DAHL: Energy-Efficient Machine Learning on Heterogeneous CPU using Task-based Programming

1st Anonynmous Author
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Anonynmous Author
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Anonynmous Author
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—The growth of artificial intelligence raises concerns about electricity consumption, especially since the rise of generative models. At the same time, hardware is becoming increasingly heterogeneous, which is often an obstacle to efficient and frugal usage of computer resources. CPUs are a great example, as heterogeneous topologies are getting more common. Their cores with different frequencies, cache size or hierarchy, and thread number make it challenging for developers to efficiently parallelize tasks. In this context, we investigate how task-based programming can be used to adapt to such constraints. The ability to schedule tasks based on runtime informations, such as hardware topology, and worker load can solve these problems. To this end, we built `DAHL`, a task-based programming machine learning framework using the `StarPU` runtime system. As a use case, we study an image classification task, using datasets with increasingly larger images. We perform our experiments over machines with heterogeneous and homogeneous CPUs. A reference is established by comparing different `Pytorch` configurations against `DAHL`. Then we evaluate the runtime and energy gains of several `StarPU` schedulers. Results show that, specifically on heterogeneous CPUs, the default `Pytorch` configuration can be $7.5\times$ slower and consume $17\times$ more at worst. Additionally, we demonstrate that the scheduler choice on `StarPU` can impact the runtime with up to 22% speedup, and energy consumption with up to 15% energy gains. Last, we draw guidelines to choose scheduler families depending on context.**

***Index Terms*—Energy-Efficiency, Machine Learning, Heterogeneity, Scheduling, Task-Graph-Computing-Systems.**

I. Introduction

At the end of 2025, seven out of the nine planet boundaries were crossed. In this context, it is clear that attention should be paid to the sustainability of Artificial Intelligence (AI). Numerous studies have been conducted to analyze the impact of electricity [1], carbon emissions [2], water consumption [3], and more recently, on applying Life Cycle Assessment (LCA) methods [4], [5]. As a first step, reducing the electricity consumption of AI systems is a good way to mitigate indirect impacts. To this extent, improving the energy efficiency of AI frameworks seems to be a key factor that could impact both training and inference phases. However, the heterogeneity – which is present at many different levels in platforms – makes the correct use of resources harder. To this end, a lot of work have been accomplished towards training on heterogeneous clusters [6]–[9], but they mostly focus on tackling synchronization issues and optimizing training time, while ignoring the energy consumption. Furthermore, most of the Machine Learning (ML) research is oriented towards the use of GPUs. Indeed, GPUs can be even more efficient than CPUs, but it requires large investments in this specific technology. We also know that the use phase of digital equipments represents only a small part of their ecological impact. For example, in France, a governmental analysis [10] evaluated that the use phase generally represented 40% of carbon emissions, the remaining being for the manufacturing, distribution and end-of-life. This of course excludes impacts related to resource depletion and water pollution.

In this context, we think that studying CPUs might be relevant, whether for model training or inference. Through the years, computers evolved towards more and more heterogeneous architectures, for example via Non-uniform memory access (NUMA) or heterogeneous CPU topologies themselves. Intel introduced Performance-cores (P-cores), to provide speed, with multiple threads per core, Efficient-cores (E-cores), to provide a balance between performance and energy consumption, and more recently Low Power Efficient-cores (LP E-cores) designed for mobile processors. Such designs create new challenges for developers to utilize computer's resources efficiently. To this matter, Task Graph Computing System (TGCS) seem promising, as they offer scheduling capabilities able to adapt to heterogeneous architectures. To the best of our knowledge, only a few preliminary studies have been conducted on the application of TGCS to ML: [11], [12] covers inference while `NNTile` [13] covers training. They are all based on `StarPU`, a runtime system associated to the High Performance Computing (HPC) domain, designed for computations on heterogeneous hardware.

`NNTile` in particular is able to keep steady throughput when training models with up to 50 billion of parameters, while `Pytorch` falls short at 23 billion. This gives encouraging signs for the usage of `StarPU`. However, as `NNTile` specializes on large models, it suffers from task granularity issues when operating on smaller models. These issues are based of a complex balance that developers should make when implementing tasks. On one hand, small tasks – which acts on buffers to the order of vectors – will scale very well when partitioning large tensors, thus creating many parallelization opportunities. On the other hand, defining larger tasks –

which acts on multi-dimensional arrays – will adapt more to different tensor sizes, at the cost of losing some parallelization opportunities. Nevertheless, the first option results in less adaptability to workload, as using smaller inputs results in scheduling overheads.

To fill the gap, we introduce DAHL[1], a general-purpose TGCS-based ML framework using the StarPU runtime system, that we evaluate on growing workloads. In this paper, we define workload as the amount of computations – which depends on the training set size – required to train a model to a desired state. In other words, we perform training on increasingly larger images. Furthermore, we also evaluate the energy efficiency of our framework in relation to Pytorch, on machines with different CPU topologies. Then we compare StarPU schedulers to evaluate their runtime and energy gains on each architectures. Our contributions are the following:

- tool: DAHL, a TGCS-based ML framework, light with 6k source lines of C code, minimal dependencies, full control over the ML pipeline, and at disposition of the community.
- setup: comparison of the energy efficiency of homogeneous and heterogeneous CPU topologies, applied to Convolutional Neural Network (CNN) training, using both software-based energy measurements and wattmeters.
- guidelines: 1) Specifically on heterogeneous CPUs, using the default Pytorch configuration can be 7.5× slower and consume 17× more energy. Oversubscribing threads mitigate this problem. 2) Scheduler choice can speedup runtime up to 22%, and make energy gains up to 15%. 3) Work-stealing-based schedulers have the best energy gains on homogeneous CPUs. Heterogeneous CPUs results varies, but it seems that eager-based schedulers are more effective.

This paper organizes as follows: Section II introduces concepts and related works, Sec. III presents DAHL and StarPU features. Sec. IV describes the experimental methodology, we then show the results in Sec. V, which are discussed in Sec. VI before concluding in Sec. VII.

II. Background and related works

A. Parallel Computing Frameworks

Several existing parallel computing frameworks supports heterogeneity on different levels: SYCL [14] is a C++ high level programming model leveraging different hardware accelerators. It has many implementations that support various API backends including CUDA, OpenCL, OpenMP and is notably apart of Intel OneAPI. Threading Building Blocks [15] also belongs to OneAPI, but focuses on parallelizing tasks on multi-cores processors. OpenMP [16] is a multi-platform C/C++ API that helps parallelizing loops by adding annotations in the code. While it was originally limited by its fork-join model, which prevented opportunities to progress multiple independent-tasks, OpenMP 4.0 [17] introduced ways to represent task dependencies, yet it requires manually specifying these dependencies. This can become tough, especially with complex and interrelated graphs of operations, such as in ML tasks. Runtime systems are better alternatives when dealing with such complex task graphs, as they can infer task dependencies with the type of data accesses (read only, read-write, write only). Those systems can be referred as TGCS, and StarPU [18] is one of the leading framework in the HPC world. Taskflow [19] is an alternative to StarPU, nevertheless the two frameworks can be distinguished by several design choices: compared to StarPU, Taskflow leaves memory handling to the user, tasks should be explicitly allocated to CPU or GPU, but it does support in-graph control flow. Specx [20] uses speculative execution, which consists of evaluating operations in advance in order to improve the degree of parallelism, while taking the risk that the results may subsequently be invalidated. Finally, Legion [21] focuses more on data movement in distributed systems.

StarPU have been selected to build our framework upon for several reasons: Firstly, StarPU is mature and scientifically recognized, notably in HPC domain. Secondly, and most importantly, it focuses on heterogeneity handling at the machine scale. Threading Building Blocks could have been an option to specifically optimize for CPU, but using StarPU leaves the door open to extend our framework to other accelerators.

B. Parallelism in Machine Learning

The parallelization of ML workloads can be performed on several dimensions: *Sample, Operations, Attribute, Parameter* (*SOAP* [22]). *Sample* dimension, consists of parallelizing common tasks over different input samples. It can be either classic batch parallelism or more advanced techniques such as data parallelism [23]. *Operation* dimension, or inter/intra-operation parallelism [7], [24], consists of parallelizing tensor operations, such as matrix-matrix multiplications, among or across operators. *Attribute* dimension is mainly used in hybrid parallelism [6], [25] (which is useful when combined with multiple parallel dimensions), the attribute refers to the division of the input data shape itself, e.g. parallelizing over channels of an image or even splitting the image into smaller images along the width and height dimensions. Finally, *Parameter* dimension or model parallelism [26] allows the model parameters to be divided and learned in parallel.

In our tool, DAHL, we first implemented batch parallelism, as it is the classic way to achieve parallelism on a single machine. However, the use of a TGCS is not limiting at all, and other types of parallelism could be implemented easily.

C. Energy consumption

The assessment of the energy consumption of computer systems [27] is divided in two parts: the idle part which represent the base consumption when unloaded, and a dynamic part which varies, notably depending on the size of the workload. Therefore, preliminary estimations can be

[1]Distributed, Arbitrary, and Heterogeneous Learning:

made by defining a simple energy model using the equation $idle\ energy + dynamic\ energy \times usage$ [28]. For example, `Ecologits` [29] uses different models to estimate energy consumption and environmental impacts. However, reliability of those estimations is limited by the parameters of the models, such as the theoretical energy consumption of a machine. Therefore, it is preferable to use measurement methods based either on external power-meters, or software power-meters to study the actual consumption of systems. Software wattmeters retrieve cumulative energy consumption data via manufacturer interfaces, such as Running Average Power Limit (RAPL) for Intel processors, or NVIDIA Management Library (NVML) for NVIDIA graphics processors. There are many software solutions available for measuring energy, including but not limited to: `CarbonTracker` [30], `Scaphandre` [31], `CodeCarbon` [32], `PowerAPI` [33], and more recently, `Alumet` [34], and `Ecofloc` [35].

Although these software use the same interfaces to collect energy measurements, differences lays in their additional running overhead. Moreover, the total energy consumption is often underestimated compared to physical measurements taken using a wattmeter [36]. This is due to the consumption of fans, storage, and network components, which is not always available on certain hardware. Specifically, it seems that mostly laptops are equipped with monitoring capabilities of these components, in such cases it produces estimations close to wattmeter measurements [37].

In this paper, we perform the experiments both via software and wattmeter to verify our results on two sources.

III. Our framework: Distributed, Arbitrary, and Heterogeneous Learning (DAHL)

`DAHL` is a ML framework developed as a proof-of-concept to study the potential advantages brought by TGCS when running on heterogeneous hardware. It provides from scratch implementations of ML functions and utilities that use the `StarPU` runtime system. The frameworks aims at maximum flexibility and control over the ML pipeline. It offers a balance between code architecture simplicity, and the ability to exploit different hardware. For instance, accessing the implementation of each tasks should take less than four jump to definitions. This opens up easier access to research and optimizations, such as – but not limited to – studying new allocation policies, optimizing data layouts, leveraging inline assembly, or trying novel parallelization schemes. The framework is free, open source, and available on Codeberg [link will be provide at the submission for double blind submission policy]

A. Overview

To illustrate how parallelism is achieved on `StarPU` we will review a simple example of parallelization inside the convolution layer. List. 1 shows the function used to parallelize convolutions during the forward pass of the convolution layer. It takes an *input* image, as well as an *output* buffer to receive the result, in this case feature maps. Their type alias is `dahl_block`, which represent a three dimensional array. Next, *filters_p* has the type `dahl_partition_block`, which holds a partition with multiple blocks. In `StarPU`, one can partition a data buffer into a subset of children buffers. It is then possible to submit parallel tasks on these children. The type of *filters_p* implies that the data is already partitioned, meaning it is ready to receive parallel tasks on its children. The same applies for *output_p*. The loop from line 12 to 23 will then submit tasks using the common *input*, and for each corresponding *filter* and *feature_map*. Bias is then added to each *feature_map*. At runtime, `StarPU` will determine a task-graph, taking into account data dependencies, and execute these tasks in parallel when possible. Fig. 1 shows that, as involved data buffers are only accessed as read-only during the forward convolution, the resulting task graph is fully parallelized. Note that the described function treats a single sample. Therefore, parallelization on the convolution layer is achieved per sample and per feature map.

However, when a data buffer is accessed with write mode, it creates data dependencies with tasks accessing this specific buffer. For instance, in the backward convolution, weights and biases need to be updated. Thus, a synchronization point will be created in the Directed Acyclic Graph (DAG), restricting that the write task finishes first, before proceeding to other tasks depending on it. The same applies with data partitions, e.g., we first split the dataset into batches, but grouping it again requires waiting tasks that were working on each batch.

B. Codelets

As seen in Sec. III-A, data access type is crucial to determine task dependencies. This information is specified in the task metadata itself. In `StarPU`, it is represented as a structure called a *codelet*. It is the description of a task, which abstracts its different implementations. As an example, List. 2 shows the definitions related to the convolution task. The first four lines features the declaration of the CPU and GPU functions, which

```
1  void convolution_forward_sample(
2    dahl_block input,
3    dahl_block output,
4    dahl_partition_block filters_p,
5    dahl_fp const* biases)
6  {
7    // Partition by the filter dimension,
8    // each iteration will tackle a feature map
9    dahl_partition_matrix output_p =
10     dahl_block_partition_along_z(output);
11
12   for (size_t c = 0; c < filters_p.nparts; c++)
13   {
14     dahl_block filter = filters_p.children[c];
15     dahl_matrix feature_map = output_p.children[c];
16
17     // Submit a task on the given input,
18     // for each filter and feature map
19     dahl_task_convolution_2d(input, filter,
       ↪  feature_map);
20
21     // Add bias to each value of the feature map
22     dahl_task_add_value_self(feature_map, biases[c]);
23   }
24 }
```

Listing 1: Example of `DAHL` code, parallelizing convolution tasks during the forward pass, for a given sample.

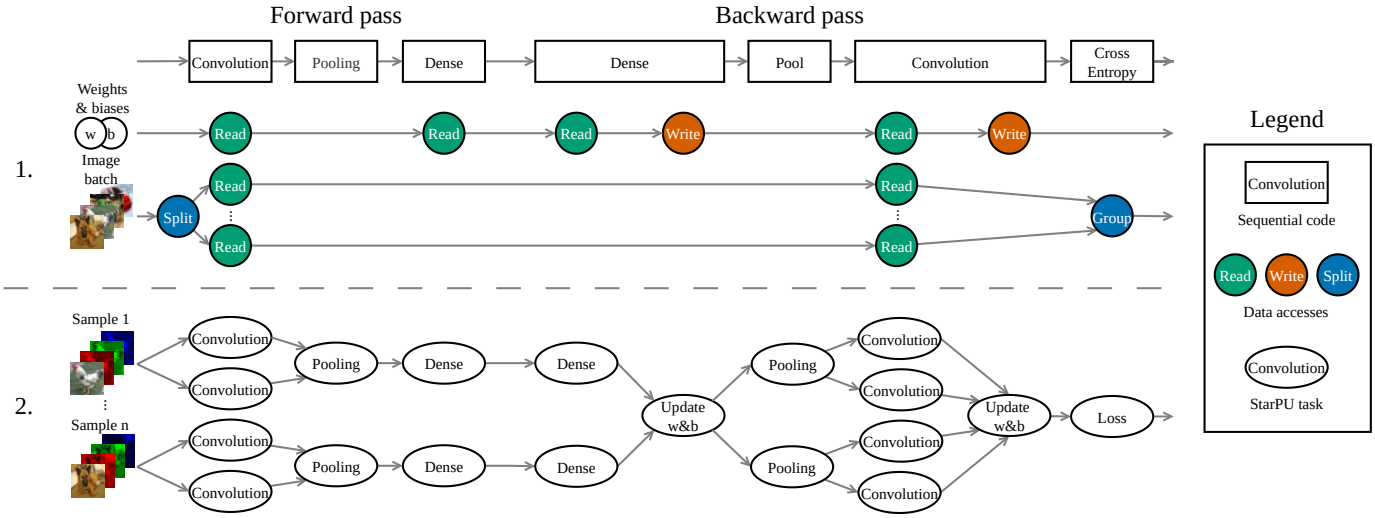

Fig. 1. Simplified representation of batch parallelism on a CNN using the Task-based programming nature of StarPU. The diagram shows 1. the sequential code with execution order of forward and backward pass, and data accesses of the dataset and model parameters; followed by 2. the resulting task graph built by StarPU.

```
1  extern void cpu_convolution_2d(
2      void *buffers[3], void *cl_arg);
3  extern void cuda_convolution_2d(
4      void *buffers[3], void *cl_arg);
5
6  static struct starpu_perfmodel
   ↪  perf_model_convolution_2d = {
7      .type = STARPU_REGRESSION_BASED,
8      .symbol = "convolution_2d"
9  };
10
11 static struct starpu_codelet cl_convolution_2d = {
12     .cpu_funcs = { cpu_convolution_2d },
13     .cuda_funcs = { cuda_convolution_2d },
14     .nbuffers = 3,
15     .modes = { STARPU_R, STARPU_R, STARPU_W },
16     .model = &perf_model_convolution_2d
17 };
```

Listing 2: Example of a `StarPU` codelet definition, here for the convolution 2d function.

contains the actual implementation. They are then referenced on line 12 and 13 in fields *cpu_funcs* and *cuda_funcs*. `StarPU` scheduler will then be able to chose at runtime where to execute the task onto if both CPU and GPU are available. Next lines includes the number of buffers taken as parameters, in addition to their respective data access types. The last field references an optional performance model used to micro-benchmark tasks. This model measures the time required to complete the given task on a specific implementation, and taking into account the input size. The model can be computed using history-based methods, linear regression, and non-linear regression, depending on tasks runtime deviation.

C. Schedulers

Here, we introduce `StarPU`'s schedulers which 1. are not stated as experimental and 2. are not priority-based, as including task priorities for this use case was not relevant. More details and additional schedulers, can be found on `StarPU`'s documentation[2]. We distinguish two types of scheduling policies: non performance modeling and performance model-based policies. The first takes scheduling decisions only based on runtime informations, such as hardware topology, workers load, and task availability. The second exploits informations from the performance models, which are calibrated over multiple runs. Such informations are then used to predict task termination time and improve overall scheduling performance. Below is a brief introduction of the schedulers studied in this paper:

- `random`: each worker have a task queue and schedules tasks randomly.
- `eager`: each worker pulls tasks concurrently from a central queue. However, it does not benefit from data prefetching as scheduling decisions are taken late.
- `peager`: performance model-based version of `eager`.
- `ws` (work stealing): each worker have a task queue and executes its own tasks, however, idle workers can steal from other queues.
- `lws` (locality work stealing): same as `ws` but takes into account spatial and temporal data locality to steal tasks from neighboring workers.
- `dmdar` (deque model data aware ready): schedules tasks where their termination time will be minimal, taking data transfers into account, and according privilege to data already available on target device. Termination time prediction is made possible via performance models.

IV. Methodology

In order to evaluate the impact of the CPU topology on energy consumption, we perform our experiments on 4 machines, 2 of them having heterogeneous CPUs and 2 homogeneous

[2]https://files.inria.fr/starpu/doc/html/Scheduling.html

TABLE I
SPECIFICATIONS OF MACHINES USED IN EXPERIMENTS.

Machine	Model	RAM	CPU	#Cores	#Threads	Frequency	Release
Rammus	Dell Precision 3680	128 GB	Intel i7-14700K	20 (8 P-cores, 12 E-cores)	28	5.6 GHz	2023
Senna	HP EliteBook 640	32 GB	Intel 125U	12 (2 P-cores, 8 E-cores, 2 LP E-cores)	14	4.3 GHz	2023
Malphite	Dell Precision 3620	16 GB	Intel E3-1240V6	4	8	4.1 GHz	2017
Poppy	Dell OptiPlex 5050	16 GB	Intel i5-7500	4	4	3.8 GHz	2017

ones. Their consumption is measured both via software measurements using Intel RAPL and using a smart plug measuring power. The use case involves training a basic CNN over several datasets with growing image sizes. `Pytorch` framework is used as a standard reference to establish comparison points with our proof-of-concept. Lastly, different `StarPU` schedulers are evaluated to compare their respective performances.

A. Platform

We evaluate four machines, denoted *Rammus*, *Senna*, *Malphite*, and *Poppy*, presented in Tab. I. All machines are desktop computers without GPUs, to the exception of *Senna*, a laptop with integrated graphics. During experiments, *Senna* runs with its integrated display turned off. *Rammus* and *Senna* are the two machines with heterogeneous CPU topologies. Whether running `Pytorch` or `DAHL`, we try to maximize the usage of each machine by using all threads. By default, `Pytorch` is able to use every threads, but `StarPU` requires compilation flags to leverage Hyper-Threading on *Malphite* in particular.

The power consumption of each machine is measured both via software, and physical methods. The `Alumet` [34] software retrieves RAPL energy counters of hardware components. However, in our case, desktop machines only report CPU and DRAM energy, in contrary to the laptop [37]. Software measuring adds a slight, static overhead on each machines. Physical measurements are obtained with the Aoetec Smart Switch 7, a smart plug that monitors energy consumption. The manufacturer reports ± 3 watts confidence interval[3]. Measurements are collected by an external Raspberry PI. Thus, it adds no overhead compared to software measuring. Lastly, note that every measurements (software or physical) are taken every 2 seconds, due to polling rate constraints on the smart plug.

B. Use case

For the use case, we picked a CNN, as the nature of computations for training such model are diverse: convolutional layers are computation intensive but require low memory to store the parameters, while it is the opposite for dense layers [26]. Also, convolutions offers a lot of parallelization opportunities, making the TGCS choice interesting. As a comparison reference, the same exact CNN architecture was reproduced on `DAHL` and `Pytorch`. It consists of basic layers: *Convolution, Relu, Max Pooling, Flatten, and Dense*. Note that random seeds and initial model parameters have been fixed for every experiments. Model accuracies are therefore equals for each dataset trained upon, giving reasonable comparisons on runtime and energy metrics. While interesting, the trade-off between energy and accuracy is out-of-scope of this article. Hyper-parameters are also fixed: 0.001 learning rate, Stochastic Gradient Descent (SGD) optimizer, 8192 samples, and 10 epochs.

Datasets involved are based on a high definition dataset, publicly available as "Fashion Product Images Dataset" [38]. Different versions of the dataset have been produced, by scaling down linearly the images in order to simulate various workload sizes. The term "pixels" refers to the total amount of values required to store an image in memory, and is equal to $height \times width \times channels$. Thus, each dataset contain a given image size, the smallest dataset containing images with $26 \times 34 \times 3$ pixels, the larger $234 \times 312 \times 3$.

Lastly, workload is defined as the amount of computational work required for an execution. It corresponds to the model training. This amount of work depends on the input size of data samples. Thus, in this paper, increasing workload, or dataset size, all refers to using larger images, which increases the total amount of computation required to train the CNN.

C. Reproducibility

Numerous precautions were taken to ensure the reproducibility of our experiments. First, every machine runs on a fresh install of NixOS 25.11 with a common, and minimal headless configuration. This ensures that the configuration and programs running on each machine are equivalent, for example, we used `Pytorch` version 2.5.1, and `StarPU` on master branch, as of April 23, 2026. This helps mitigating biases on energy measurements. Similarly, the machines have been used exclusively as part of the experiments. Thus, the out-of-the-plug energy only includes the idle consumption, in addition to the CNN training. Keeping the idle part is important, as we also want to compare the training cost, depending on the energy profile of each machines.

Concerning the case study itself, programs have been improved iteratively, so that both `Pytorch` and `DAHL` efficiently used resources on each machines. For example, *Malphite* and *Poppy* were memory-bounded on bigger datasets, so we implemented asynchronous data loaders to prevent this. However, Sec. V-B brings critics towards the efficient usage of machines. Lastly, all combinations of framework, scheduler, and dataset have been run randomly, and a warm-up phase have been included to mitigate thermal biases. Also, each experiment have been repeated five times to compute 95%

[3]https://aeotec.freshdesk.com/support/solutions/articles/6000219912-smart-switch-7-f-plug-technical-specification

confidence intervals. On average, the relative half-width of the confidence intervals was small across experiments, with a median of $\pm 0.28\%$ (mean $\pm 0.62\%$) for runtime, and $\pm 0.57\%$ (mean $\pm 0.96\%$) for energy consumption.

V. Results

In this section, we introduce experimental results showing different tendencies between heterogeneous and homogeneous CPU topologies when training a CNN. We first compare our proof-of-concept with the research standard framework `Pytorch`. We both showcase a default `Pytorch` configuration, and an optimized one that was chosen among the best configurations we tried. We evaluate their performances based on training time, average power and total energy consumption. Next, we study the behavior of different schedulers with the same metrics, in addition to the proportion of time spent executing tasks. Lastly, we emphasize the runtime speedups and energy savings of our different schedulers compared to a random-based scheduler.

A. Comparison with Pytorch

As stated before, two `Pytorch` configurations are used. The first one, `pytorch-regular`, is a default configuration that would be used by anyone if no execution-related parameters were tweaked. The second have been chosen among many configurations to find the best performing one. After trying Data Parallel (DP), Distributed Data Parallel (DDP), and `OpenMP` backend, we found out that oversubscribing threads ($2 \times \#CPU\ threads$) worked the best, hence the name `pytorch-morethreads`. For `DAHL`, we choose the `dmdar` scheduler, as it is recommended by `StarPU` when performance models are available.

On Fig. 2, we observe that only machines with homogeneous CPU topology show linear runtime on `pytorch-regular` as the datasets grows. Heteregeneous one have a differences up to $7.5\times$ slower runtime and $17\times$ more consumption than `pytorch-morethreads`, specifically on *Rammus*. This is explained by the fact that oversubscribing threads results in linear runtime and consumption no matter the CPU topology. This, however, comes at the cost of increasing context switches: $10\times$ more on *Rammus* and $5.6\times$ on *Senna*. CPU usage also reaches lower values on average, as `pytorch-regular` is around 90% while `pytorch-morethreads` is at 40% usage at most. Furthermore, the impact of configuration on an heterogeneous CPU is not the same on observed machines. On *Rammus*, we notice power differences between 125 W and 225 W, leading to significant consumption differences. However, *Senna*'s power only varies around 4 W, no matter the configuration. This only leads to runtime penalty, impacting less total consumption: at worse $1.3\times$ more consumption on *Senna* while it can reach $17\times$ more on *Rammus*. The low power consumption of *Senna* laptop, combined to a runtime similar to *Malphite*, leads to the lowest total consumption among all machines. Now comparing `DAHL` to `Pytorch`, runtime and consumption is consistently lower on `DAHL` no matter the machine or

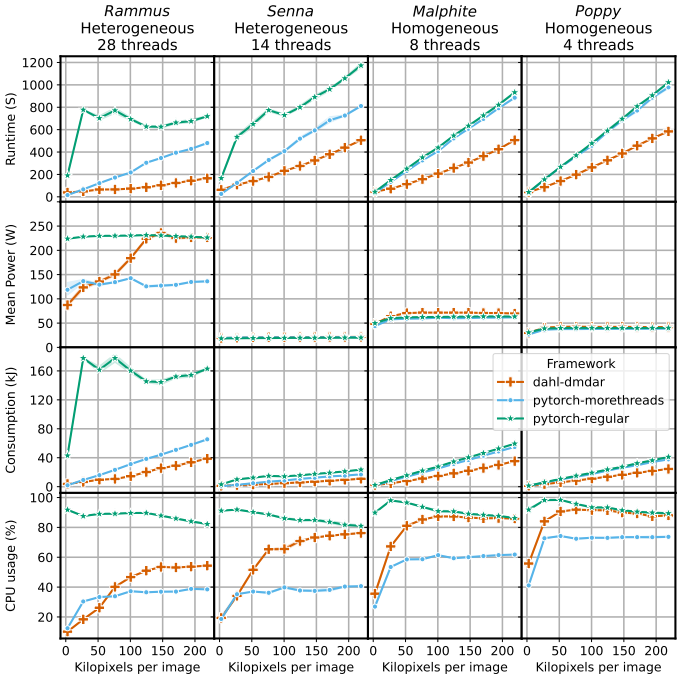

Fig. 2. Evaluation of `Pytorch` and `DAHL` on CNN training over datasets with growing image sizes.
Reading: training a CNN using `pytorch-regular`*, on Rammus server, on a dataset with images of* 50 *kilopixels, takes on average* 700 *seconds, draws* 230 *watts, and consumes* 160 *kilojoules.*

datasets. From left to right order of machines, the difference between `DAHL` and `pytorch-morethreads` consumption is up to $1.6\times$, $1.2\times$, $1.4\times$, and $1.5\times$ higher for `Pytorch`. More specifically, on *Rammus*, we notice that the average power grows as the images get larger, ranging from 85 W, to 243 W on average. This behavior is really different from both `Pytorch` configurations, where their power usage is either the lower or the upper bound. This can be explained by the CPU usage, which shows greater amplitudes on `DAHL` than on `Pytorch` configurations. Indeed, on `StarPU`, the scheduler is capable of allocating CPUs according to the workload. This is not the default behavior of `StarPU`. By default, it keeps CPU workers awake permanently to increase their reactivity, leading to full CPU and power usage. We used the `-enable-blocking-drivers` flag to permit CPU workers to sleep when unused.

B. Comparing schedulers

As seen in previous experiments, runs on heterogeneous CPU tend to have non-linear runtime despite workload (here images) growing linearly. In this case, 3 schedulers were impacted especially on small datasets: `ws`, `lws`, and `peager`. On *Rammus*, differences are up to $10\times$ slower compared to the `random` scheduler. On *Senna*, the impact is lower, with up to $4\times$ slower. Similarly, *Rammus* shows diverging mean powers, more or less pronounced over the schedulers, ranging from 50 W to 257 W. *Senna*'s power ranges around 4 W, *Malphite* 25 W, and *Poppy* 15 W. These power values help

us understand the consumption differences among machines. *Senna* is still the machine with the lowest consumption while achieving the same task. The difference is up to $4.8\times$, $3.5\times$, and $2.6\times$ more consumption compared to *Rammus*, *Malphite*, and *Poppy*.

In addition, to understand how effective the parallelization is achieved on each machine, we need to look at the time proportion metrics. The two last rows of Fig. 3 indicate the average proportion of time that each worker spend executing tasks or sleeping. Other time categories include, callbacks, overheads, scheduling, and waiting times due to data transfers. Nevertheless, the categories mentioned earlier added up to less than 1% of the total time. Executing and sleeping are therefore the primary states that constitutes our experiments. A large proportion of sleeping time may indicate a lack of parallelism. However, increasing batch size to augment parallelism did not lead to better execution time proportion. Furthermore, the sleeping time proportion reduces as image size increase, hinting towards a task granularity issue. In other words, *Rammus* and *Senna* may be oversized, for an efficient parallelization of such small workloads. This is illustrated by the fact that machines with higher performances tend to reach an executing time proportion superior to 50% later. Respectively from left to right: 150, 100, 50, 25 kilopixels on average. Fig. 4 introduces two metrics showing runtime speedup and energy savings to highlight relative gains in comparison to the results of the `random` scheduler, our baseline. This choice is motivated by the fact that this scheduler is the simplest one, shows predictable behaviors across all machines, and follows tendencies akin to `pytorch-morethreads`. As a reference, this last is on average 65% slower on all machine, and consumes 43% more energy compared to `random`. Also, note that the following interpretations will only take into account data points after the dashed vertical lines. We consider that trainings with more than 50% of time spent sleeping are less relevant. Additional experiments have been performed on *Rammus* to provide more insightful data.

On homogeneous CPUs, the best schedulers are work-stealing based, with up to 22% speedup on both machines, and up to 13% and 10% on *Malphite* and *Poppy*. For heterogeneous CPUs, `dmdar` shows good results on both machines, with up to 13% and 5% speedup, and up to 6% and 2.5% energy saving on *Rammus* and *Senna* respectively. Moreover, additional experiments performed on *Rammus* indicate decline in runtime gains while energy savings rise, on this specific scheduler at least. Some explanation is that `dmdar` execution proportion reaches a plateau after 50%, as well as the power usage at 225 W. In contrary, work-stealing and eager-based schedulers tend to obtain similar speedup gains ranging from 5% to 8%, but without any energy gains associated as they appear negative: from -12% to -15%. Additionally, *Senna* seems to benefit the most from eager-based schedulers with 17% speedup and 15% energy saving. These schedulers obtains comparable gains on homogeneous machines, respectively 17% speedup and 13% energy saving on *Malphite*, 10% and 5% for *Poppy*.

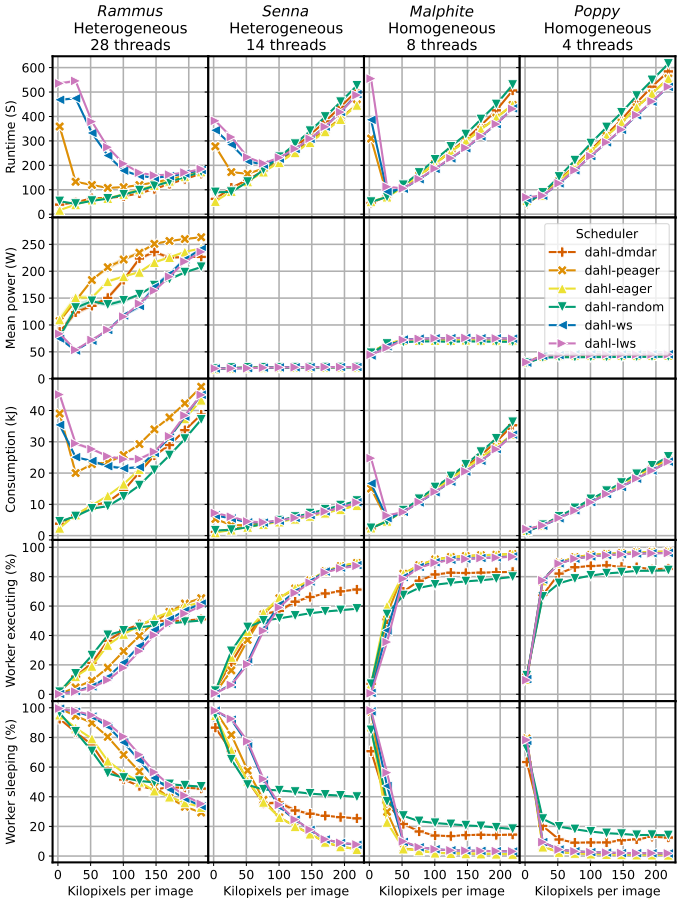

Fig. 3. Evaluation of `DAHL` using different `StarPU` schedulers on CNN training over datasets with growing image sizes.
Reading: training a CNN using the `lws` scheduler, on Rammus server, on a dataset with images of 100 *kilopixels, takes on average* 200 *seconds, draws* 115 *watts, consumes* 160 *kilojoules, and 20% of time is spent executing tasks, the remainder being mostly sleeping.*

To summarize, we observe that work-stealing-based schedulers have the best gains on homogeneous CPUs, while eager-based seems to be more effective on heterogeneous ones. *Rammus* results are more difficult to interpret, as those schedulers indeed improve runtime, but does not imply energy savings. The `dmdar` scheduler shows great adaptability no matter the topology, however it tends to fail to adapts to smaller workloads, and falls off on bigger workloads. In contrary, the random scheduler is globally good when running undersized workload and small tasks. Finally, it is worth noting that, despite `StarPU` focusing on heterogeneity, using it tends to benefit runtime and energy consumption even on homogeneous CPUs.

VI. Discussions

This study highlights results that we classify into three categories: the efficient usage of heterogeneous CPUs, the adaptability of TGCS, and the runtime versus energy trade-off. In the scope of our experiments (use case, selected datasets, and machines) we observe that `Pytorch` requires additional configuration to obtain efficient usage of heterogeneous CPUs.

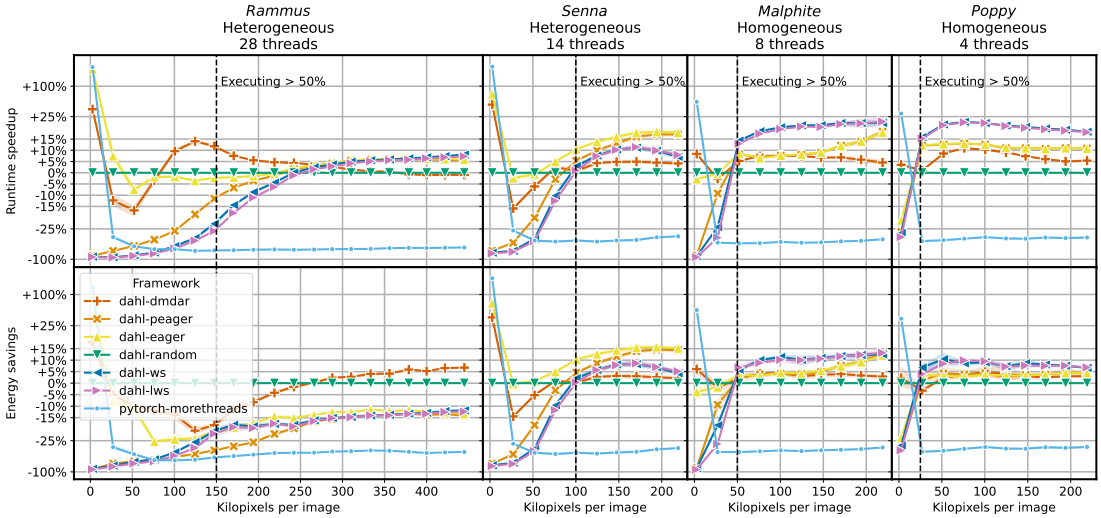

Fig. 4. Comparison of frameworks on CNN training over datasets with growing image sizes, in relation to a baseline, the `random` scheduler. Dashed vertical lines delimits the point where time spent executing tasks exceeds 50% (right of the line) on each machine.
Reading: training a CNN using the `ws` scheduler, on Malphite server, on a dataset with images of 150 *kilopixels, is on average* 21% *faster and saves* 10% *energy consumption. However on Senna, gains are about about* 10% *runtime and* 7% *energy.*

Oversubscribing threads have shown as the best solution runtime and energy wise. However, it does increase context switches and seems to cap the CPU usage bellow 50%. To this extent, TGCS-based experiments provide better effectiveness. We notice standard context switches values, while having wider ranges of CPU usage depending on workload size. There is no need to oversubscribe threads, as each worker can be bound to physical threads. Furthermore, last experiments shown that on average, our best `Pytorch` configuration still consumed 43% more energy, and was 65% slower than `DAHL`. Such results are encouraging towards the usage of TGCS in ML. Nonetheless, results should be contextualized regarding the nature of the comparison. Indeed, `Pytorch` is a general purpose ML framework focusing on multiple accelerators and backends. While `DAHL` is not limited to CNNs, it clearly does not suffer from architecture's overheads required to support diverse execution contexts. However, it is not limited to CPU only, as `StarPU` also permits multi-CPU and multi-GPU execution, and distributed computing with MPI. Extending `DAHL` with this capabilities is on-going future work.

On another hand, we also noticed differences among TGCS' schedulers themselves. Runtime gains can be up to 20% and energy savings up to 15%. In particular, work-stealing-based schedulers shows the best results on most of machines, the only exception being on the only laptop, *Senna*. It remains uncertain why eager-based schedulers performs better on the laptop. As these schedulers does not permit data prefetch, it might indicate that this laptop is less impacted by cache misses. However, in all scenario, work-stealing and `peager` schedulers shown the worse results on small workloads. The more powerful server, *Rammus*, demonstrates the adaptiveness of TGCS, as the average CPU usage and power grows depending on the workload, with wider ranges than `Pytorch` experiments. This behavior is still true on other machines for the CPU usage, but to a lesser extent on their power, as it is machine dependent. Typically, desktops consumes more and have wider power ranges.

Last, these experiments emphasize the trade-off between faster and more frugal execution. For example, on the largest dataset, the training on *Rammus* can be up to 2.5× faster but can consume up to 4× more energy. The laptop exhibits the best overall trade-off, thanks to decent runtime and consumption. Also, we noticed energy gains as runtime declined on *Rammus* with `dmdar`. Previous HPC work highlights this behavior, as the most energy-efficient solution is not always the one with the shortest runtime [39]. Finally, limits of this methodology include the low samples of machines. In particular, more heterogeneous CPUs could be tested to validate observations made on `Pytorch` configurations. Additionally, we recognize that machines that comes with heterogeneous CPUs are in general much more recent, which implies better energy efficiency. This itself is not a problem for individual analysis, but more for comparing efficiency between machines.

VII. Conclusion

This work shows that Heterogeneous CPU topologies introduce new challenges in terms of scheduling, and that it has a real impact on energy efficiency of ML frameworks. By conducting experiments on multiple machines with different CPU topologies, and training a CNN with datasets of varying sizes, we demonstrate that real differences between topologies. Homogeneous ones are already greatly optimized by default, and follows a linear grow both in runtime, regardless of the workload used. In contrary, heterogeneous ones require additional tweaking in order to fully utilize CPU resources. One way might be to oversubscribe threads, but at the cost of capping CPU usage and high context switches. Another one is to use a TGCS such as `StarPU` which takes into

account CPU heterogeneity. This gives a lot of potential to switch between multiple schedulers in order to find the correct trade-off between runtime and energy gains.

For future works, similar experiments could be performed on GPUs to determine whether comparable observations could be made against `Pytorch` and the different `StarPU` schedulers. In addition, this would allow `StarPU` to schedule tasks on both CPUs and GPUs, to evaluate data transfers overhead over the energy consumption. Also, Adding multi-node capabilities to `DAHL` could provide a solution for training on heterogeneous clusters in an energy efficient way.

REFERENCES

[1] C. E. Tripp, J. Perr-Sauer, J. Gafur, A. Nag, A. Purkayastha, S. Zisman, and E. A. Bensen, "Measuring the energy consumption and efficiency of deep neural networks: An empirical analysis and design recommendations," *arXiv preprint arXiv:2403.08151*, 2024.

[2] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021.

[3] P. Li, J. Yang, M. A. Islam, and S. Ren, "Making ai less 'thirsty'," *Commun. ACM*, vol. 68, no. 7, p. 54–61, 2025.

[4] S. Falk, D. Ekchajzer, T. Pirson, E. Lees-Perasso, A. Wattiez, L. Biber-Freudenberger, S. Luccioni, and A. van Wynsberghe, "More than carbon: Cradle-to-grave environmental impacts of genai training on the nvidia a100 gpu," *arXiv preprint arXiv:2509.00093*, 2025.

[5] A. Berthelot, E. Caron, M. Jay, and L. Lefèvre, "Estimating the environmental impact of generative-ai services using an lca-based methodology," *Procedia CIRP*, vol. 122, pp. 707–712, 2024.

[6] Z. Cai, X. Yan, K. Ma, Y. Wu, Y. Huang, J. Cheng, T. Su, and F. Yu, "Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 1967–1981, 2022.

[7] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Accpar: Tensor partitioning for heterogeneous deep learning accelerators," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 342–355, 2020.

[8] C. Nie, J. Maghakian, and Z. Liu, "Cannikin: Optimal adaptive distributed dnn training over heterogeneous clusters," in *Proceedings of the 25th International Middleware Conference*, Middleware '24, p. 299–312, Association for Computing Machinery, 2024.

[9] Y. Ding, N. Botzer, and T. Weninger, "Hetseq: Distributed gpu training on heterogeneous infrastructure," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 17, pp. 15432–15438, 2021.

[10] T. Brilland, E. Fangeat, J. Meyer, and M. Wellhof, "Évaluation de l'impact environnemental du numérique en france," 2025.

[11] O. Beaumont, J.-F. David, L. Eyraud-Dubois, and S. Thibault, "Staronnx: a dynamic scheduler for low latency and high throughput inference on heterogeneous resources," in *European Conference on Parallel Processing*, pp. 375–386, Springer, 2024.

[12] O. Beaumont, J.-F. David, L. Eyraud-Dubois, and S. Thibault, "Exploiting processor heterogeneity to improve throughput and reduce latency for deep neural network inference," in *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 37–48, 2024.

[13] A. Mikhalev, A. Katrutsa, K. Sozykin, and I. Oseledets, "Nntile: a machine learning framework capable of training extremely large gpt language models on a single node," 2025.

[14] R. Reyes and V. Lomüller, "Sycl: Single-source c++ accelerator programming," in *Parallel Computing: On the Road to Exascale*, pp. 673–682, IOS Press, 2016.

[15] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, p. 298, 2008.

[16] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[17] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 338–347, 2016.

[18] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," in *European Conference on Parallel Processing*, pp. 863–874, Springer, 2009.

[19] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2022.

[20] P. Cardosi and B. Bramas, "Specx: a c++ task-based runtime system for heterogeneous distributed architectures," *PeerJ Computer Science*, vol. 11, p. e2966, 2025.

[21] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[22] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks.," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[23] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, *et al.*, "Pytorch distributed: Experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[24] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, *et al.*, "Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 559–578, 2022.

[25] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in accelerating convolutional neural networks," in *International Conference on Machine Learning*, pp. 2274–2283, PMLR, 2018.

[26] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[27] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Comput. Surv.*, vol. 46, no. 4, 2014.

[28] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A.-C. Orgerie, and M. Quinson, "Predicting the energy-consumption of mpi applications at scale using only a single node," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 92–102, 2017.

[29] S. Rincé and A. Banse, "Ecologits: Evaluating the environmental impacts of generative ai," *Journal of Open Source Software*, vol. 10, no. 111, p. 7471, 2025.

[30] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," *arXiv preprint arXiv:2007.03051*, 2020.

[31] B. Petit, "Scaphandre."

[32] V. Schmidt, K. Goyal, A. Joshi, B. Feld, L. Conell, N. Laskaris, D. Blank, J. Wilson, S. Friedler, and S. Luccioni, "Codecarbon: estimate and track carbon emissions from machine learning computing," *Cited on*, vol. 20, 2021.

[33] I. SPIRALS, "Powerapi."

[34] G. Raffin, D. Trystram, and O. Richard, "Alumet: a modular framework to standardize the measurement of energy consumption," in *Workshop on Performance and Energy Efficiency in Concurrent and Distributed Systems*, Springer, 2025.

[35] H. Valera, F. Ravat, P. Roose, J. Song, and N. Vallès-Parlangeau, "Ecofloc: outil de mesure énergétique multi-composants," in *CNRIUT 2025 Bayonne-Pays Basque*, 2025.

[36] M. Jay, V. Ostapenco, L. Lefevre, D. Trystram, A.-C. Orgerie, and B. Fichel, "An experimental comparison of software-based power meters: focus on cpu and gpu," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 106–118, 2023.

[37] G. Raffin and D. Trystram, "Dissecting the software-based measurement of cpu energy consumption: A comparative analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 36, pp. 96–107, 2025.

[38] P. Aggarwal, "Fashion product images dataset." https://www.kaggle.com/datasets/paramaggarwal/fashion-product-images-dataset, 2019. Accessed: 2026-02-17.

[39] B. Subramaniam and W.-c. Feng, "Statistical power and performance modeling for optimizing the energy efficiency of scientific computing," in *2010 IEEE/ACM Int'l Conference on Green Computing and Commu-*

nications & Int'l Conference on Cyber, Physical and Social Computing, pp. 139–146, 2010.