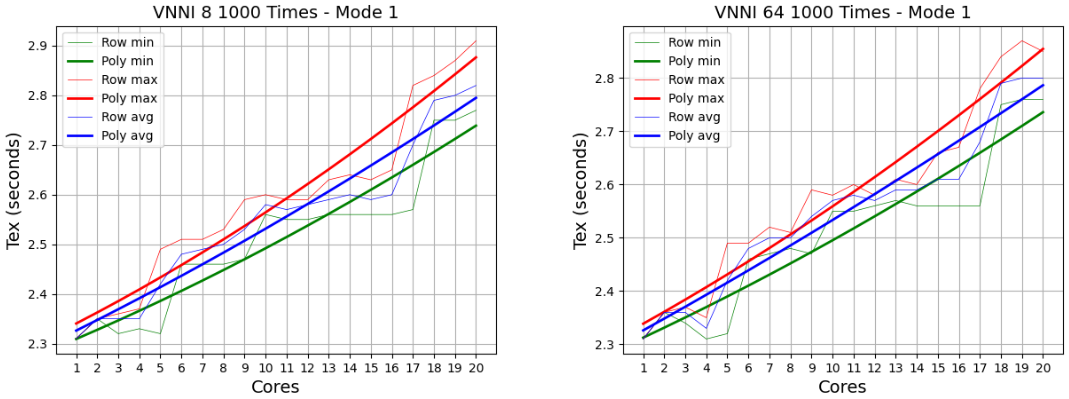

Figure 7.5: VNNI size execution time charts

As it can be seen in the Figure 7.5, the obtained maximum, minimum and average execution times of the VNNI behaviour test are almost exactly in all the size copnfigurations. The difference between using any size for filling the *_mm512* number type that use the VNNI instructions does not impact on their execution time performance. Exactly the same conclusion can be deduced from the charts that shown the frequency during the previous executions on Figure 7.6. In addition, supporting on the Figure 7.7, it can be stated that, even in terms of energy consumption, the fact that choosing different configurations of sizing does not really affect the performance of using the VNNI instruction set.

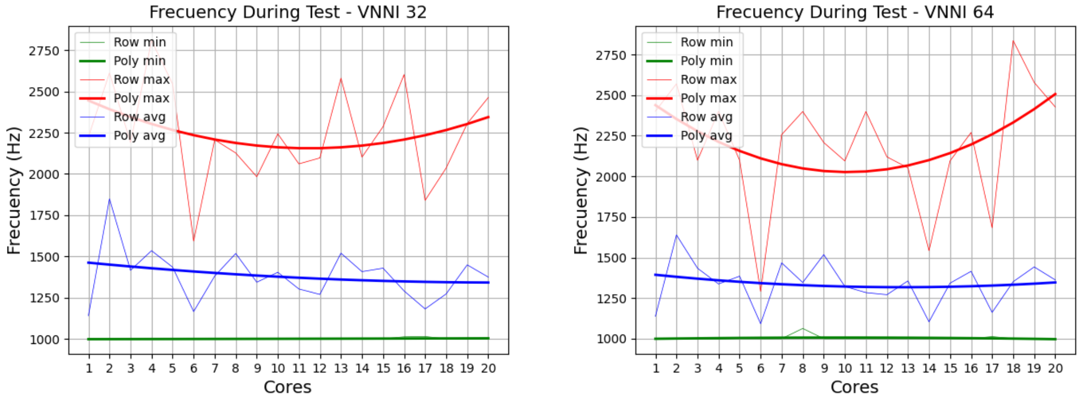

Figure 7.6: VNNI size execution time charts

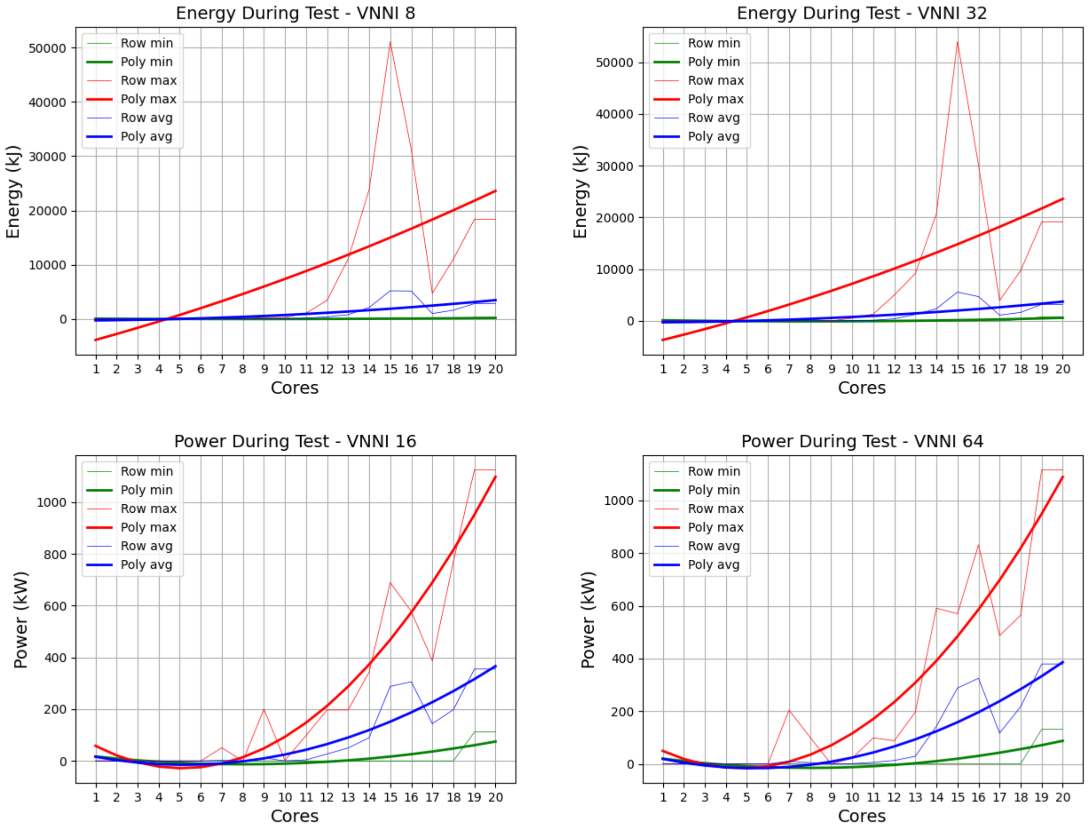

Figure 7.7: VNNI size energy charts

7.2.2 Execution modes analysis

As it was introduced and explained in Section 5.1.3, every execution mode from 1 to 4 of the VNNI benchmark refers to a concrete instruction of this set. On this comparison, it is wanted to analyse if these four instructions have any differences in the moment of executing them. All of them are prepared to do a similar mathematical calculation, being the most differentiable ones, the modes 1 and 2 with respect to modes 3 and 4. So the next results will be aimed to that difference, aiming at first to the execution time.

The below Figure 7.8, shows that the results from executing the VNNI behaviour as execution mode 1, which refers to executing the *_mm512_dpbusd_epi32(src, a, b)* instruction, and execution mode 4, which refers to executing the *_mm512_dpwssds_epi32(src, a, b)* instruction, are almost exactly to each other.

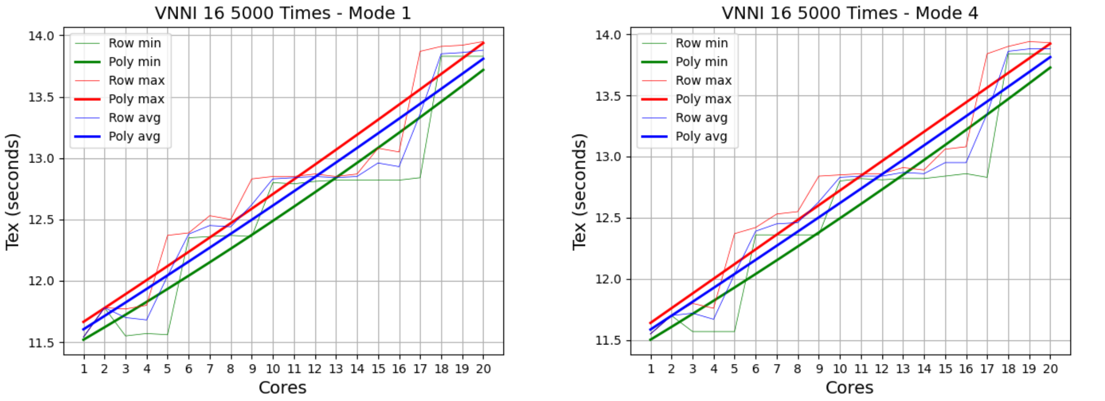

Figure 7.8: VNNI mode execution time charts

There is no considerable deviations between the results taken in both cases. It is not relevant which VNNI instruction will be executed, the performance is not going to be influenced by executing one or other instruction of the set. This conclusion gain more strength looking at the power and energy. The next chart on the Figure 7.9 will show that.

7.3 Analysis of the Zagreus execution inside Singularity

According to what it was explained in the previous Section 4.4 and its chapter, *Singularity* will be implemented to see if its use affects the performance of the VNNI instruction set executions. It was expected to get worst results by using *Singularity* because it brings other functionalities that did the execution more comfortable and versatile. As you can see on the Figure 7.10, the charts shows a difference of ten seconds on the same configuration between using and not using *Singularity*. The performance difference is more or less 8%. This difference is a considerable when talking about the programs that are executed on clusters. For example, on the execution of a three-hour program, an 8% means that it is necessary 14.4 minutes more to execute the same program inside Singularity container. This may not be seen as a huge difference, but in the example of executing a program that needs a year to be completed, the difference between executing inside Singularity or not is a month that is a lot of extra time.

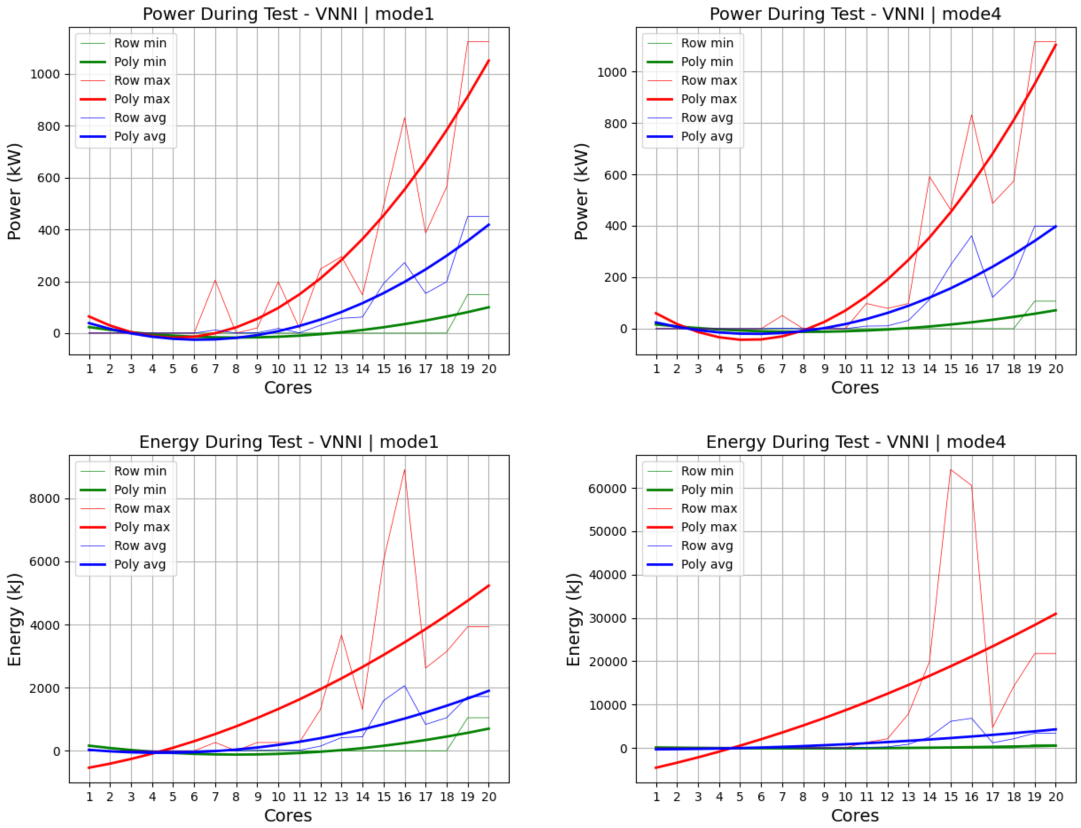

Figure 7.9: VNNI mode execution energy and power charts

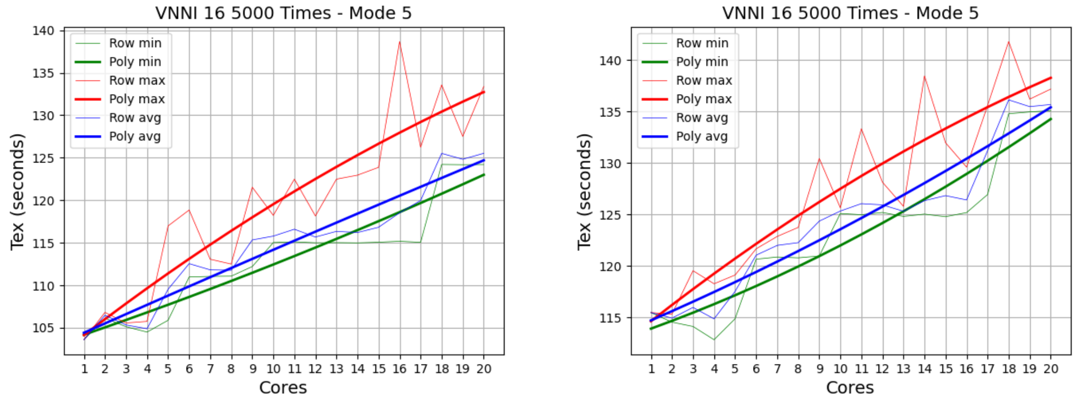

Figure 7.10: *Singularity* usage comparison charts

8. CHAPTER

Final conclusion and future work

The creation of this project was defined to measure and analyse the performance that the last Intel Cascade Lake series processors could reach with the usage of VNNI instruction set. In the course of the development of this project, a benchmark called *Zagreus* was self created to measure and test these instructions on the stated processors. Afterwords, the results obtained by executing *Zagreus* benchmark in different ways were formatted with another self created program, this time, the one named *get_results.py*. Some chart were generated assigning as a basis the formatted results, as such concluding with their support the following.

With regard to the comparison of the performance of the usually more used AVX512 instruction set, this VNNI instructions reach better performance results even tough the calculations these VNNI instructions do are more complex than the AVX512 ones. Referring to the amount of manners of executing the *Zagreus* benchmark, it was concluded that the size of filling the *_m512* numbers used on the VNNI instructions is not relevant to their performance. In addition, as well as the previous sizing, the four instructions inside the VNNI set are equal in terms of performance.

Notwithstanding the fact that the analysis done until this time of project were not so relevant, it was discovered that the different amount of cores using on the execution really impact in their performance. These configurations being between the 15 and 17 cores are the worst performing ones. Furthermore, it is also observed that using *Singularity* containers to execute the benchmark alters its efficiency. So, it is not worth using this *Singularity* technology due to the differences in the execution time of using them or not.

Personally, this project has helped me in my growth and improvement on being the best Computer Engineer I could be. I have learned to approach a project of this calibre taking into account the unforeseen events that may occur in its progress, which could make it deviate less from its path in the search of the final purpose. This end of degree project, whose shape was given by four years worth of hard work and study hours, set the end of one of the most important phases of my life. It also gives me the possibility to put the conceptual, practical, and academic skills acquired throughout these four years into practice. During this learning period, I have been capable of resolving all kinds of issues related to my future job, as well as granting access to the working life as a Computer Engineer.

It is imperative that research is not only focused on exploring what it was created for, besides being open to further research and analyses. In brief, it is not enough to just scratch the surface, but it is sought to delve and elaborate as much as possible on the theme. In case someone desires to continue investigating about this topic, some possible future work will be introduced in the next paragraphs.

The first line of investigation could be the comparison between VNNI instruction set usage on the processors capable of executing them and any kind of GPU that is also able to execute the set. The most important thing on this future work has to be the creation of a program that simulates how the VNNI instructions exactly work on a processor in a GPU. With that, both technologies run the same instruction the same way in order to be able to compare them on the best possible conditions.

On the other hand, as it was discovered on the development process of this project, the effect of the use of 15 to 17 cores executing VNNI instructions could also be worth researching. The performance this core amount gained in comparison to the other is divided by a huge difference. In this case, the main aim of this investigation would be to deepen on how the CPU manage the communications done between all the cores, so conclusions can be drawn to explain why the underperformance of this core amount happens.

Bibliography

[1] Avx512. https://www.intel.es/content/www/es/es/architecture-and-technology/avx-512-overview.html.

[2] Avx512 3rd gen. https://www.intel.com/content/dam/www/public/us/en/documents/product-overviews/dl-boost-product-overview.pdf.

[3] Containers. https://www.icot.es/introduccion-a-los-containers/.

[4] Intel intrinsics. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[5] Matplotlib. https://matplotlib.org/.

[6] msr interface. https://man7.org/linux/man-pages/man4/msr.4.html.

[7] Neural network. https://www.analyticsvidhya.com/blog/2021/05/beginners-guide-to-artificial-neural-network/.

[8] perf event interface. https://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[9] Powercap. https://www.kernel.org/doc/html/latest/power/powercap/powercap.html.

[10] Rapl program. https://web.eece.maine.edu/~vweaver/projects/rapl/.

[11] Singularity. https://sylabs.io/guides/2.6/user-guide/introduction.html.

[12] Slurm workload manager. https://slurm.schedmd.com/documentation.html.

Appendices

A. APPENDIX

Execution result formatting and chart generation

Once the *Zagreus* benchmark was executed, some *.out* type files were generated. Every execution will have its own output file. Thus, the program will link all these outputs into another new file, and generate different charts with all the read data. In order to visualize this results in a better way and format and then generate some charts from them, a *Python* based script has been developed and implemented. This script has the main function of taking all the previously generated *.out* files and format them, how it does is what is going to be explained in this appendix chapter. The explanations will be supplemented by parts of the code of the script that accords to the explanation is giving at the moment, and all the code references will contain the same line numbers as in the original script.

A.1 *get_results.py* definition and explanation

get_results.py is the name of the script that generate the formatted results and different charts is separated on two parts. First of the will be getting the raw results, which are generated without a very good visualization. So it will take all the *.out* file one by one, and it will calculate some statistical parameters from them. This will be explained deeper on a subsection later on this appendix chapter.

The second part of the script, consist of taking the formatted results generated with the previous part of the program and plot all of them on a more comfortable view. This is going to be done with the help of some *Python* packages and mathematical expressions, which will allows the script to generate plots with statistics and round them on a linear rounded equations.

A.1.1 Result taking and formatting code part

As mentioned before, this part of the program will read the results of the *Zagreus* benchmark, and supporting with some private variables will create some mathematical statistics. The selected statistics are the minimum, maximum and average of the time spend on each configuration of the benchmark test. The next Figure A.1 is an example of a view of a raw results of the benchmark test.

```
#AVX mode
#1 mode
#500 times
1.93s
1.99s
2.00s
2.00s
2.00s
2.00s
2.02s
2.59s
2.71s
2.72s
#Frequency
CPU MHz:             1933.661
#1000 times
3.89s
3.93s
3.95s
3.96s
3.97s
3.97s
3.97s
3.97s
5.10s
5.40s
#Frequency
CPU MHz:             1919.666
```

Figure A.1: .out result type file example

This example only shows the first two configurations of the complete file, but it is enough to see how the result are generated and previously will be taken and formatted. As you can see, different hints will be described with the use of the # character, with the purpose of facilitate the different sections of the results.

On this concrete example, a *AVX512* mode behaviour was launched, first five hundred times and hereinafter one thousand times. It is shown also that it was executed with ten cores of the CPU, due to ten time execution results, measured on seconds, were saved from one section to other, which ends with the current frequency of the CPU on the execution of the configuration measured on MHz.

Once the examples of the raw result are shown, it is feasible to start explaining how the Python script takes this results step by step. First of all is to ensure about the environment is ready for work, so the necessary directories, where the output is going to be stored, will be created. Some variables will also be initialized for saving the data while the formatting is being done. These variables will be *int* type numbers, as it can be seen on the next Figure A.2:

```
        mode = 1
        summatory = 0
        lines = 0
        minimum = 100000000
        maximum = 0
        frec_num = 0
```

Figure A.2: *get_results.py* time variables initialization

- *mode:* will save what mode of the behaviour the result is referencing to, and it will change from 1 to 5 values, starting from the value 1, that is, mode 1.
- *summary:* saves the sum of the time values of a test configuration, to calculate the average time with *lines* and initialized as 0.
- *lines:* saves the number of the time result lines read, to calculate the average time with *summary*, and it starts as 0.
- *minimum:* saves the minimum execution time for each configuration of the benchmark, initialized as 100000000, due to, any execution time will be less than that value, and that simplifies the code.
- *maximum:* saves the maximum execution time for each configuration of the benchmark, starting at 0.
- *frec_num:* saves the current frequency value position on the test, for getting easier to separate correctly frequency results, initialized as 0.

On the same way that these variables work, there are more variables used to save the frequency maximum, minimum and average during the different stages of the test. One for minimum counting, initialized as the time variable on 100000000, another one for maximum and two variables for average, named as, count and summatory. There will be four of these variables for the first behaviour, and each size value of the second behaviour of *Zagreus*. This is the Figure A.3 code part which reference to their initialization.

```
        frec_min_avx = 100000000
        frec_max_avx = 0
        frec_count_avx = 0
        frec_summatory_avx = 0

        frec_min_size8 = 100000000
        frec_max_size8 = 0
        frec_count_size8 = 0
        frec_summatory_size8 = 0

        frec_min_size16 = 100000000
        frec_max_size16 = 0
        frec_count_size16 = 0
        frec_summatory_size16 = 0

        frec_min_size32 = 100000000
        frec_max_size32 = 0
        frec_count_size32 = 0
        frec_summatory_size32 = 0

        frec_min_size64 = 100000000
        frec_max_size64 = 0
        frec_count_size64 = 0
        frec_summatory_size64 = 0
```

Figure A.3: *get_results.py* frequency variables initialization

At this point, the Python program will start getting all the data, alternating from getting execution times and frequency values from each configuration done on the whole test execution. This is where the hints written after with # makes sense. The raw results will be read line by line, and each line will be categorized by the string value read after # character with Pythons *startwith()* function.

This function returns true if the line it is reading on the moment starts with the same string that it is passed as parameter. With that, the script is able to different sections of time and frequency. For example, if it reads *#X mode* followed with *#Y times*, where X is any *exec_mode* and Y any *command_num*, it knows that the next lines until reading *#Frequency* are referencing to the execution times of X and Y configuration.

However, the section between reading *#Frequency* and *#X mode*, where X is any *exec_mode*, consist of a single line and references to the frequency obtained as a result of execute the previous X and Y configuration. In some parts of this information taking, a descriptive string will be written to so the last result format is clearer. On the next page with Figure

A.4, there are shown the code parts that are equivalent to the execution time and frequency taking, with the addition of how the variables that we explained before are used.

```
        for line in file:
        if line.startswith("#"):
            if line.startswith("#1 mode"):
                mode = 1
                frec_num+=1
                res.write("\n\n\n(500M) Min Max Avg (1000M) The Min Max Avg (2000M) Min Max Avg (5000M) Min Max Avg\n")

            elif line.startswith("#2 mode"):
                mode = 2
                res.write("\n")

            elif line.startswith("#3 mode"):
                mode = 3
                res.write("\n")

            elif line.startswith("#4 mode"):
                mode = 4
                res.write("\n")

            elif line.startswith("#5 mode"):
                mode = 5
                res.write("\n")

        else:
            number = float(line.split("s")[0])
            if number < minimun:
                minimun = number
            if number > maximun:
                maximun = number
            sumatory+=number
            lines+=1
```

Figure A.4: *get_results.py* execution time taking

The highlight of this part of the code, is to see how the variables created to generate the statistics of time taking are used. *number* variable will save the execution time is reading at the moment. The *minimum* variable only saves a new value if the execution time is reading on the moment is lower than the actual *minimum* value. The same happens with *maximum*, but just if the number is higher than the actual value.

summatory and *lines* instead, works on a different way. Each of them will be increasing their value for each time execution line read. These variables will be used to calculate the average of the execution time, in case that it is using more than one core, and will be written alongside *maximum* and *minimum* using the format described on the below Figure A.5 code lines:

The frequency taking it is done on another section of the *get_result.py* code, but just after the time taking. As it can see on the next Figure A.6, frequency minimum, maximum and

```
        res.write("Mode" + str(mode) + "," + str(minimun) + ","
        + str(maximun) + "," + str("{:.2f},".format(sumatory/lines)))
```

Figure A.5: *get_results.py* average calculation and time variables writing

average variables are used on the same way as it does on time taking. It has to be also mentioned, that this code part repeats for every variable described on Figure A.3.

```
        elif line.startswith("CPU"):
            sumatory = 0
            lines = 0
            minimun = 100000000
            maximun = 0

            if frec_num == 1:
                frec = float(line.split(":")[1].rstrip(" "))
                if frec < frec_min_avx:
                    frec_min_avx = frec
                if frec > frec_max_avx:
                    frec_max_avx = frec
                frec_count_avx+=1
                frec_summatory_avx+=frec
```

Figure A.6: *get_results.py* execution frequency taking

On this section of the script, the variables for time taking are reset with initialization values as well, due to they are prepared for reading properly the next configuration times. As the final step of this part of the Python program, the values of each frequency section variables will be written using the format that is described on the next Figure A.7 lines, leaving all prepared to generate charts from, the formatted results.

```
        res.write("Frecuency During Test\n")
        res.write("\nMin frecuency avx: " + str(frec_min_avx)+" Hz")
        res.write("\nMax frecuency avx: " + str(frec_max_avx)+" Hz")
        res.write("\nAverage avx: " + str("{:.2f} Hz".format(frec_summatory_avx/frec_count_avx)))
```

Figure A.7: *get_results.py* execution frequency taking

A.1.2 Chart generating part

The chart generation will be done using the *matplotlib.pyplot* [5] Python package. It will be working along with *numpy*, another Python package created for making some mathematical functions. In this case, second degree equations were going to be used, which will create the visual estimations lines for the growth of time and frequency values, plotted on the charts in addition to their exact values.

A chart will be created for each configuration available and executed on *Zagreus* benchmark, gathering the results obtained from all the amount cores executions of that configuration. So a hundred charts of execution time results will be generated for each benchmark execution and five charts for the frequency measured on them. Each of the charts will be made with setting the number of the cores on the X axis and the value want to be plotted on the Y axis (Frequency or time). The type of results will be printed, referencing to each of them with one colour.

Red will describe the results of the *maximum* statistic, and will be plotted as two different lines. The thinner one will show the exact values of the variable on each core amount execution and the thicker one the second degree equation created with that exact values. On the same way, another two lines will be plotted for *minimum* with green colour and *average* with blue colour. An example of a generated chart is given in the next section as Figure A.10. In addition, the next Figure A.8 shows the values read from the formatted results and how the equations are calculated.

```
    def generate_chart_avx(x, y, z, times, mode, node):
        name = 'AVX512 ' + str(times) + ' Times - Mode ' + str(mode)
        minimun = []
        maximun = []
        average = []
        cores = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

        for core in range(1, 21):
            line_num = 0
            file = open("res/zagreus_node" + str(node) + "_" + str(core) + "_cores.res", "r")
            for line in file:
                if line.startswith("Mode"+ str(mode)):
                    if line_num == 0:
                        minimun.append(float(line.split(",")[x]))
                        maximun.append(float(line.split(",")[y]))
                        average.append(float(line.split(",")[z]))
                        line_num += 1
            file.close()

        coefficients = np.polyfit(cores, minimun, 2)
        poly = np.poly1d(coefficients)
        new_minimun = poly(cores)

        coefficients = np.polyfit(cores, maximun, 2)
        poly = np.poly1d(coefficients)
        new_maximun = poly(cores)

        coefficients = np.polyfit(cores, average, 2)
        poly = np.poly1d(coefficients)
        new_average = poly(cores)
```

Figure A.8: *get_results.py* chart equations creation

As repeated previously in the explanation of how this script works, this is only an example of how one configuration chart is generated. After calculation of the necessaries and useful equations, the six lines that make up the chart will be plotted, as well as the title, name of the axis, grid and other chart configurations shown in Figure A.9.

```
        plt.plot(cores, minimun, color='green', linewidth=0.5, label='Row min')
        plt.plot(cores, new_minimun, color='green', linewidth=2, label='Poly min')
        plt.plot(cores, maximun, color='red', linewidth=0.5, label='Row max')
        plt.plot(cores, new_maximun, color='red', linewidth=2, label='Poly max')
        plt.plot(cores, average, color='blue', linewidth=0.5, label='Row avg')
        plt.plot(cores, new_average, color='blue', linewidth=2, label='Poly avg')

        plt.title(name, fontsize=14)
        plt.xlabel('Cores', fontsize=14)
        plt.ylabel('Tex (seconds)', fontsize=14)
        plt.grid(True)
        plt.legend(loc='upper left')
        plt.xticks(cores,cores)
        plt.savefig('charts/avx512/avx512_' + str(times) + 'M_mode_' + str(mode) + '.png')
        plt.close()
```

Figure A.9: *get_results.py* chart plotting

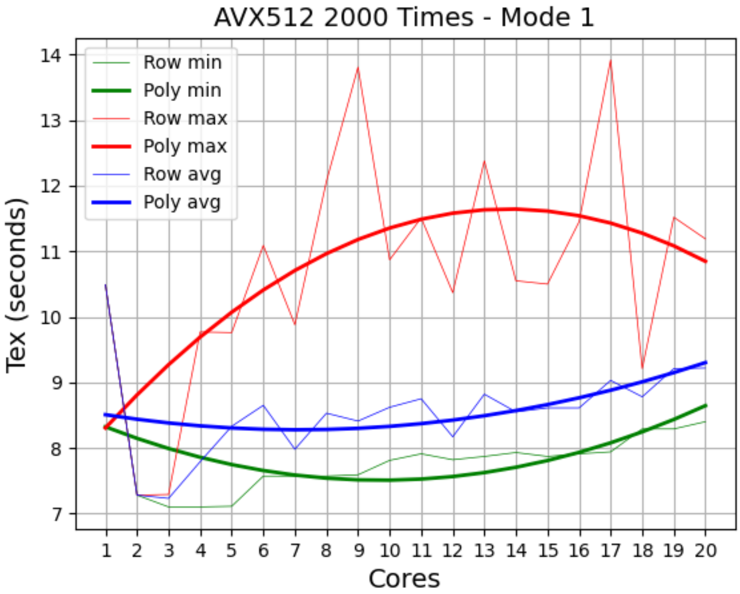

Figure A.10: Execution time generated chart example

B. APPENDIX

Energy consumption chart generating additions

Following the result formatting and generating for the new measures, the respective chart of them will be created. Likewise, the previous chart generation, this new charts will follow the same structure that it does on frequency Figure A.8 charts. But this time two different definitions will be done for each new measure, one that plots the different size performances and the other plotting different execution mode performances. The code for this is described on the next listings plotted as kW and kJ due to the high numbers are reached as J and W.

```
def generate_chart_power_vnni_mode(node, mode):
    name = 'Power During Test - VNNI | mode' + str(mode)
    minimun = []
    maximun = []
    average = []
    cores = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

    for core in range(1, 21):
        file = open("res/zagreus_node" + str(node) + "_" + str(core) + "_cores.res", "r")
        for line in file:
            if line.startswith("Min power mode" + str(mode) +  ": "):
                    minimun.append(int(line.split(": ")[1].split(".")[0]))
            if line.startswith("Max power mode" + str(mode) +  ": "):
                    maximun.append(int(line.split(": ")[1].split(".")[0]))
            if line.startswith("Average power mode" + str(mode) +  ": "):
                    average.append(int(line.split(": ")[1].split(".")[0]))
        file.close()

    coefficients = np.polyfit(cores, minimun, 2)
    poly = np.poly1d(coefficients)
    new_minimun = poly(cores)

    coefficients = np.polyfit(cores, maximun, 2)
    poly = np.poly1d(coefficients)
    new_maximun = poly(cores)

    coefficients = np.polyfit(cores, average, 2)
    poly = np.poly1d(coefficients)
    new_average = poly(cores)

    plt.plot(cores, minimun, color='green', linewidth=0.5, label='Row min')
    plt.plot(cores, new_minimun, color='green', linewidth=2, label='Poly min')
    plt.plot(cores, maximun, color='red', linewidth=0.5, label='Row max')
    plt.plot(cores, new_maximun, color='red', linewidth=2, label='Poly max')
    plt.plot(cores, average, color='blue', linewidth=0.5, label='Row avg')
    plt.plot(cores, new_average, color='blue', linewidth=2, label='Poly avg')

    plt.title(name, fontsize=14)
    plt.xlabel('Cores', fontsize=14)
    plt.ylabel('Power (kW)', fontsize=14)
    plt.grid(True)
    plt.legend(loc='upper left')
    plt.xticks(cores,cores)
    plt.savefig('charts/power/power_mode' + str(mode) + '.png')
    plt.close()
```

Figure B.1: Power chart generation code for execution mode comparison

```python
def generate_chart_ener_vnni(node, size):
    name = 'Energy During Test - VNNI ' + str(size)
    minimun = []
    maximun = []
    average = []
    cores = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

    for core in range(1, 21):
        file = open("res/zagreus_node" + str(node) + "_" + str(core) + "_cores.res", "r")
        for line in file:
            if line.startswith("Min energy size" + str(size) +  ": "):
                    minimun.append(int(line.split(": ")[1].split(".")[0]))
            if line.startswith("Max energy size" + str(size) +  ": "):
                    maximun.append(int(line.split(": ")[1].split(".")[0]))
            if line.startswith("Average energy size" + str(size) +  ": "):
                    average.append(int(line.split(": ")[1].split(".")[0]))
        file.close()

    coefficients = np.polyfit(cores, minimun, 2)
    poly = np.poly1d(coefficients)
    new_minimun = poly(cores)

    coefficients = np.polyfit(cores, maximun, 2)
    poly = np.poly1d(coefficients)
    new_maximun = poly(cores)

    coefficients = np.polyfit(cores, average, 2)
    poly = np.poly1d(coefficients)
    new_average = poly(cores)

    plt.plot(cores, minimun, color='green', linewidth=0.5, label='Row min')
    plt.plot(cores, new_minimun, color='green', linewidth=2, label='Poly min')
    plt.plot(cores, maximun, color='red', linewidth=0.5, label='Row max')
    plt.plot(cores, new_maximun, color='red', linewidth=2, label='Poly max')
    plt.plot(cores, average, color='blue', linewidth=0.5, label='Row avg')
    plt.plot(cores, new_average, color='blue', linewidth=2, label='Poly avg')

    plt.title(name, fontsize=14)
    plt.xlabel('Cores', fontsize=14)
    plt.ylabel('Energy (kJ)', fontsize=14)
    plt.grid(True)
    plt.legend(loc='upper left')
    plt.xticks(cores,cores)
    plt.savefig('charts/energy/energy_size' + str(size) + '.png')
    plt.close()
```

Figure B.2: Energy chart generation code for size comparison

C. APPENDIX

Singularity containers

The implementation of Singularity will be done following the same criteria as in the RAPL explanation chapter. The project will end testing this technology along with *Zagreus*, which allow any program to be executed with a special container. It is supposed that *Singularity* technology usage does not low the performance of the CPU, neither the execution times and energy consumption. This chapter will contain all the steps taken to implement and use *Singularity* tool with *Zagreus* benchmark.

C.1 What is wanted to get

As it was explained and introduced on its own Section 4.4 in the *Preliminaries* chapter, this *singularity* technology is aimed to work with complex and demanding programs, without decreasing their performance and results. *Singularity* is based on the use of the container, as *Docker* does, offering all the advantage and versatility they provide.

Usually, the usage of the containers comes together with a decrease on the performance of the program it is being executed on them, and as more demanding the program is, more will decrease its performance. *Singularity* developers assure that the container they made for demanding programs does not have any impact in the execution of the program is going to be executed on it. Prove that what they offer works in the way they assure, is what is wanted to do with the execution of the benchmark test using *Singularity*, taking out conclusions from comparing the results obtained previously with the results will be obtained with *Singularity* implementation and usage.

C.2 Creation of RAPL launcher scripts

The only change it is necessary to do on the previous files and functionalities of the program to implement *Singularity*, is to change, one more time, the way the *Zagreus* benchmark will be executed. As it is done also with RAPL, new scripts will be created to separate the different ways to execute the benchmark, setting each launcher for each way of execution. First, two new launchers will be implemented, that will be created along its respective *.sbatch* file creation scripts. These two scripts will be called as *launcher_sing* and *launcher_rapl_sing*, created like it is seen on Figures C.1 and C.2:

```
#!/bin/sh
for i in {1..20}
do
    ./launcher_sing_info $i $1 > slurm.sbatch
    sbatch slurm.sbatch
done
```

Figure C.1: *launcher_sing* script description

```
#!/bin/sh
for i in {1..20}
do
    ./launcher_rapl_sing_info $i $1 > slurm.sbatch
    sbatch slurm.sbatch
done
```

Figure C.2: *launcher_rapl_sing* script description

The scripts launch *Zagreus* benchmark with *Singularity*, one of them using the first version of the program without energy measures and the second one with RAPL implemented too. With that, it is possible to compare the obtained results from two different sources, making the conclusion of their comparison more reliable.

```
        for k in ${dolar}(seq 1 ${dolar}CORE)
        do
            priscilla exec ./Zagreus/main 1 ${dolar}i 16 ${dolar}j &
        done
```

```
        for l in ${dolar}(seq 1 ${dolar}CORE)
        do
                priscilla exec ./Zagreus/main 2 ${dolar}i ${dolar}j ${dolar}k &
        done
```

Figure C.3: *launcher_sing_info* script description

```
for k in ${dolar}(seq 1 ${dolar}CORE)
do
    ./Zagreus/rapl_logger.sh priscilla exec ./Zagreus/main 1 ${dolar}i 16 ${dolar}j &
done
```

```
for l in ${dolar}(seq 1 ${dolar}CORE)
do
        ./Zagreus/rapl_logger.sh priscilla exec ./Zagreus/main 2 ${dolar}i ${dolar}j ${dolar}k &
done
```

Figure C.4: *launcher_rapl_sing_info* script description

As it is plain to see on the figures above, the way of executing any program with *Singularity* is to call to *priscilla exec* alias just before the execution command of what it is wanted to be launched. This alias executes what is wanted to be executed with *Singularity* inside the container. After launching on these two different modes, the results will be saved on a *.out* file. Like it is done until now with the other *.out* files, they will be formatted with the *get_result.py* Python program, generating a *.res* and their respective charts. In case of launching with RAPL too, the charts about the energy and power measures also will be generated. Due to the *.out* files will not contain any new information, it is not necessary to do any changes on the formatting program.