


Master in High Performance Computing

Power Efficiency of High Energy Physics Applications on CPU and GPU

Supervisor(s):
Domenico Giordano,
Filippo Spiga,
Ivan Girotto

Candidate:
Keshvi Tuteja

9th edition
2022–2023

Acknowledgements

I would like to thank my supervisors Dr. Domenico Giordano, Dr. Filippo Spiga and Dr. Ivan Girotto for providing me such an incredible opportunity to work at CERN and for their guidance and support. Thank you for giving me the freedom to follow different approaches and for giving valuable suggestions whenever I faced any issues. I express my sincere gratitude towards Gonzalo Menendez Borge and Dr. Andrea Valassi for being so patient and helpful whenever I had any doubts or needed any advice regarding the code, and for striking interesting discussions. I would like to thank the other group members for providing me a positive workspace. I couldn't have asked for a better place.
I express my gratitude towards Dr. Irina Davydenkova for giving pivotal suggestions. This acknowledgement would be incomplete without thanking my parents for always having my back and my sister who has been my support system since day one. I am also grateful for my friends for always being there.

Gratitude is all I have!

Contents

CHAPTER 1

Introduction

In the recent years, there has been a significant hike in electricity prices. Given the growing demand for computational resources at CERN and the rising electricity prices, it will be beneficial to enhance the power efficiency of the High Energy Physics (HEP) simulations being done at CERN. It is expected that the high luminosity phase of the LHC (as shown in fig. 1.1) will demand approximately 10x more computational resources than it demands currently. Hence, it is necessary to optimize the power and time efficiency of these simulations.

HEP simulations comprise of three main stages:

- **Event Generation**: In this stage, the particles produced after the collisions are generated according to the likelihood of their production.
- **Simulation + Digitization**: In this step, interaction of collision product with the detector is simulated which results in electronic signals.
- **Reconstruction**: Here, electronic signals are translated to the particles passing through the detector. This step is done for the simulated data as well as the real data (the data coming from LHC).

The simulated data is then compared with the real data. In this project, I was particularly interested in benchmarking the event generation step since it is relatively easier to vectorize event generation and hence, one can exploit architectures like GPU or even different CPU vectorizations. In this thesis, we have compared power efficiency of CPU and GPU while running the MadGraph application, encapsulated in a docker container.

We used an event generation application called MadGraph [15] for this project and used HEP-benchmark-suite [6] and HEPscore [7] [20] to benchmark it. We discuss more about these tools in chapter 2. In chapter 3, we discuss how one can run the suite and how the configuration of the suite and HEPscore can be customized to run the containerized MadGraph application. We also discuss about how to customize the configuraion to measure metrics of interest, for example: power consumption and load. Lastly, in chapter 4, the analysis of the results has been done for this benchmarking project.

Conclusions and open issues regarding the work presented in this thesis are discussed in chapter 5.

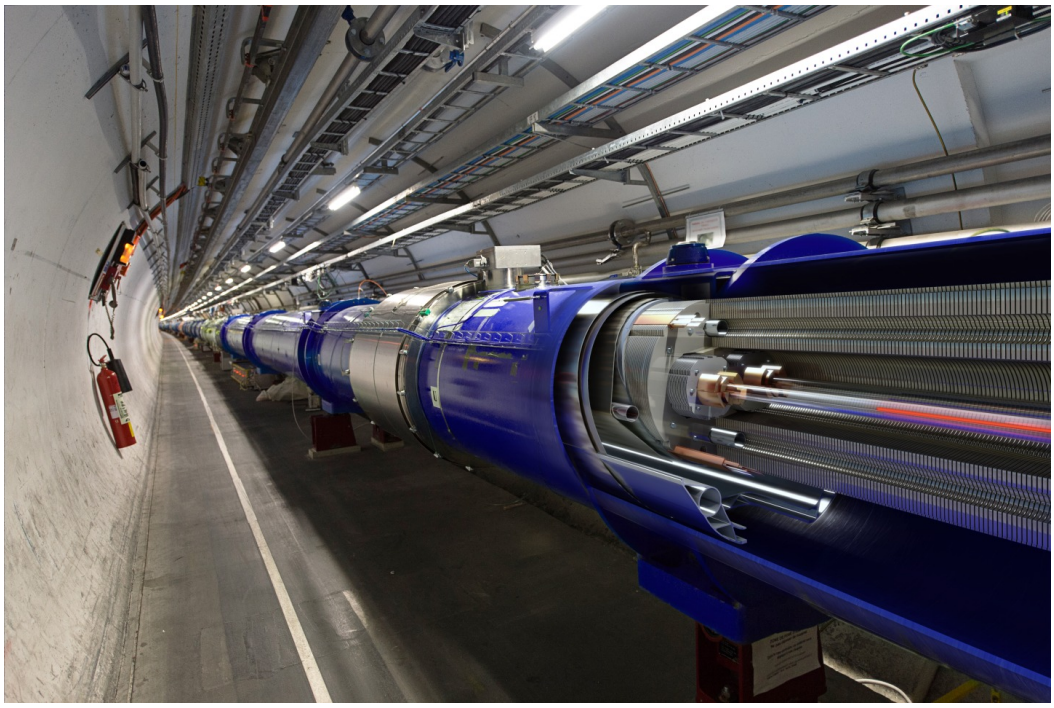

Figure 1.1: Large Hadron Collider [14]

CHAPTER 2

Formalism

2.1 Applications and tools used

In this section we discuss the tools that were used for benchmarking, namely, HEPscore, HEP-benchmark-suite and the MadGraph workload.

2.1.1 HEPscore

HEPscore is an application which orchestrates the execution of containerized workloads [8]. These workloads mimic the usage of WLCG resources. HEPscore can be configured to execute certain benchmark containers on different architectures. Currently, the application supports Docker/Singularity containers. Though HEPscore is designed to use the containers from the HEP-workloads project [8], it can still work with any Docker/Singularity container which conform to the HEP Workloads'output JSON schema.

Computation of HEPscore

The application computes the final score based on how time efficient certain benchmarks are on a given server.
For a given configuration, HEPscore is determined by computing the geometric mean of the performance scores while executing each workload included in the configuration. The performance scores typically represent the event throughput of the workload process. Here, event throughput is the "events" processed per second, where one "event" represents one LHC collision. This is done for a specific server, and then the

server is assigned a score, telling us how good/bad the machine is for the kind of jobs, the user is interested in.

To standardize the workload scores, each score is normalized relative to the score of the reference server, which is specified in the configuration settings under the key 'reference_machine'. For HEPscore23, the reference server is described as 'Intel CPU Gold 6326 CPU @ 2.90GHz - 64 cores SMT ON'.

After normalization, the scores are averaged using the geometric mean and then re-scaled according to the value indicated in the configuration settings under the key 'scaling'. This resulting value is then referred as HEPscore score.

HEPscore can be executed independently or using the HEP-benchmark-suite. Running it with the suite provides additional functionalities. More about it is discussed in the next section.

2.1.2 HEP-benchmark-suite

The suite is a toolkit capable of benchmarking various workloads for the HEP experiments running at CERN. It can be used for the following:

- Imitating the usage of WLCG resources for experiment workloads: This allows running imitation of applications running on WLCG .
- Allowing running the benchmarks on heterogeneous hardware: This makes it possible to use the Suite to run on GPU and CPU (ARM/ X86,..).
- Collecting hardware metadata and comparing various platforms: This allows the user to compare the outcome of the becnmark under the similar conditions.
- Having prompt feedback and being able to publish results: The suite provides the feature to directly publish the results in a database at CERN.
- Probing resources on a cluster or on a cloud: This feature allows to suggest deletion and re-provisioning of under-performing resources.

Figure 2.1 shows the high level architecture of the suite.The suite is capable of running HEPscore (along with the other benchmarks) and it also provides support for additional plugins. We used energy measurement plugins to determine various metrics like: power consumption, load, memory usage, etc. HEP Benchmark suite workflow can be described best by fig 2.2. Here, various servers from different data centres are benchmarked using the suite and their results comprising of a json summary are published to an AMQ broker. This json file contains HEPscore results and plugin observations besides many other metrics of interest. Metadata (such as

UID, CPU architecture, OS, Cloud name, IP address, etc.) are also included into the searchable results. These results can then be analyzed and visualized by researchers, site managers, etc.

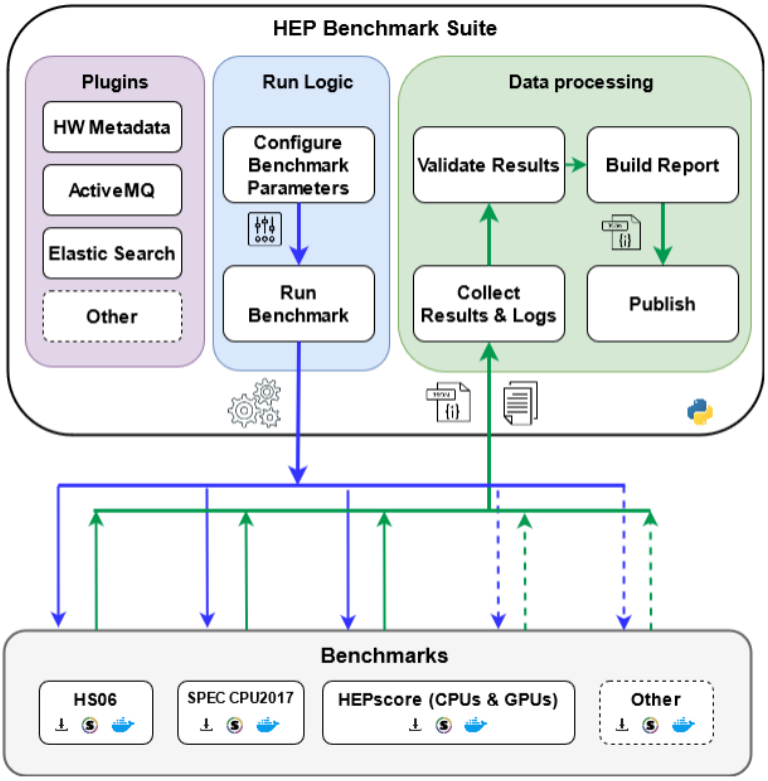

Figure 2.1: High Level architecture of the HEP-benchmark-suite [6]

2.1.3 MadGraph

MadGraph event generator software

MadGraph is an event generator application used in the realm of particle physics. It calculates matrix elements for the Feynman diagrams that one is interested in. The matrix element is related to the probability of the occurrence of the physical process. These matrix elements are then used to calculate the cross sections, which represent the likelihood of the process happening in a collision. Cross sections are essential for predicting how often a particular process will be observed in experiments. After the cross sections are calculated, MadGraph can be used to simulate event generation. It generates random events based on the calculated probabilities and

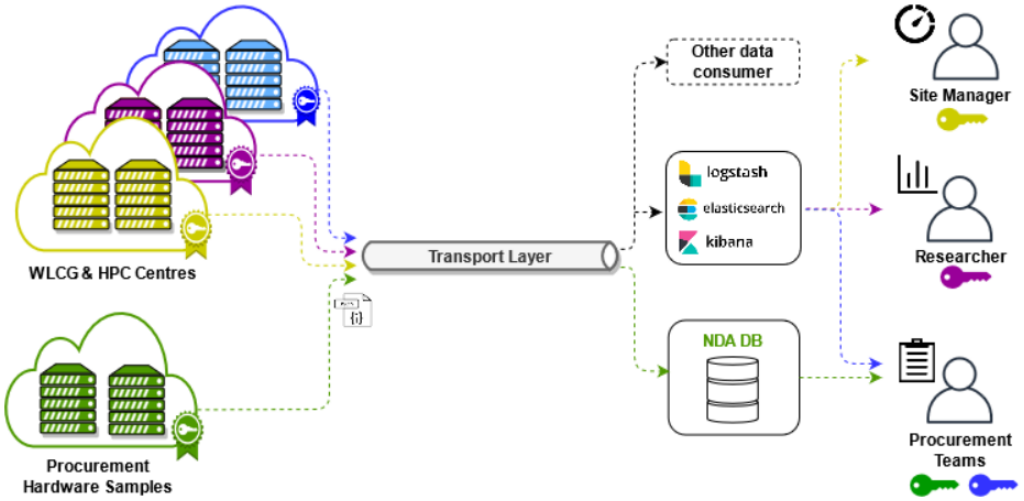

Figure 2.2: HEP-benchmark-suite workflow [6]

kinematic constraints. Each event contains information about the particles produced and their momenta. Researchers can analyze the generated events and compare them with experimental data or theoretical predictions. By comparing the simulated events with the real data, scientists can test the validity of the Standard Model or look for deviations that might indicate new physics beyond the Standard Model.

The MadGraph workload

The MadGraph workload is based on a (not yet publicly released) version of the MadGraph event generator software which has been ported to GPU and CPU platforms (vector instructions). Note that the reference [16] mentions that *The "sa" benchmark is currently based on a standalone application, using a simplified phase space sampling, which cannot be used in production by the experiments*.

The workload allows for various physics processes to be executed in parallel: eemumu ($e^+e^- -> \mu^+\mu^-$), ggtt (g g $-> t\bar{t}$), ggttg (g g $-> t\bar{t}$ g), ggttgg (g g $-> t\bar{t}gg$). These processes differ in the relative importance of arithmetic intensity and memory access, for example, 'ggttgg' is the most computationally intensive process. And therefore, 'ggttgg' benchmark is executed by default.

The MadGraph workload [16] can be run on CPU as well as GPU. One can run the stand alone docker container on CPU by executing the following command:

Listing 2.1: Running the docker container for MadGraph on CPU

```
docker run -d gitlab-registry.cern.ch/hep-benchmarks/
hep-workloads/mg5amc-MadGraph4gpu-2022-bmk:v0.7
```

To run it on GPU:

Listing 2.2: Running the docker container for MadGraph on GPU

```
docker run --rm --security-opt=label=disable
--device nvidia.com/gpu=all gitlab-registry.cern.ch/
hep-benchmarks/hep-workloads/mg5amc-MadGraph4gpu
-2022-bmk:v0.7 --extra-args "--gpu"
```

In the above command, we are passing `gpu` as an extra argument. One could also run this container with the suite as done for the work presented in the thesis.

Command Line Interface (CLI) parameters

MadGraph workload supports the following extra arguments:

- `--cpu` : run only the C++ benchmarks on CPU (1 or more copies)
- `--gpu` : run only the CUDA benchmarks on GPU (1 copy)
- `--both` : run both the C++ benchmarks on CPU and the CUDA benchmarks on GPU (1 copy)
- `-eemumu` : $e^+ e^- --> \mu^+ \mu^-$ (low computational intensity, high overhead from memory access/copy)
- `-ggtt` : $g\ g --> t\bar{t}$ (low computational intensity, high overhead from memory access/copy)
- `-ggttg` : $g\ g --> t\bar{t}g$ (higher computational intensity)
- `-ggttgg` : $g\ g --> t\bar{t}gg$ (even higher computational intensity)
- `-dbl` : double precision 'd' benchmarks
- `-flt` : single precision 'f' benchmarks
- `-inl0` : benchmarks without C++ aggressive inlining
- `-inl1` : benchmarks with C++ aggressive inlining

- `-p` : '`-p<nblks>,<nthrs>,<niter>` for number of threads, blocks and iterations

The default is set to '`--both -ggttgg -dbl -flt -inl0`' (both CPU and GPU, ggttgg only, both d and f, inl0 only).

2.1.4 Vectorization

In the work presented in this thesis, we compared CPU and GPU. In the case of CPU, we used various vectorization techniques supported by x86 architecture. SSE4 [11], AVX2 [10], 512Y, and 512Z [9] are all extensions to the x86 instruction set architecture used in modern CPUs. They provide additional instructions and capabilities to enhance performance in various tasks, particularly in areas like multimedia processing, scientific computing, and cryptography. Here's a brief overview of each:

- **SSE4 (Streaming SIMD Extensions 4)**: SSE4 is a set of SIMD (Single Instruction, Multiple Data) instructions introduced by Intel in 2006 with their Penryn microarchitecture. SSE4 includes several sub-extensions (SSE4.1 and SSE4.2), which offer enhancements such as string and text processing instructions, improved dot product calculations, and faster CRC32 checksum generation. SSE4 instructions are widely used in multimedia applications and some scientific computing tasks.

- **AVX2 (Advanced Vector Extensions 2)**: AVX2 is an extension of Intel's AVX instruction set, introduced with their Haswell microarchitecture in 2013. AVX2 provides additional SIMD instructions to accelerate floating-point and integer arithmetic operations, including vectorized multiply, accumulate, and permute operations. AVX2 is particularly beneficial for numerical computing workloads and is supported by many modern CPUs.

- **512Y and 512Z (AVX-512)**: AVX-512 is an advanced extension of the AVX instruction set, introduced by Intel with their Knights Landing and Skylake-X microarchitectures. AVX-512 extends the width of vector registers to 512 bits, allowing for even more parallelism in SIMD operations. The "Y" and "Z" in AVX-512 refer to different subsets or extensions of AVX-512 instructions, each providing additional capabilities and optimizations. AVX-512 is designed for high-performance computing workloads, including scientific simulations, artificial intelligence, and data analytics.

  These extensions play a crucial role in improving the performance of software applications that leverage parallelism, making them essential for various com-

puting tasks ranging from scientific simulations to multimedia processing and machine learning.

2.1.5 Docker

Docker [2] provides support to package and run an application in an isolated environment. This isolated environment is called a container. These containers are lightweight and contain all the dependencies required to run the application. The analysis we did was based on containers. Sharing containers ensures that everyone is working with the same application in the same environment. Figure 2.3 shows the Docker architecture. Docker provides support for easy and hassle free deployment of an application in the following manner:

- development of an application based on containers
- distributing and testing via containers
- deployment of application as a container

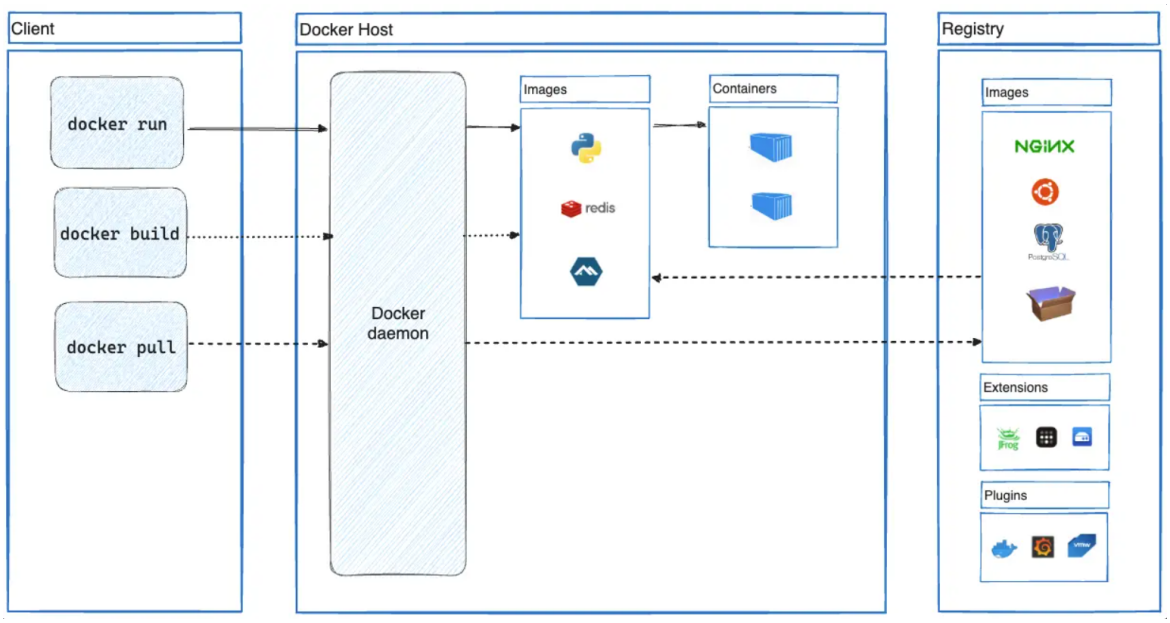

Figure 2.3: Docker architecture [2]

Docker versus Singularity

Like Docker, Singularity [19] is also one of the containerization technologies. Docker containers can also be converted to and or run from Singularity. We prefer singularity because it needs no root privileges. This feature makes Singularity containers more suitable for use in environments where users do not have administrative access to the host system. The HEPscore containers however are built with Docker, then we do convert them to Singularity and store the Singularity Image File (SIF) images. While Docker and Singularity both offer containerization solutions, they have different focuses and are suited to different use cases. Docker is widely used in various environments for its flexibility and extensive ecosystem, while Singularity is favored in HPC and research environments for its security features and compatibility with scientific computing workflows since Docker containers may not always be directly compatible with HPC environments due to differences in security policies and resource management.

Docker image

A Docker image serves as a fundamental building block in container technology, encapsulating an application and all its dependencies into a portable and self-contained package. These images are created using a Dockerfile, which defines the environment and configuration needed to run the application. Leveraging layered file systems, Docker images promote efficiency and reproducibility, enabling developers to easily share and deploy applications across diverse environments. Each image consists of a series of read-only layers, representing incremental changes made during the image's construction. This layered approach fosters rapid image deployment and efficient resource utilization, as only the modified layers need to be rebuilt when updating or customizing an image. Moreover, Docker images are stored in repositories such as Docker Hub [3], facilitating seamless distribution and collaboration within the developer community. The MadGraph containers, like all HEPscore containers, are available on a respository at CERN: "registry.cern.ch" as seen in listings 2.1 and 2.2.

Docker container

A Docker container is a lightweight, standalone, and executable software package that encapsulates an application along with its dependencies and runtime environment. Built from Docker images, containers offer a consistent and reproducible environment across different systems, enabling developers to package applications once and run them anywhere. Containers leverage operating system-level virtualization to

isolate processes and resources, providing efficient utilization of system resources and ensuring application portability. With Docker, developers can easily create, deploy, and manage containers using simple commands, streamlining the software development lifecycle. Containers facilitate microservice architectures, enabling applications to be broken down into smaller, modular components that can be independently developed, deployed, and scaled. Additionally, Docker containers promote scalability, agility, and consistency in application deployment, making them an essential tool for modern software development and deployment workflows.

Docker CLI

The Docker CLI is a robust tool for managing containerized applications and orchestrating Docker containers efficiently. With just a few simple commands, users can perform a wide range of tasks, from pulling images to launching and managing containers. For instance, to pull a Docker image from Docker Hub, you can use the docker pull command followed by the name of the image. Here's an example:

Listing 2.3: Pulling an image

```
docker pull nginx
```

This command downloads the latest version of the NGINX web server image from Docker Hub onto your local machine. Once the image is pulled, you can use the docker run command to create and start a container based on that image. For example:

Listing 2.4: Running a container in detached mode

```
docker run -d --name my-nginx -p 8080:80 nginx
```

This command creates a new container named my-nginx, maps port 8080 on the host to port 80 on the container, and runs the NGINX web server within the container. The -d flag tells Docker to run the container in detached mode, meaning it runs in the background. You can then use various other commands like docker ps to view running containers, docker stop to stop a container, and docker rm to remove a container, among others. With its simplicity and versatility, the Docker CLI streamlines the process of container management and empowers users to build, deploy, and manage containerized applications with ease.

CHAPTER 3

Implementation

The implementation of the benchmarking process involved several steps, including learning to run the HEPscore standalone, executing HEPscore with the suite, extending the configuration to measure power consumption, and more. This analysis is specifically focused on the MadGraph workload, considering both CPU and GPU performance. The process of analyzing can be broken down into the following steps, each of which is elaborated upon in detail in this chapter:

- **Running HEPscore with the suite:** While HEPscore can function independently, utilizing the suite offers additional advantages, therefore, it is recommended to run it with the suite.

- **Customizing the suite configuration for the MadGraph workload for CPU as well as GPU:** The suite allows for the simultaneous execution of multiple workloads, including MadGraph, but requires tailored configuration adjustments according to user's needs.

- **Extending the configuration of the suite to include the Energy measurement plugins:** A crucial aspect of our analysis involved integrating the plugins to measure energy consumption and other crucial metrics on both CPU and GPU platforms.

- **Publishing the results**: The benchmarking results can be published to the WLCG Benchmark Database, ensuring wider accessibility and transparency within the scientific community.

3.1 Running HEPscore with the suite

The suite can be executed using the `run_hepscore.sh` script [18] available on the suite repository. The script can be edited according to the user's needs. To run the script, a machine must have at least 20 GB of free hard disk space and the following prerequisites installed on the system:

- Apptainer/Singularity (version 1.1.6 or higher)
- Python version 3.9 or higher;
- python3-pip;
- git

To run the script, one can use the following command:

Listing 3.1: Running hep-score

```
./run_HEPscore.sh -s SITE
```

The above command executes the workloads in HEPscore. This bash script serves as a versatile tool for installing and executing the HEP-Benchmark-Suite, automating various configuration options and providing flexibility in deployment. Upon execution, the script first checks for necessary parameters, including site name, certificate details for publishing results, and container executor preference. It then proceeds to set up the suite environment, including creating a Python virtual environment and installing the suite components either from the repository or pre-built wheels.

The script provides options for installation only (-i flag), running the suite only (-r flag), or both. During execution, it ensures that no other instances of the suite are running and verifies available workspace's space. The suite is then run, with memory usage monitored throughout the process. Upon completion, the script handles various post-execution tasks, including creating a tarball for error analysis, prompting to send results to AMQ, and checking for memory usage anomalies.

The script is also responsible for reporting the results in the desired directory. The report also includes metadata about the server's running conditions. More information about the CLI parameters are given in the script itself. The results can also be published to the WLCG Benchmark Database.

HEPscore can also be executed as a stand-alone instead of being executed through the suite. To do that, one can use the python virtual environment and follow the following steps:

Listing 3.2: Installing hep-score

```
python3 -m venv HS23env
source HS23env/bin/activate
pip3 install git+https://gitlab.cern.ch/
hep-benchmarks/hep-score.git@v1.5
```

To run HEPscore, one can execute the following command:

Listing 3.3: Running hep-score standalone

```
hepscore -v PATH_TO_WORKDIR
```

3.1.1 Usage of hep-score Command

The `hep-score` command provides a versatile interface for benchmarking tasks, offering various options to customize execution and output. When invoked, it requires a positional argument specifying the base output directory where benchmark results will be stored. Additionally, it supports several optional arguments (one example is given below, rest are provided in Appendix A.1:

- `-m [singularity,docker]` or `--container_exec [singularity,docker]`: Allows the user to specify the container platform for benchmark execution, with Singularity being the default option.

Furthermore, the `hep-score` command provides usage examples for different scenarios:

1. Running benchmarks via Docker while displaying verbose information.
2. Running benchmarks using Singularity (default) with a custom benchmark configuration.
3. Listing built-in benchmark configurations.
4. Running with a specified built-in benchmark configuration.

Overall, the `hep-score` command offers a comprehensive set of options to cater to various benchmarking needs, making it a powerful tool in performance evaluation tasks.

Now that we have seen how to run and install the suite and HEPscore, let's look at how to customize its configuration in the next section.

3.2 Customizing the configuration

To run any of the workloads, one needs to configure the suite as well as the HEPscore application. A user might be interested in certain workloads only, or might want to pass different parameters while executing the workloads. For that, one can customize the configuration. Let's see how one can customize the suite and the HEPscore configurations and incorporate those in `run_HEPscore.sh`.

3.2.1 Custom configuration for the suite

We used the following configuration for the suite:

Listing 3.4: Configuration file for the benchmark suite

```
activemq:
  server: $SERVER
  topic: $TOPIC
  port: $PORT
  key: $CERTIFKEY
  cert: $CERTIFCRT

global:
  benchmarks:
  - hepscore
  mode: $EXECUTOR
  publish: $PUBLISH
  rundir: $RUNDIR
  show: true
  tags:
    site: $SITE
hepscore:
  config: $WORKDIR/$HEPSCORE_CONFIG_FILE
  version: v1.5
  options:
      userns: True
      clean: True
```

In this context, the `activemq` key serves a pivotal role in the publishing process. Within this key, the parameters `key` and `cert` are employed to denote the respective

paths leading to the key and certificate files necessary for authentication during result publication via the messaging service. These credentials, obtained upon request, play a vital role in securing the transmission of benchmarking data. The `benchmarks` field, nestled within the broader configuration framework, offers a versatile array of options. Here, benchmark names such as `hepscore`, `hs06`, `db12`, among others, can be specified, catering to diverse analytical needs and performance evaluation metrics. Similarly, the `mode` parameter within the configuration delineates the execution mode, affording flexibility between singularity, docker, and other modes as per operational requirements. Delving deeper into the `hepscore` section, one can use the `config` key by providing a path to the designated hepscore configuration file. This file, when utilized during execution, ensures adherence to specific configurations tailored to the benchmarking process.

3.2.2 Custom configuration for HEPscore

We used the following configuration file for `HEPscore` to run the MadGraph container on CPU:

Listing 3.5: Configuration file for HEPscore

```
hepscore_benchmark:
  benchmarks:
    mg5amc-MadGraph4gpu-2022-bmk:
      results_file: mg5amc-MadGraph4gpu-2022_
      summary.json
      ref_scores:
        ggttgg-sa-cpp-d-inl0-best: 1
        ggttgg-sa-cpp-d-inl0-none: 1
        ggttgg-sa-cpp-f-inl0-best: 1
        ggttgg-sa-cpp-f-inl0-none: 1
        ggttgg-sa-cpp-d-inl0-sse4: 1
        ggttgg-sa-cpp-f-inl0-sse4: 1
        ggttgg-sa-cpp-d-inl0-avx2: 1
        ggttgg-sa-cpp-f-inl0-avx2: 1
        ggttgg-sa-cpp-d-inl0-512y: 1
        ggttgg-sa-cpp-f-inl0-512y: 1
        ggttgg-sa-cpp-d-inl0-512z: 1
        ggttgg-sa-cpp-f-inl0-512z: 1
      weight: 1.0
```

```
      version: v0.7
      args:
        events: 100
        extra-args: "--cpu"
settings:
  name: MadGraph_Keshvi
  reference_machine: "CPU Intel(R) Xeon(R)
  CPU E5-2630 v3 @ 2.40GHz" registry: $REGISTRY://
  gitlab-registry.cern.ch/hep-benchmarks/hep-

  workloads$REGISTRY_SUFFIX
  method: geometric_mean
  repetitions: 3
  retries: 1
  scaling: 1
  container_exec: $EXECUTOR
```

In this section, we delve into the configuration details for executing the 'mg5amc-MadGraph4gpu-2022-bmk' benchmark. We specify that the results obtained from this benchmarking exercise should be stored in 'mg5amc-MadGraph4gpu-2022_-summary.json', ensuring accessibility and organization for subsequent analysis.

The `ref_scores` section plays an essential role for collecting essential performance metrics from the benchmark container output. The metrics mentioned under this section are collected and reported during the benchmarking process. Each sub-score within this section is associated with a reference score obtained from the specified `reference_machine`, facilitating a comparison and evaluation process. These sub-scores undergo normalization, where each is divided by its corresponding reference score. Subsequently, the geometric mean is computed from these normalized scores, resulting in the derivation of a comprehensive final score for the benchmark container. This approach ensures robustness and reliability in performance assessment.

Furthermore, the version and the weight for the container is specified in `version` and `weight` respectively.

The `args` keyword offers a level of customization, allowing users to pass additional arguments to the benchmarking process. In the provided configuration (Listing 3.5), we include `events:100` and `--cpu` as extra arguments, configuring the MadGraph container to execute with 100 events on the CPU.

Under the `settings` section, specifications are made regarding the reference machine, identifying it as the "CPU Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz".

Additionally, attention is given to specifying URLs for running the container from various repositories using the `registry` parameter, which supports options like Docker or Singularity.

Planning and consideration are evident in parameters such as `method`, `repetitions`, `retries`, and `scaling`, each designed to ensure control and accuracy in benchmark execution and result interpretation.

It's essential to note that the `ref_scores` may vary depending on the device and the specific process under investigation. The HEPscore configuration for running MadGraph on the GPU is provided in Appendix A.2, offering a comprehensive guide for GPU-based performance assessment.

This configuration can be seamlessly integrated into the `run_hepscore` bash script for benchmarking the workloads with the hep-benchmark-suite. For further insight into MadGraph ref-scores, documentation is provided in Appendix A.2.1.

3.2.3 Incorporating the changes in the execution script

Now that we have our configuration files for the HEPscore as well as the suite, we can incorporate these in the `run_hepscore.sh` bash script in the `hepscore_install()` function as follows:

Listing 3.6: Incorporating the custom configuration in `run_hepscore.sh`

```
        :
hepscore_install(){
        :
        :
        # HEPSCORE_CONFIG_FILE_CREATION
        cat > $WORKDIR/$HEPSCORE_CONFIG_FILE <<EOF
hepscore_benchmark:
        :
        :
EOF
        # SUITE_CONFIG_FILE_CREATION
        cat > $WORKDIR/$SUITE_CONFIG_FILE <<EOF2
activemq:
        :
        :
EOF2
        if [ -f $WORKDIR/$HEPSCORE_CONFIG_FILE ]; then
```

```
        cat $WORKDIR/$HEPSCORE_CONFIG_FILE
    fi
    }
    :
```

One can just run this bash script to run the MadGraph workload with the hep-benchmark-suite.

3.3 Using the Energy measurement plugins

Along with measuring performance, various other metrics are of importance to us. Studying how the power consumption varies with the load, memory-usage, etc. could give interesting insights and help us use the best possible resources in terms of time and energy efficiency. To be able to do that, we use a plugin supported by the suite [4]. The plugins are designed to run concurrently, and thus they do not interfere with the benchmarking process. They are executed in three phases:

- pre : before the benchmarking starts
- during: during benchmarking
- post: after the benchmarking

To use a plugin, one needs to configure the plugin in the suite's configuration file. It is discussed in the following sub section.

3.3.1 Configuration

The configuration should include a `plugins` sections under which the user must list the used plugins. These plugin names must be identical to the class names in `hepbenchmarksuite/plugins/registry`. To use the energy measurement plugins, one can configure it in the configuration file in the `hepscore_install()` function as follows:

Listing 3.7: Configuring plugins in the suite

```
    :
hepscore:
    :
  clean: True
```

```
plugins:
  CommandExecutor:
    metrics:

      load:
        command: uptime
        regex: 'load average: (?P<value>\d+.\d+),'
        unit: ''
        interval_mins: 0.1

      server-power-consumption:
        command: sudo ipmitool dcmi power reading
        regex: 'Instantaneous power reading:
        \s*(?P<value>\d+) Watts'
        unit: W
        interval_mins: 0.1

      gpu-power-consumption:
        command: nvidia-smi --query-gpu=power.draw --
        format=csv,noheader,nounits
        regex: '(?P<value>\d+(.\d+)?).*'
        unit: W
        interval_mins: 0.1
      gpu-usage:
        command: nvidia-smi --query-gpu=utilization.gpu
        --format=csv,noheader,nounits
        regex: '(?P<value>\d+(.\d+)?).*'
        unit: W
        interval_mins: 0.1
```

In our system setup, users can choose what data they want to track. They do this by specifying certain metrics within the `plugins` section, under the `CommandExecutor` keyword. For example, we're interested in keeping an eye on how busy the processor cores are, how much power the server uses, and how the GPU is doing.

To get this data, we need to tell the system what commands to use. For instance, we use a tool called `ipmitool` to find out how much power the server is using. This tool needs special access rights, so it's important to be aware of that. The relevant

information can be accessed through the regular expression `regex`.

Each metric we track has a unit of measurement, like watts for power or percentage for CPU load. We also decide how often to take measurements. In our case, we take readings every 0.5 minutes, or every 30 seconds, to get a good picture of what's happening.

For longer tasks that last hours, it's better to take measurements every few minutes. But for tasks that change quickly, like fast-paced simulations, it's helpful to take measurements every few seconds. This way, we can stay on top of things and make informed decisions about our system's performance.

Results

The results are stored in the suite report (in a `json` file). The data for pre, during and the post benchmarking phases are stored under different keys. The following example shows a snippet of the report for the pre benchmarking phase:

Listing 3.8: Pre phase plugin results

```
    :
    :
    :
  "plugins": {
    "CommandExecutor": {
      "pre": {
        "power-consumption": {
          "start_time": "2023-08-11T15:02:08.767934Z",
          "end_time": "2023-08-11T15:02:08.767934Z",
          "values": [
            604.0
          ],
          "statistics": {
            "min": 604.0,
            "mean": 604.0,
            "max": 604.0
          },
          "config": {
            "interval_mins": 0.16666666666666666,
            "command": "sudo ipmitool dcmi power reading",
            "regex": "Instantaneous power reading:\\
```

```
            s*(?P<value>\\d+) Watts",
            "unit": "W",
            "aggregation": "sum"
        }
    }
:
:
:
```

Here, the values for the pre phase are stored under the `pre` keyword. Each metric's data is kept under its respective keyword. For example, we were interested in the `power-consumption` metric here.

The start and end times are also recorded, along with the observed values. In this case, since this is pre-phase of plugin, we only observed one value before the benchmarking began. Under the `statistics` keyword, we can see the minimum, maximum, and average values, giving us an idea of the overall range. These values are important when the plugin stores many values during benchmarking.

Now that, we have discussed the necessary theoretical background and how to run and configure the suite, let's analyze the results in the next section.

CHAPTER 4

Benchmarks

4.1 Results

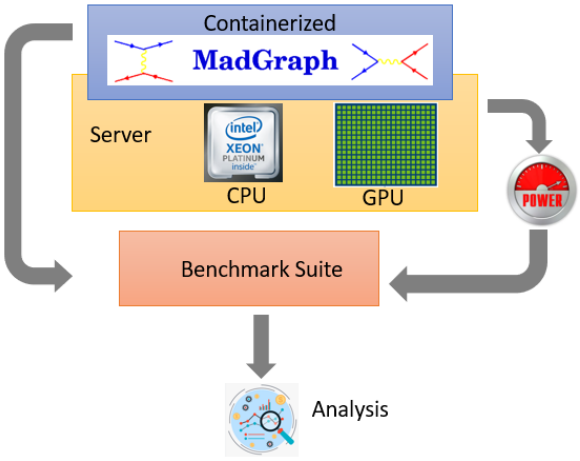

Figure 4.1: Measurement Methodology

The server where we ran the MadGraph application is powered by an Intel Platinum 8362 CPU running at 2.80GHz, comprising of 64 cores, and an Nvidia L4 GPU, as shown in Figure 4.1. This server was chosen because similar machines are employed for running HEP simulations being done at CERN.

4.1.1 Power Efficiency

The power consumption of the server during the execution of 100 events of the MadGraph application is illustrated in Figure 4.2. Our goal here isn't to compare speed but rather to focus on the power usage of both CPU and GPU when each of those are working at full capacity, in other words, when the resources of the machine are fully saturated. We can see that the power consumption is significantly higher for the CPU only run.

In this scenario, the graph displays two distinct curves: the blue curve corresponds to computations executed on the CPU, while the orange curve (both dotted and solid) corresponds to the GPU run. Notably, for the GPU run, the CPU serves solely to offload tasks to the GPU, acting as an orchestrator for offloading computational work. This is not conventionally true in general, since this is just a stand-alone container. Currently, all calculations are being done by GPU (random number generation, mapping of random numbers to particle momenta, calculation of matrix elements from particle momenta). In a realistic scenario, there would be some calculations on CPU as well. This will soon be interfaced with HEPscore.

During CPU-only operations, the container manages executions with varying CPU vectorization techniques, including no vectorization, SSE4, AVX2, and others. Here, the first narrow peak can be ignored since it corresponds to HEPscore initialization. Refer to the figure 4.5 for the following bullet points since it provides labelling for the sake of clarity.

- The next two plateau-like features on the graph (blue curve) represent computations performed in both double and single precision without any vectorization.
- Subsequent plateaus occur for SSE4 vectorization in both double and single precision.
- The next two plateaus after the SSE4 vectorizations represent the AVX2 vectorization in both double and single precision.
- The next two plateaus after the AVX2 vectorizations represent the 512y vectorization in both double and single precision.
- The next two plateaus after the 512y vectorizations represent the 512z vectorization in both double and single precision.

Across all instances, it's evident that double precision computations consistently require approximately twice the time of single precision computations, aligning with the expected behavior. Also, refer to the section 4.1.3 for more details.

Here, both the solid blue and orange curve show power consumption of the entire server. While the dotted orange curve shows the power consumption solely of the GPU while running the benchmarks on GPU.

In summary, the graph provides insights on the comparison of power consumption for CPU and GPU. The graph also provides insights into the time taken for computations under various vectorization techniques, highlighting the inherent differences in performance between double and single precision calculations.

Based on the plots, the power consumption across different vectorization modes: none, SSE4, AVX2, 512Y and 512Z is the same. This consistency implies a substantial reduction in total energy consumption when employing vectorization techniques. Therefore, the amount of energy being consumed doesn't increase with the employment of vectorization techniques, even when the throughput rises significantly.

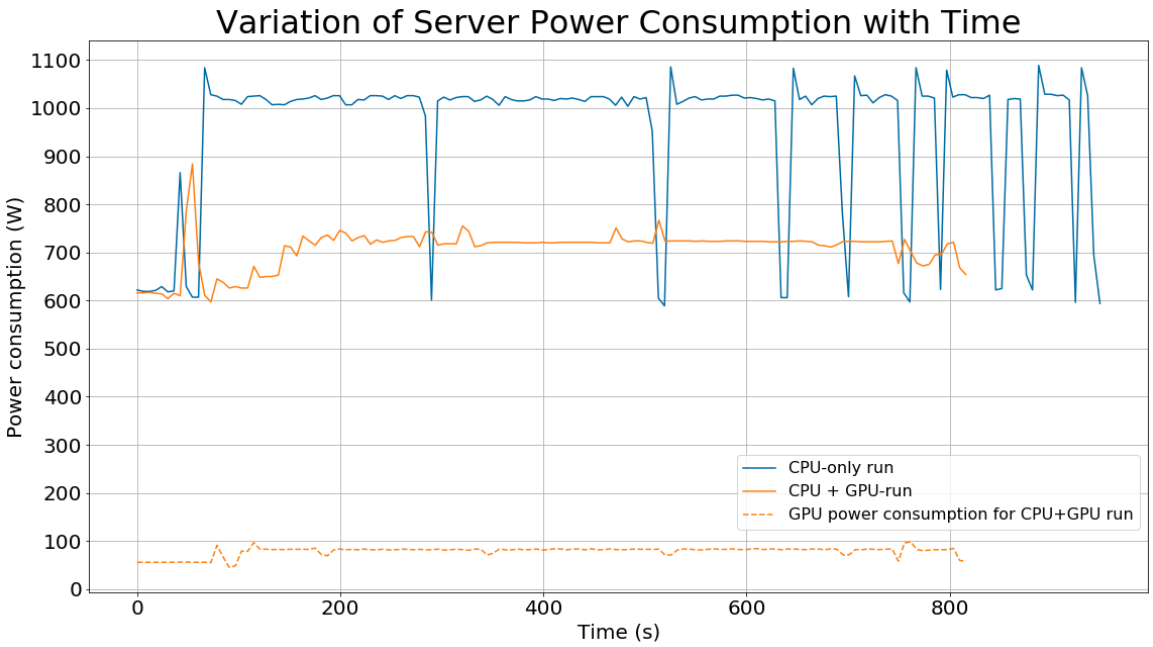

Figure 4.2: Variation of Server Power Consumption with Time

4.1.2 Load and Utilization

The Intel Platinum core CPU used for our analyses typically operates on ≈ 500 Watts [12] for a dual socket system. One can observe that the increase in CPU load leads to a increase in power consumption as shown in the CPU-only curve and as can be seen in figure 4.3. As the load increases, a large number of transistors on CPU become active which could have led to the increase in the power consumption. Moreover, increasing the load also leads to the increase in the number of physical cores being used at any given time, therefore, the power consumption increases.One can also see that the CPU load is minimal for the CPU + GPU run since CPU is just being used to offload the work to the GPU. The Nvidia L4 GPU typically operates on ≈ 72 Watts [13] and it increases to ≈ 80 when the GPU is being used.

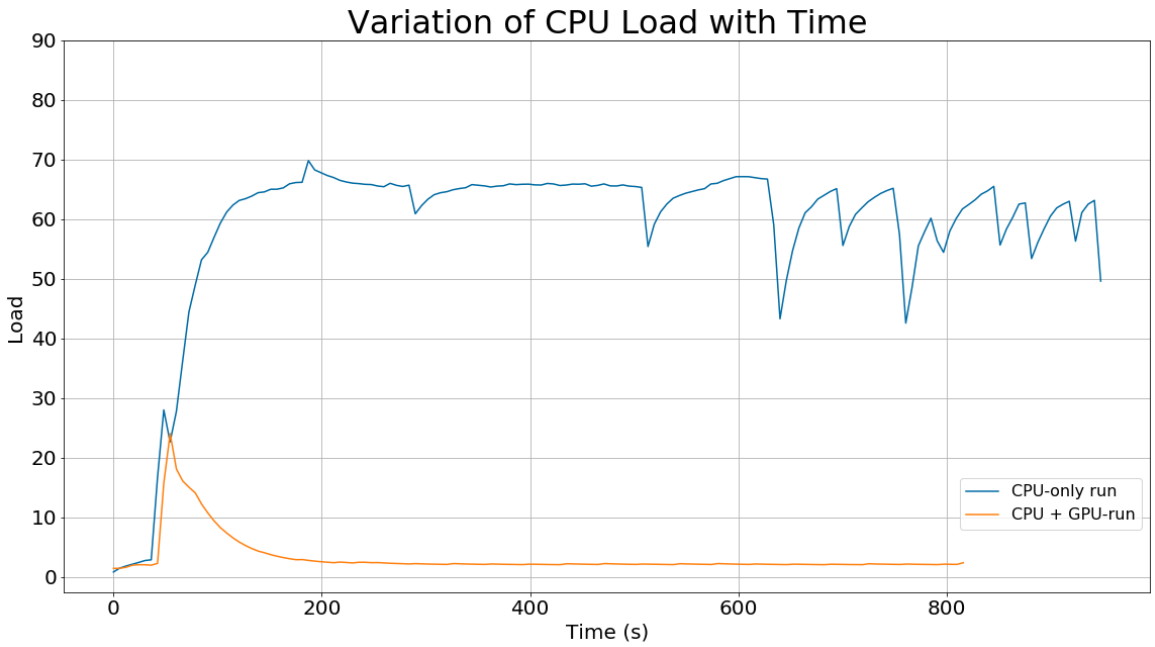

Figure 4.3: CPU Utilization

Figure 4.4 shows that the GPU utilization was 100% for the CPU + GPU run.

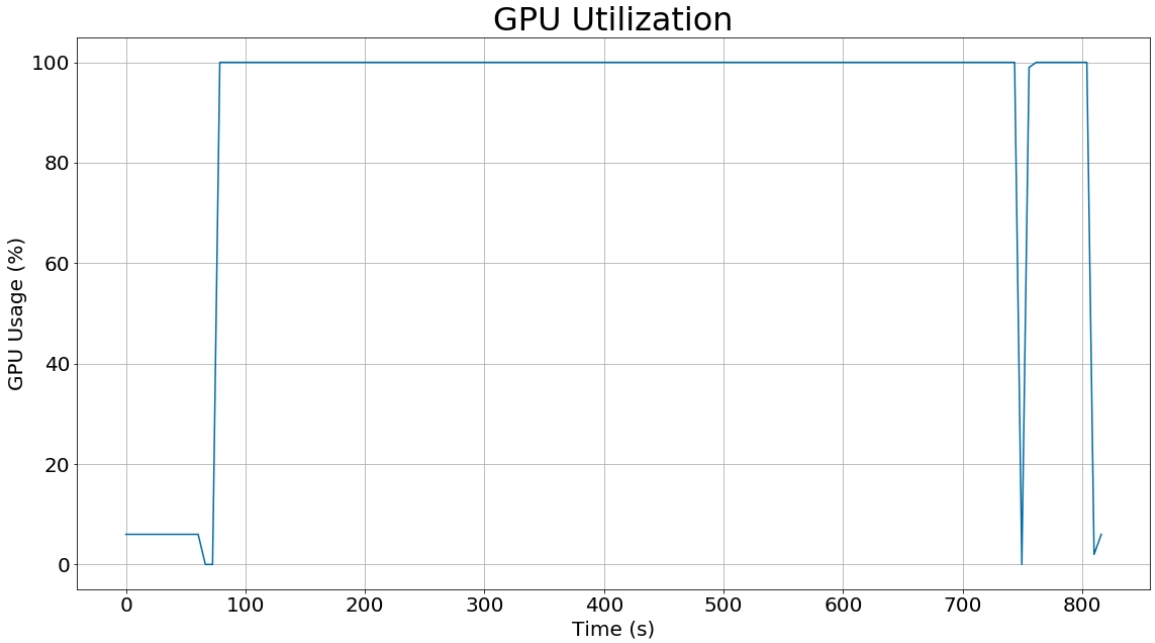

Figure 4.4: GPU Utilization

4.1.3 HEPScore

Figure 4.5 shows the hepscores for CPU (different vectorizations) and GPU. The little graph on the top right is only to make visualization easier. As mentioned in 2.1.3, the prefix `d` refers to the double precision computations and the prefix `f` refers to the single precision computations. The suffixes `none`, `sse4`, `avx2`, `512y` and `512z` are the vectorizations that were used. It can be inferred that the single precision computations are approximately twice as time efficient as the double precision calculations while using vectorization on CPU. When compared with the plot on the right, one can see how the length of the hep-score bar is inversely proportional to the length of each peak. This tells us that the higher the hepscore, more time efficient is the corresponding configuration. As expected, the '512z' vectorization is the most time efficient. Let us look at each vectorization one by one:

- **none**: `d` and `f` have the same throughput, as numerical precision with no vectorization is almost irrelevant
- **SSE4**: HEPscore gains a factor of 2 in `d` and a factor of 4 in `f` since, in a 128 bit (SSE4) SIMD (Single Instruction, Multiple Data) register one can fit two 64-bit doubles and four 32-bit floats
- **AVX2**: HEPscore gains a factor of 4 in `d` and a factor of 8 in `f` since, in a 256 bit (AVX2) SIMD register one can fit four 64-bit doubles and eight 32-bit floats
- **512y**: It is similar to avx2 because it uses the same 256 bit (ymm) registers
- **512z**: HEPscore gains a factor of 8 in `d` and a factor 16 in `f` since, in a 512 bit (AVX512) SIMD register (zmm) one can fit eight 64-bit doubles and sixteen 32-bit floats. Here, we lose a bit from clock slowdown when the machine is full. **This clock slowdown is a common phenomena in x86 when full vector units are used.**

For more details about MadGraph and the vectorizations, refer to the paper [22]. Since we were using an Nvidia L4 GPU here, which mainly provides support for floating point computations only, the double precision score is really low for the GPU.

Also, it can be noted that given a precision, say, double precision, the matrix elements remain exactly the same within the precision. For example, the matrix elements will remain identical for no vectorization, SSE4, AVX2, 512y, 512z or CUDA.

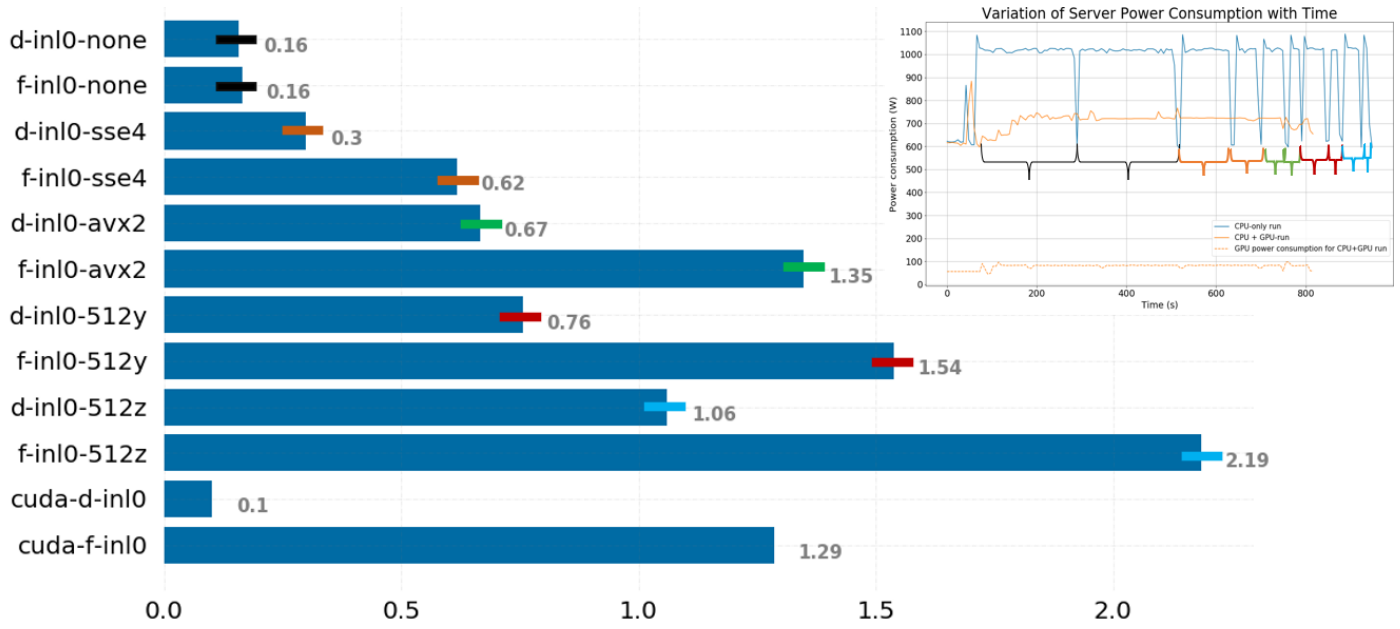

Figure 4.5: HEPScore for different Vectorizations

4.1.4 Miscellaneous

Apart from these results, work was conducted on the analysis of effect of the Intel microcode update [17] on different vectorizations. This microcode update was done as a mitigation strategy for the Gather Data Sampling (GDS) [21] (also called the 'Downfall') vulnerability [5]. The vulnerability stems from memory optimization functionalities present in Intel processors, inadvertently disclosing internal hardware registers to software. Consequently, this enables untrusted software to retrieve data stored by unrelated programs, a scenario that typically wouldn't occur. Intel released a microcode update to mitigate this vulnerability. This update could potentially affect the vectorization units. The microcode update was done on all the CERN servers as a security measure. The results we present here, in figure 4.6 are preliminary results. These results compare HEPscore for the execution of the MadGraph container, before and after the microcode update. It can be seen that the ratio stays 1 for all the vectorizations except the 512y vectorization. These are just preliminary results and not yet understood entirely. It could also be related to the way the data was collected. This test must be repeated for further analysis.

The analysis that was done for Madgraph was also done for the `cms-hlt` workload [1]. These results are preliminary and need further assessment and analysis. Some of the results are presented in appendix A.3.

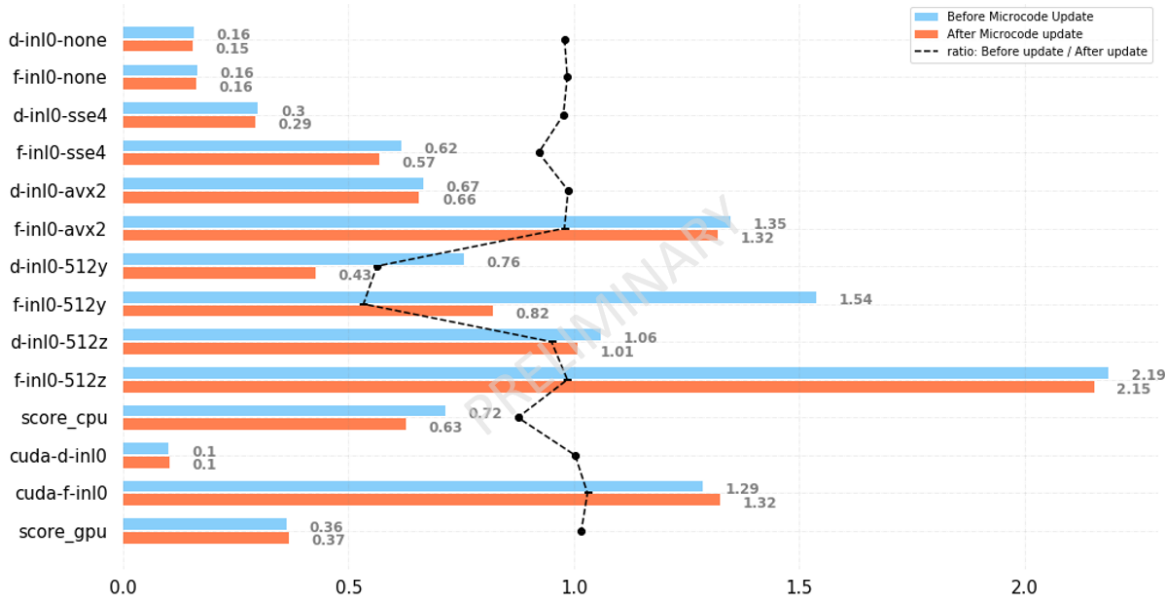

Figure 4.6: Comparison before and after the microcode update

CHAPTER 5

Conclusion and Outlook

5.1 Conclusion

In this thesis, a comprehensive analysis of the power consumption and performance characteristics of the server was done. The server comprises of an Intel Platinum 8362 CPU running at 2.80GHz with 64 cores and an Nvidia L4 GPU.

The primary objective was to investigate power consumption under various scenarios. A particular focus was put upon comparing CPU-only and CPU+GPU execution when the computational resources were fully utilized. A total of 100 events of the MadGraph application were executed to gather data for the analysis.

The results clearly demonstrate a significant difference in power consumption between CPU-only and CPU+GPU execution. When running solely on the CPU, power consumption was notably higher. In contrast, when offloading computation to the GPU, while keeping the CPU load minimal, power consumption decreased significantly. This outcome highlights the energy efficiency gained by leveraging GPU acceleration for computationally intensive tasks.

Furthermore, our analysis delved into different CPU vectorization strategies, such as no vectorization, SSE4, and AVX2, among others. As expected, single-precision computations consistently outperformed double-precision calculations across all vectorization strategies, taking approximately half the time.

We also examined the power consumption characteristics of the Intel Platinum core CPU, which typically operates at around 500 Watts in a dual-socket system. It was observed that increasing CPU load led to a corresponding increase in power consumption, as evident in the CPU-only execution curve.

On the GPU side, the Nvidia L4 GPU consumed approximately 72 Watts under

normal conditions, which increased to approximately 80 Watts when the GPU was actively engaged.

This tells us that the power consumption indeed increases while performing computationally intensive tasks (more load). But an important result is that the power consumption remains in the similar range while using different vectorizations. One of the big takeaways from this work is: while using 512z (double and single precision), one gains a factor of 16 and 8 respectively in the performance. But the power consumption remains very similar to what was observed for no vectorization. Thus one can do more work in less amount of time (and therefore consume very less power when implementing vectorization).

In summary, our analysis concludes that for the given dataset and workload, CPU-only execution results in higher power consumption as compared to the CPU+GPU execution. And that the MadGraph container is highly parallel (also because it is a toy model). Since the container is highly parallel, we gain performance with different vectorizations and precisions. The power consumption for different vectorizations remains the same, hence, we can save a significant amount of energy using 512z or 512y vectorization techniques.

5.2 Outlook

We aim to integrate the current studies with the upcoming MadGraph container. We also aim to calibrate with the new data for more accurate results. In this report, `ref-scores` were taken to be 1.0 for simplicity but it should ideally be calculated on the reference machine and then the HEPscores for our server should be calibrated accordingly. The analysis can also be done on new generation Nvidia GPUs and ARM architectures.

In figure 4.2, the solid orange curve seems much longer than it should be. This is because the GPU workload is not guaranteed to run exactly the same number of events as the CPU workload. So, comparing the horizontal scales of the blue and orange curves cannot be done at face value and the analysis requires a bit more work.

The work presented in this thesis was done for MadGraph but the same can also be done for other workloads, especially the workloads which already have a GPU interface. Further studies are needed on different servers to verify if the current findings can be generalised. There are multiple components that could be server-dependent: server motherboard, power module, model of CPU and/or GPU etc.

Lastly, the analysis done for the microcode update needs further work and more tests. This could confirm whether the update actually affects the performance for different vectorizations.

APPENDIX A

Appendix A

A.1 HEPscore CLI parameters

The `hep-score` command provides a versatile interface for benchmarking tasks, offering various options to customize execution and output. When invoked, it requires a positional argument specifying the base output directory where benchmark results will be stored. Additionally, it supports several optional arguments:

- `-m [singularity,docker]` or `--container_exec [singularity,docker]`: Allows the user to specify the container platform for benchmark execution, with Singularity being the default option.
- `-S` or `--userns`: Enables user namespace for Singularity, if supported.
- `-c` or `--clean`: Cleans residual container images from the system after the run.
- `-C` or `--clean_files`: Cleans residual files and directories after execution and tar results.
- `-f [CONFFILE]` or `--conffile [CONFFILE]`: Specifies a custom config YAML to use instead of the default.
- `-l` or `--list`: Lists built-in benchmark configurations and exits.
- `-n [NAMEDCONF]` or `--namedconf [NAMEDCONF]`: Uses the specified named built-in benchmark configuration.

- `-r` or `--replay`: Replays output using an existing results directory specified by `OUTDIR`.
- `-o [OUTFILE]` or `--outfile [OUTFILE]`: Specifies the summary output file path/name.
- `-y` or `--yaml`: Creates YAML summary output instead of JSON.
- `-p` or `--print`: Prints configuration and exits.
- `-V` or `--version`: Shows the program's version number and exits.
- `-v` or `--verbose`: Enables verbose mode, displaying debug messages.

Furthermore, the `hep-score` command provides usage examples for different scenarios:

1. Running benchmarks via Docker while displaying verbose information.
2. Running benchmarks using Singularity (default) with a custom benchmark configuration.
3. Listing built-in benchmark configurations.
4. Running with a specified built-in benchmark configuration.

Overall, the `hep-score` command offers a comprehensive set of options to cater to various benchmarking needs, making it a powerful tool in performance evaluation tasks.

A.2 The HEPscore configuration for running MadGraph on GPU

We used the following HEPscore configuration to run MadGraph on GPU:

Listing A.1: The HEPscore configuration for running MadGraph on GPU

```
hepscore_benchmark:
  benchmarks:
    mg5amc-MadGraph4gpu-2022-bmk:
      results_file: mg5amc-MadGraph4gpu-
```

```
      2022_summary.json
      ref_scores:
        ggttgg-sa-cuda-d-inl0: 1
        ggttgg-sa-cuda-f-inl0: 1
      weight: 1.0
      version: v0.7
      args:
        events: 100
        extra-args: "--gpu"
      gpu: TRUE
  settings:
    name: MadGraph_Keshvi
    reference_machine: "CPU Intel(R) Xeon(R)
    CPU E5-2630 v3 @ 2.40GHz"
    registry: $REGISTRY://gitlab-registry.cern.ch/
    hep-benchmarks/hep-workloads$REGISTRY_SUFFIX
    method: geometric_mean
    repetitions: 3
    retries: 1
    scaling: 1
    container_exec: $EXECUTOR
```

A.2.1 More about MadGraph

AS per the reference [16], the following scores for the MadGraph workload are reported (where <process> is eemumu, ggtt, ggttg or ggttgg):

- <process>-sa-cuda-d-inl0 : CUDA (GPU), double precision, without inlining
- <process>-sa-cuda-d-inl1 : CUDA (GPU), double precision, with inlining
- <process>-sa-cuda-f-inl0 : CUDA (GPU), single precision, without inlining
- <process>-sa-cuda-f-inl1 : CUDA (GPU), single precision, with inlining
- <process>-sa-cpp-d-inl0-none : C++ (CPU), double precision, without inlining, scalar (no SIMD)

- <process>-sa-cpp-d-inl0-sse4 : C++ (CPU), double precision, without inlining, SSE4
- <process>-sa-cpp-d-inl0-avx2 : C++ (CPU), double precision, without inlining, AVX2
- <process>-sa-cpp-d-inl0-512y : C++ (CPU), double precision, without inlining, AVX512 with 256-bit ymm registers
- <process>-sa-cpp-d-inl0-512z : C++ (CPU), double precision, without inlining, AVX512 with 512-bit ymm registers
- <<process>-sa-cpp-d-inl1-none : C++ (CPU), double precision, with inlining, scalar (no SIMD)
- <process>-sa-cpp-d-inl1-sse4 : C++ (CPU), double precision, with inlining, SSE4
- <process>-sa-cpp-d-inl1-avx2 : C++ (CPU), double precision, with inlining, AVX2
- <process>-sa-cpp-d-inl1-512y : C++ (CPU), double precision, with inlining, AVX512 with 256-bit ymm registers
- <process>-sa-cpp-d-inl1-512z : C++ (CPU), double precision, with inlining, AVX512 with 512-bit ymm registers
- <process>-sa-cpp-f-inl0-none : C++ (CPU), single precision, without inlining, scalar (no SIMD)
- <process>-sa-cpp-f-inl0-sse4 : C++ (CPU), single precision, without inlining, SSE4
- <process>-sa-cpp-f-inl0-avx2 : C++ (CPU), single precision, without inlining, AVX2
- <process>-sa-cpp-f-inl0-512y : C++ (CPU), single precision, without inlining, AVX512 with 256-bit ymm registers
- <process>-sa-cpp-f-inl0-512z : C++ (CPU), single precision, without inlining, AVX512 with 512-bit ymm registers
- <process>-sa-cpp-f-inl1-none : C++ (CPU), single precision, with inlining, scalar (no SIMD)

- `<process>-sa-cpp-f-inl1-sse4` : C++ (CPU), single precision, with inlining, SSE4
- `<process>-sa-cpp-f-inl1-avx2` : C++ (CPU), single precision, with inlining, AVX2
- `<process>-sa-cpp-f-inl1-512y` : C++ (CPU), single precision, with inlining, AVX512 with 256-bit ymm registers
- `<process>-sa-cpp-f-inl1-512z` : C++ (CPU), single precision, with inlining, AVX512 with 512-bit ymm registers

A.3 Miscellaneous results

In this section we present the miscellaneous results which were done as part of the process but are not included in the main text as these are preliminary and haven't been analyzed yet.

A.3.1 Benchmarking 'cms-hlt' workload

The analysis that we did for MadGraph was also done for for the 'cms-hlt' benchmark. The results are presented in figures A.1, A.2 and A.3. These results still need to be understood and analyzed. For the sake of completeness, we present these result here.

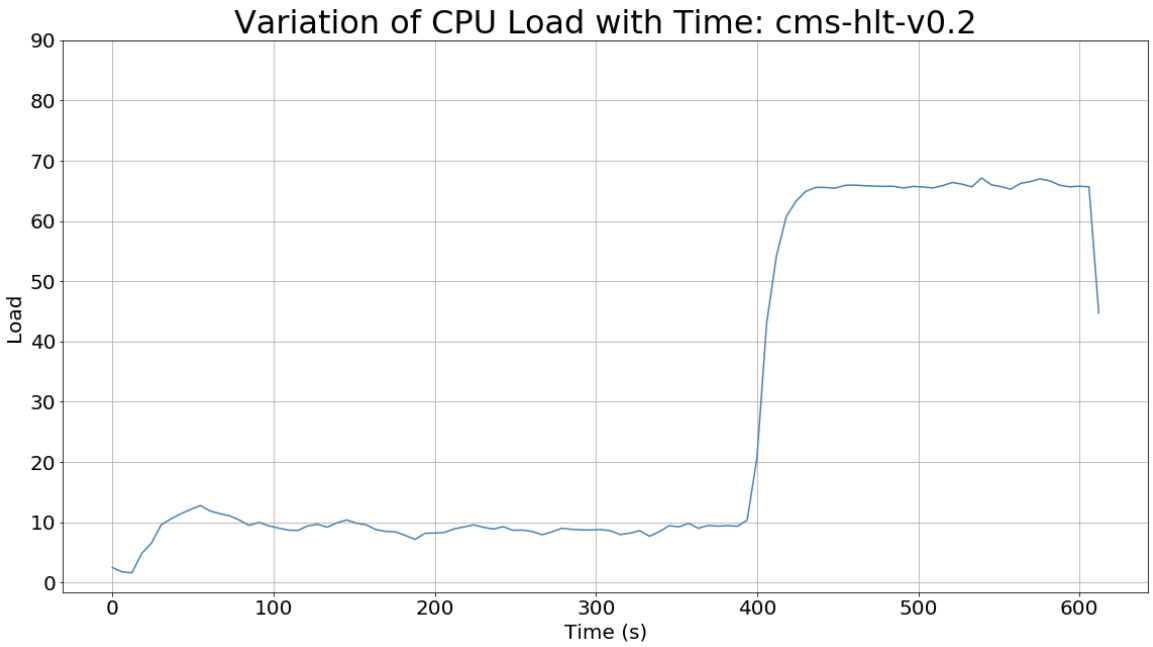

Figure A.1: CPU Utilization for CMS-HLT benchmark

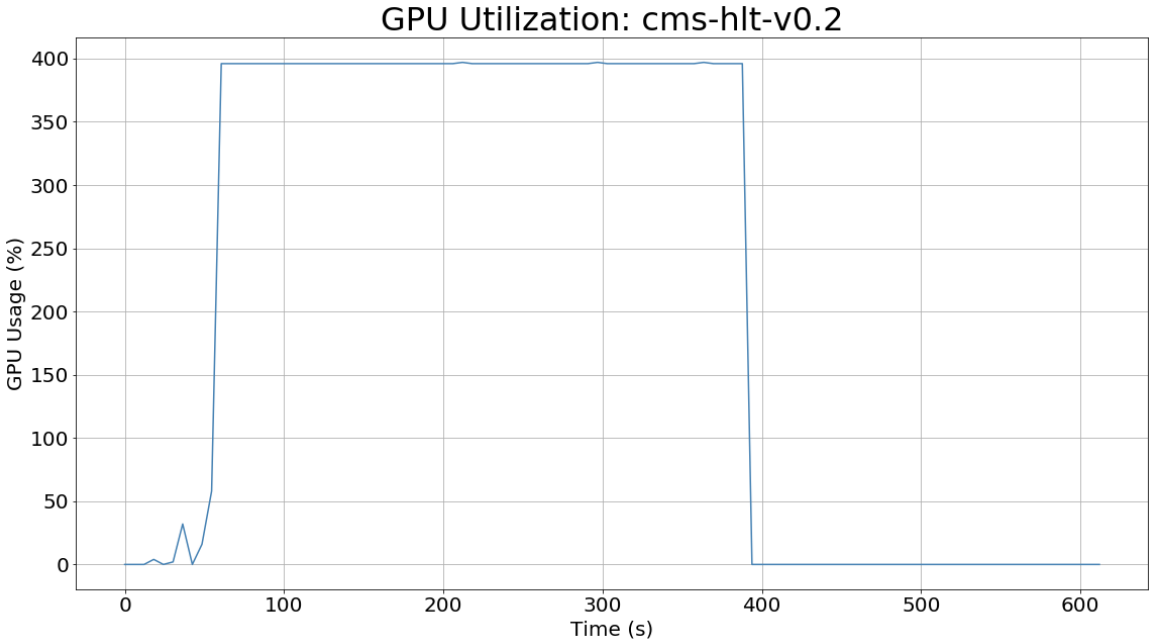

Figure A.2: GPU Utilization for CMS-HLT benchmark

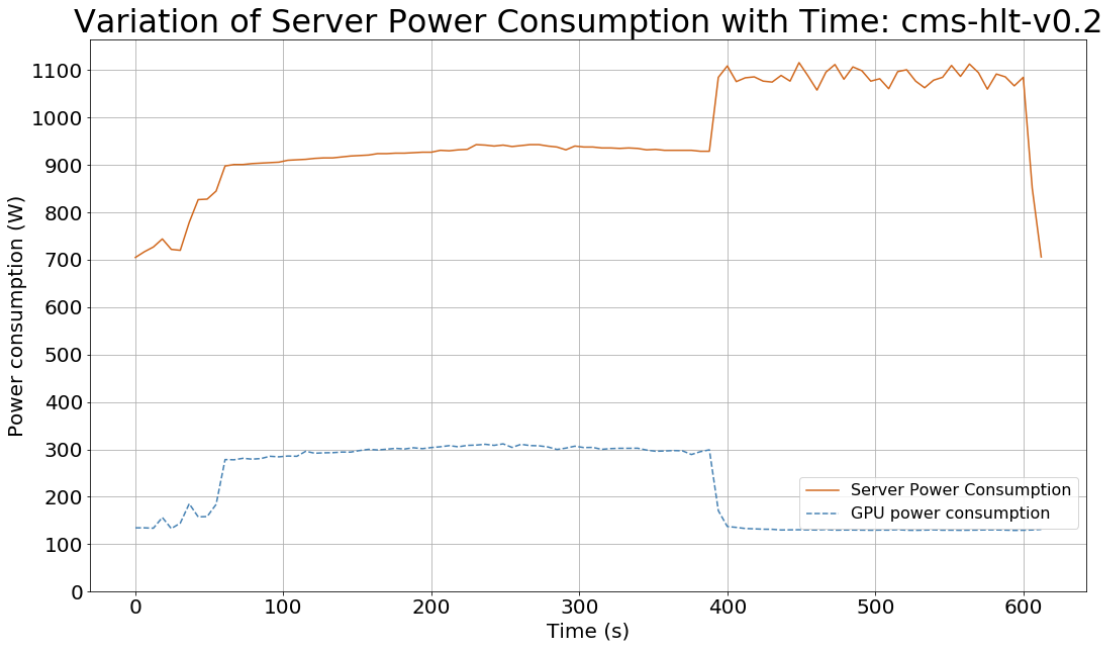

Figure A.3: Variation of server power consumption for CMS-HLT benchmark

Bibliography

[1] Cms workload. URL: https://gitlab.cern.ch/hep-benchmarks/hep-workloads/-/tree/master/cms?ref_type=heads.

[2] Docker. URL: https://docs.docker.com/.

[3] Dockerhub. URL: https://hub.docker.com/.

[4] Energy plugin. URL: https://gitlab.cern.ch/hep-benchmarks/hep-benchmark-suite/-/tree/qa/hepbenchmarksuite/plugins.

[5] Gather data sampling. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/gather-data-sampling.html.

[6] hep-benchmark-suite. URL: https://gitlab.cern.ch/hep-benchmarks/hep-benchmark-suite/.

[7] hep-score. URL: https://gitlab.cern.ch/hep-benchmarks/hep-score.

[8] hep-workloads. URL: https://gitlab.cern.ch/hep-benchmarks/hep-workloads.

[9] Intel AVX-512. URL: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html.

[10] Intel AVX2. URL: https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/009/intel-advanced-vector-extensions-2-intel-avx2/.

[11] Intel SSE4 programming reference. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-705230.pdf.

[12] Intel xeon platinum 8362 processor. URL: https://ark.intel.com/content/www/us/en/ark/products/217216/intel-xeon-platinum-8362-processor-48m-cache-2-80-ghz.html.

[13] L4 tensor core gpu, nvidia. URL: https://www.nvidia.com/en-us/data-center/l4/.

[14] Lhc image gallery. URL: https://home.cern/resources/image/accelerators/lhc-images-gallery.

[15] Madgraph. URL: http://madgraph.phys.ucl.ac.be/.

[16] Madgraph container. URL: https://gitlab.cern.ch/hep-benchmarks/hep-workloads/-/tree/master/mg5amc/madgraph4gpu-2022.

[17] microcode-20230214 release. URL: https://github.com/intel/Intel-Linux-Processor-Microcode-Data-Files/releases/tag/microcode-20230214.

[18] run_hepscore.sh. URL: https://gitlab.cern.ch/hep-benchmarks/hep-benchmark-suite/-/raw/master/examples/hepscore/run_HEPscore.sh.

[19] Singularity. URL: https://docs.sylabs.io/guides/3.5/admin-guide/admin_quickstart.html.

[20] Domenico Giordano, Jean-Michel Barbet, Tommaso Boccali, Gonzalo Menndez Borge, Christopher Hollowell, Vincenzo Innocente, Walter Lampl, Michele Michelotto, Helge Meinhard, Ladislav Ondris, Andrea Sciab, Matthias J. Schnepf, Randall J. Sobie, David Southwick, Tristan S. Sullivan, Andrea Valassi, Sandro Wenzel, John L. Willis, and Xiaofei Yan. Hepscore: A new cpu benchmark for the wlcg, 2023. arXiv:2306.08118.

[21] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[22] Andrea Valassi, Taylor Childers, Laurence Field, Stephan Hagebck, Walter Hopkins, Olivier Mattelaer, Nathan Nichols, Stefan Roiser, David Smith, Jorgen Teig, Carl Vuosalo, and Zenny Wettersten. Speeding up madgraph5 amc@nlo

through cpu vectorization and gpu offloading: towards a first alpha release, 2023. arXiv:2303.18244.

List of Figures

Listings

Abbreviations

CERN Conseil europen pour la Recherche Nuclaire.

CLI Command Line Interface.

CPU Central Processing Unit.

GPU Graphics Processing Unit.

HEP High Energy Physics.

LHC Large Hadron Collider.

WLCG Worldwide LHC Computing Grid.