Performance Analysis of Distributed GPU-Accelerated Task-Based Workflows

Marcos N. L. Carvalho[*]
marcos.nogueira@upc.edu
UPC, Barcelona, Spain
NKUA & Athena RC, Athens, Greece

Anna Queralt
anna.queralt@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Oscar Romero
oscar.romero@upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Alkis Simitsis
alkis@athenarc.gr
Athena Research Center
Athens, Greece

Cristian Tatu
cristian.tatu@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

Rosa M. Badia
rosa.m.badia@bsc.es
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

We present an empirical approach to identify the key factors affecting the execution performance of task-based workflows on a High Performance Computing (HPC) infrastructure composed of heterogeneous CPU-GPU clusters. Our results reveal that the execution performance in distributed GPU-accelerated task-based workflows highly depends on several interrelated factors regarding the task algorithm, dataset, resources, and system employed. In addition, our analysis identifies key correlations among these factors, presents novel observations, and offers guidelines toward designing an automated method to handle task-based workflows in modern, high-compute capacity, CPU-GPU engines.

1 INTRODUCTION

Data Science (DS) pipelines are essential for many software systems today. Such pipelines are composed of multiple processing stages that perform different tasks to move data from one stage to a next stage [7]. The relationship between these stages creates complex workflows, where data preparation, training and evaluation of Machine Learning (ML) models are just examples of processing that is frequently done over big datasets. To process large amounts of data efficiently, distributed and parallel applications are required and *task-based workflows* provide a high-level programming abstraction to develop such applications [55]. High Performance Computing (HPC) clusters are being widely used to process DS workloads because of the massive parallelism enabled by distributed architectures [54]. In addition, improvements in the hardware industry have increased the popularity of accelerators, such as graphic processing units (GPUs), making modern distributed infrastructures even more heterogeneous [2]. This evolution has led to distributed GPU-accelerated task-based workflows [6], which provide massive processing power to users by leveraging both *task-level parallelism* of distributed CPUs and *thread-level parallelism* of GPUs. In this context, tasks are distributed and processed in parallel in CPUs and each task, internally, has its threads parallelized by GPUs.

Considering the performance of such workflows, we face three core challenges: *(i) Ad hoc design:* Lacking concrete guidelines and optimization heuristics, the workflow developer often resorts to intuition to assign tasks to GPU accelerators. And oftentimes, due to the huge design space of execution parameters (a.k.a. factors), the developer might exhaustively rerun representative workloads in order to identify efficient execution settings [10]. In practice, even expert developers spend a considerable amount of time searching for appropriate, but often sub-optimal, settings. *(ii) Resource wastage:* Finding fitting settings results into efficient resource usage. On the other hand, a poor combination of execution parameters leads to load imbalance and waste of resources. For example, a non desirable situation would be to keep the CPUs busy while the GPUs stay idle during workflow processing. *(iii) Complexity:* Previous studies of the problem have focused on individual factors that could affect execution performance, such as block size, dataset input size, etc. In our study, we argue and show that this is not sufficient, as in real-world scenarios the workflow execution performance is typically affected by a *combination of factors*.

To motivate the discussion, consider the following example that attempts to improve the performance of a task-based workflow using GPU accelerators. The workflow represents a CPU-based and a GPU-accelerated version of a distributed implementation of the K-means algorithm[1]. In this experiment, we consider single task (on 1 CPU core and 1 GPU device) and parallel tasks (on all CPU cores and GPU devices) execution. Generally, task processing involves data computation, represented by the task user code, and data movement (we detail these in Section 3.3). Figure 1 shows the execution performance at different task processing stages: (i) For a single task, a 5.69x speedup of GPU over CPU is achieved when only the parallel fraction of a task user code is considered; this is the fraction of the task user code where threads can be parallelized on a GPU. (ii) The speedup, again for a single task, is reduced to 1.24x when the total execution of the task user code is considered; this includes the CPU-GPU communication overhead and the serial (i.e., non-parallel) fraction of the task as well. (iii) Interestingly, for parallel tasks execution (i.e. all tasks being distributed and processed in parallel), we observe that the overall performance of GPUs is worse than CPUs (-1.20x speedup). Arguably, as a partial analysis of the performance of GPUs vs. CPUs in distributed task-based workflows may produce misleading or incomplete results, we need to conduct a more thorough and principled analysis. And this exactly is the goal of our work.

To efficiently run distributed task-based workflows, we need to identify and characterize the factors affecting the performance at

[*]The author pursues a joint PhD degree under the auspices of DEDS (No 955895), a Horizon 2020 MSCA ITN, and he is co-affiliated with Universitat Politècnica de Catalunya in Spain, Athena Research Center and National and Kapodistrian University of Athens (a degree awarding institute for Athena R.C.) in Greece.

[1] The experiment involves a 10 GB dataset processed by 256 tasks distributed on a cluster with 128 CPU cores and 32 GPU devices. For more details, see Section 4.

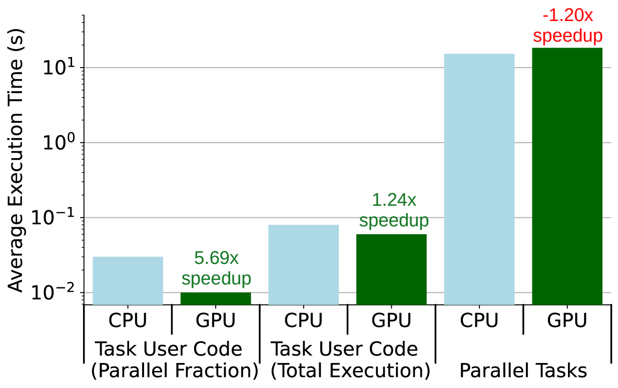


Figure 1: Performance of distributed K-means at different processing stages on CPUs and GPUs

all task processing stages, as those presented in Figure 1. Earlier work has identified a number of factors affecting the performance of heterogeneous CPU-GPU processing in a multitude of scenarios, as illustrated in Figure 2. Different performance limitations arise depending on the infrastructure used, such as single, server machines and compute clusters.

On a single machine, the main limitations are CPU-GPU *data transfer bottleneck* and *device speedup*. Given a task user code, a potential mitigation technique to overcome CPU-GPU communication would be to increase computation on GPUs using techniques such as staged pipeline and zero-copy [60]. The more computation, the higher the device speedup. The amount of computation in a task heavily depends on its *arithmetic intensity*, i.e., the number of math operations divided by number of bytes read [10, 45], and the *task granularity*, i.e., the amount of data to process [78]. However, tasks may involve parallel fraction processing that exploits thread-level parallelism on GPUs, and serial fraction processing executed on CPUs. Hence, the parallel fraction processing within tasks is a key factor to consider in order to fully utilize GPUs, as it defines the abundance of thread-level parallelism [22]. Earlier work has studied the issues of CPU-GPU communication and parallel fraction independently, but has not considered both problems in tandem, exactly as it happens in real settings.

On compute clusters, tasks are distributed and processed in parallel on multiple nodes, thus, scaling out *task-level parallelism* [72] and providing *memory robustness* [62] to GPUs by breaking the input dataset into chunks. Such an infrastructure adds non-negligible overheads, including storage I/O [27, 38, 69, 70], network I/O [6, 26, 34, 78], and task scheduling [2, 25].

Regardless of the infrastructure used, task granularity is frequently considered as a key performance factor. However, increasing task granularity alone is not always the best strategy to improve the performance of distributed task-based workflows, as it increases thread-level parallelism at the cost of reducing task-level parallelism, which in turn leads to load imbalance between CPUs and GPUs. Our empirical analysis reveals that the performance of distributed GPU-accelerated task-based workflows is a result of many interrelated factors involving, beside task granularity, CPU-GPU data transfer, task parallel fraction, storage I/O, network I/O, task scheduling, etc. We argue that focusing on a single factor separately (e.g. increasing task granularity to maximize device speedup) as the current related work suggests, leads to sub-optimal designs. On the other hand, combining multiple performance factors (e.g. multi-level parallelism) to optimize task-based workflows in heterogeneous CPU-GPU, distributed environments involves a non trivial design complexity, which makes this a quite challenging problem. The literature misses a detailed performance analysis to characterize the performance of task-based workflows and study how to balance thread-level and task-level parallelism to maximize resource utilization.

Our contributions. We present a systematic performance analysis of task-based workflows and make the following contributions:

- A systematic analysis of *thread-level parallelism*; our results reveal that the gains provided by GPUs are highly affected by the parallel fraction processing within tasks.
- A systematic analysis of *task-level parallelism*; our results reveal that depending on the task granularity, scheduling policy, and storage architecture used, system overheads can be significantly high, dominating the total execution time and eliminating the potential gain of GPU processing.
- A method to identify what are the factors affecting the performance of task-based workflows, how these relate to each other and to the algorithms, datasets, resources, and distributed execution system employed; our results indicate that certain execution parameters are highly correlated with the performance of task-based workflows and can be considered as key factors.
- We present novel observations and offer guidelines toward designing an automated method to handle task-based workflows in modern, high compute capacity, CPU-GPU engines.

Structure. The rest of the paper is organized as follows. Section 2 presents the related work. Sections 3 and 4 describe background concepts and the method used in our analysis, respectively. Section 5 presents our experimental analysis, and Section 6 concludes the paper.

2 RELATED WORK

The literature about heterogeneous CPU-GPU processing is wide and several approaches have been proposed to improve the performance of CPU-GPU processing. In Figure 2, we present a classification of the state of the art on CPU-GPU processing and highlight in red the related work that is within the scope of our analysis.

In particular, the related work on heterogeneous CPU-GPU has focused on frameworks [27, 31, 71], libraries [20, 56, 66], schedulers [10, 25, 30], cost-based models [10, 53, 62], programming models [3, 34, 69], and benchmarks [14, 78]. Previous performance analysis studies focused on specific applications like deep learning models, ETL processes, image classification, and query performance [9, 10, 29, 32, 42, 63]. Our study targets generic task-based workflows in CPU-GPU environments, follows a bottom-up analysis from thread-level to task-level parallelism, and considers a multiplicity of factors affecting workflow performance.

Previous works have also explored different ways to use GPUs, for example, as the primary processor [27, 62], as an accelerator [73, 78] or in the context of heterogeneous CPU-GPU processing [32, 59, 69, 71]. Regarding the application field, several papers have demonstrated interesting results by using GPUs to accelerate database query processing [10, 32, 62, 71] and data-intensive analytics applications with task-based workflows [2, 3, 9, 29, 42, 78], dataflows [15, 57], and graph processing [39, 76].

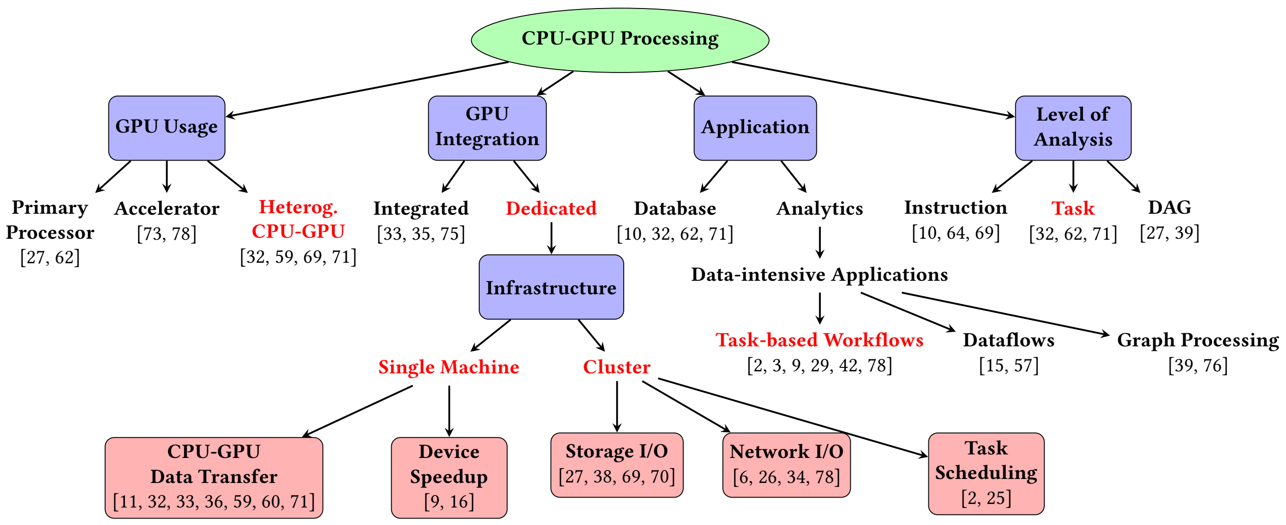

CPU-GPU Processing
GPU Usage
GPU Integration
Application
Level of Analysis
Primary Processor [27, 62]
Accelerator [73, 78]
Heterog. CPU-GPU [32, 59, 69, 71]
Integrated [33, 35, 75]
Dedicated
Database [10, 32, 62, 71]
Analytics
Instruction [10, 64, 69]
Task [32, 62, 71]
DAG [27, 39]
Infrastructure
Data-intensive Applications
Task-based Workflows [2, 3, 9, 29, 42, 78]
Dataflows [15, 57]
Graph Processing [39, 76]
Single Machine
Cluster
CPU-GPU Data Transfer [11, 32, 33, 36, 59, 60, 71]
Device Speedup [9, 16]
Storage I/O [27, 38, 69, 70]
Network I/O [6, 26, 34, 78]
Task Scheduling [2, 25]

Figure 2: A taxonomy for CPU-GPU processing (the scope of our study is highlighted in red text)

Our focus is on analyzing the impact of GPUs on task-based workflows, which are nowadays commonly used in various applications, such as data science pipelines. Different levels of analysis can be found in the literature, spanning instruction-level optimization [10, 64, 69], task-level [32, 62, 71] and distributed acyclic graph (DAG) evaluation [27, 39].

Some approaches investigate using integrated GPUs [33, 35, 75]. However, the majority of previous work considers dedicated GPUs due to the higher processing capacity compared to integrated GPUs [60]. Depending on the infrastructure used, i.e. single machine or cluster, different limitations may arise for dedicated GPUs.

Limitations on single server machine. One of the main limitations reported in the literature about the performance of heterogeneous CPU-GPU processing is the data transfer bottleneck [11, 32, 33, 36, 59, 60, 71]. This bottleneck has been studied extensively and several techniques have been proposed to alleviate it. The techniques include overlapping transfer with execution, data compression, approximation, caching, data locality, single-pass algorithms, heterogeneous execution and faster system bus [60]. Some studies have also proposed increasing the amount of data to process (i.e., task granularity) as much as the GPU memory supports in order to increase the throughput and the device speedup [9, 16].

However, recent publications have demonstrated that this strategy is not always the best option to improve the performance of heterogeneous CPU-GPU environments [32, 61]. Additionally, the parallel fraction processing within a task is a factor that can limit performance as much as the CPU-GPU communication. A theoretical analysis of the parallel fraction is done in [53], but there is no empirical study about it in the literature. Moreover, no previous work has evaluated the impact that these limitations all together may cause in the performance of GPU-accelerated tasks. Our study reveals that it is worth using GPUs when parallel fraction processing time is large enough to overcome not only CPU-GPU processing time, but also serial fraction processing time.

Limitations on cluster deployment. Distributed environments enable scaling out task processing over several nodes at the cost of adding potential bottlenecks, such as data storage I/O [27, 38, 69, 70], network I/O [6, 26, 34, 78], and task scheduling overheads [2, 25]. Although these are relevant limitations to consider when assessing the performance of parallel task processing, there is no study on how to tune task granularity to balance task-level and thread-level parallelism and smooth such overheads. In addition, there is no work showing how the scheduling policy used and the storage architecture (i.e. local or shared disk) may affect these overheads.

Our study shows that such factors must be considered in tandem in order to distribute tasks efficiently on available CPU cores while at the same time utilizing GPUs. To the best of our knowledge, ours is the first study to relate multiple factors to the execution performance of distributed GPU-accelerated task-based workflows. In the past, other works have considered multiple factors but not all together and not for distributed GPU-accelerated workflows [2, 6].

3 PRELIMINARIES

Several systems facilitate the development and execution of parallel applications on distributed infrastructures (e.g, clusters, clouds, containerized platforms) such as Apache Spark [74] and COMPSs [4]. They all share a similar flow. The developer submits an application (e.g., matrix multiplication, K-means, neural network, etc.) to such a system, which then generates a Directed Acyclic Graph (DAG) based on the data dependencies between its tasks (e.g., math operations, such as dot product, blocked matrix multiplication, etc.). The system handles parallelism and distribution of tasks by scheduling and allocating resources to execute the tasks in heterogeneous processors (e.g., CPU or GPU) of a cluster. Furthermore, the required data to process the tasks is accessed through a storage architecture. Although HPC architectures typically decouple processing from storage using shared disks, the option of using local disks is also available. An overview of a typical distributed task-based workflow processing system is illustrated in Figure 3. Next, we describe the core components of such a system and set the basis of our analysis.

3.1 DAG creation

In distributed GPU-accelerated task-based workflows, an application is broken into small manageable parts defined as tasks. Each task takes an input data, perform some calculation over it, and

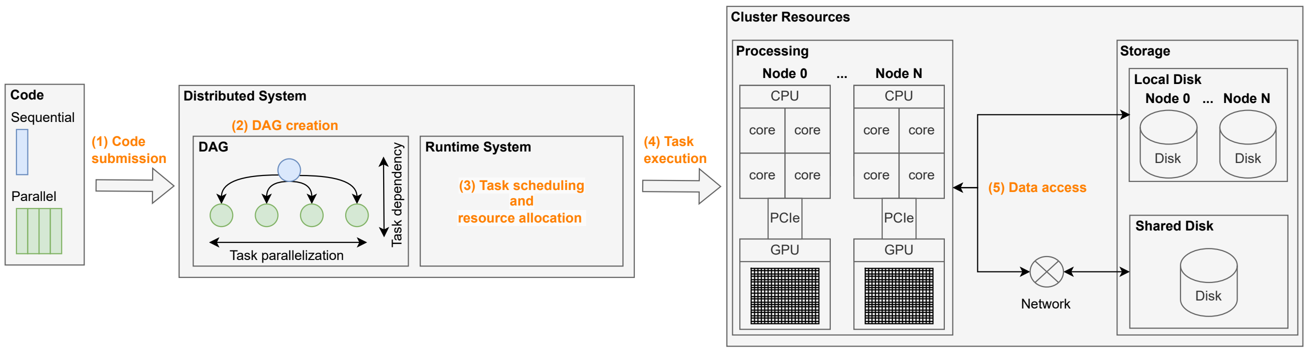


Figure 3: An overview of a distributed task-based workflow processing system

return an output to be used by the subsequent dependent tasks, if any. When an application code is sent to a processing system (e.g., COMPSs), the data dependencies between tasks are automatically identified and an execution DAG is generated, composing an execution workflow. In this DAG, the vertices represent tasks and the edges represent dependencies. The shape of the DAG reveals key characteristics of task processing. In particular, the width of the DAG shows the degree of parallelism (i.e., #parallel tasks generated) and the height of the DAG shows the degree of task dependency.

3.2 Task scheduling

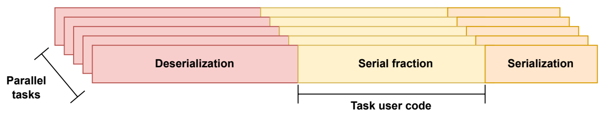
The system runtime schedules the tasks of a DAG that are free of dependencies on the available resources. In general, various scheduling policies are typically available prioritizing aspects, such as task generation order and data locality. Depending on the scheduler selected, the parallel task execution time may vary due to scheduling overheads, which may be low (e.g., for prioritizing task generation) or high (e.g., when considering data locality).

3.3 Task execution

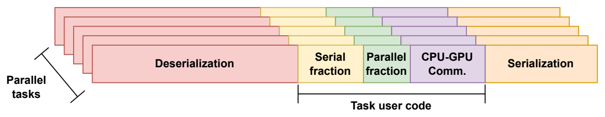
Figure 4 shows an abstract illustration of the typical processing stages of a task, namely the task user code that relates to data computation, and serialization and deserialization that relate to data movement. The task user code might be serial (i.e., runs single-threaded) and/or parallel (i.e., multi-threaded). A *serial task* only contains serial code. A *partially parallel task* contains both serial and parallel code fractions, and a *fully parallel task* only parallel code. Besides data computation, the user code also includes inter/intra-CPU, inter/intra-GPU, and CPU-GPU communication functionality. For the scope of this work, here we only consider CPU-GPU communication; i.e., the overhead to move data between CPUs and GPUs over a bus.

There are two levels of parallelism in task execution: *thread-level parallelism* and *task-level parallelism.*

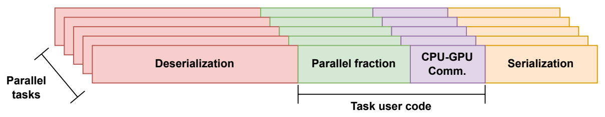
Thread-level parallelism. This takes place in the parallel fraction of a task user code. Although thread parallelism is feasible in CPUs (e.g., internally multi-threaded or parallelized with OpenMP [46]), several frameworks recommend assigning a task to a single CPU to avoid CPU over-subscription and hence, leverage full task parallelism in CPUs. Our initial micro-benchmarks corroborate this practice, but we plan to investigate this further in future work. Hence, we consider that serial tasks are assigned to CPUs, and partially or fully parallel tasks to GPUs, which accelerate them with thread parallelism. Note, that a GPU-accelerated task is not assigned directly to a GPU; it is first processed in a



(a) Serial task



(b) Partially parallel task



(c) Fully parallel task

Figure 4: Abstract task processing stages

CPU core to deserialize data, then a GPU executes the parallel code, and the result is sent to a CPU core to serialize the output.

Task-level parallelism. Multiple tasks can be executed in parallel. The degree of task parallelism is affected by *task dependencies* and *available resources.* Independent tasks are processed in parallel on available CPU cores in the cluster. Tasks with dependencies are processed sequentially, as soon as the execution of their dependencies has finished. Task parallelism is limited by the amount of available GPU devices in the cluster. Since cluster nodes typically have fewer GPU devices than CPU cores, GPU-accelerated tasks have lower degree of task parallelism. For example, in a cluster containing 128 CPU cores and 32 GPU devices, we can execute in parallel a maximum of 128 CPU-based tasks and only 32 GPU-accelerated tasks. Still, although GPU-accelerated tasks reap reduced benefits from task parallelism, they achieve considerable speedups due to thread parallelism. And this creates a critical trade-off between thread-level and task-level parallelism.

3.4 Data access

Some tasks may need to read from and write data to a storage device (e.g., disk or memory), which results to storage I/O and data transfer overheads e.g., due to data deserialization and serialization, respectively. The overhead varies depending on the storage architecture. For example, with local disks the data is accessed

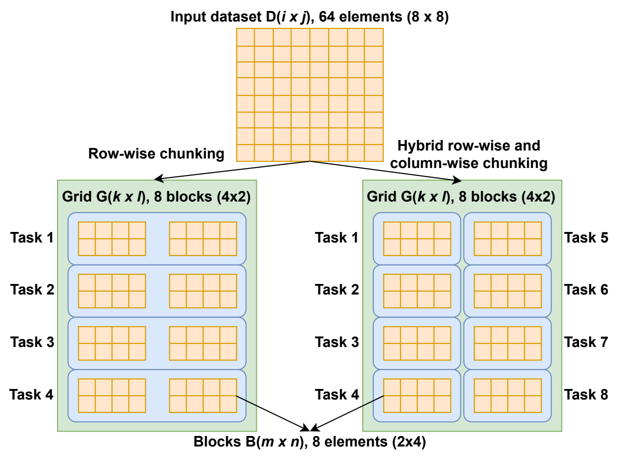

Input dataset D(i x j), 64 elements (8 x 8)
Row-wise chunking
Hybrid row-wise and column-wise chunking
Grid G(k x l), 8 blocks (4x2)
Grid G(k x l), 8 blocks (4x2)
Task 1
Task 2
Task 3
Task 4
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Task 7
Task 8
Blocks B(m x n), 8 elements (2x4)

Figure 5: Example data partitioning and task parallelization

locally without having a communication overhead. With shared disks, processing and storage are decoupled and the data is accessed from a shared file system accessible from all nodes of the network, which however (from a performance perspective) adds considerable communication overhead due to network latency, resource contention, etc.

3.5 Programming model

Without loss of generality, in our analysis, we consider that the input data is in the form of a matrix. This is in accordance with typical applications in parallel programming and popular libraries, such as CUDA [44] and dislib [17].

The processing system splits the input dataset (here, matrix) into blocks. And the blocks are typically organized in grids (e.g., ds_array in dislib, grid in CUDA, etc.). Figure 5 shows an example partitioning scheme for an input dataset comprising 64 elements in an 8x8 matrix. Assuming a block size of 8 elements organized in 4 columns and 2 rows, the grid will contain 8 blocks. Depending on the application (e.g., an algorithm), a chunking policy determines how to organize the blocks of a grid and assign them to tasks (task parallelism). We refer to task granularity as the blocks of the grid assigned to a single task. Blocks sent to GPU for processing are further chunked to get parallelized at a thread level (thread parallelism) using specialized libraries (e.g., CuPy for CUDA [20]).

Note that most systems do not provide a direct way to control how many tasks are spawned. Still we can achieve this indirectly by tuning the block size, which is determined by the developer. In our experiments, we set the task granularity to one block per grid to control the exact number of tasks get spawned (see also Section 4.4.4). Therefore, large blocks result into coarse-grained tasks and small blocks result into fine-grained tasks.

Formally, let $\boldsymbol{D}_{i\times j}$ be the input dataset with i rows and j columns, $\boldsymbol{B}_{m\times n}$ be a block, and $\boldsymbol{G}_{k\times l}$ be a grid with k blocks per row and l blocks per column. The size of $\boldsymbol{D}_{i\times j}$ (i.e., $i \times j$) gives the total number of elements of the input dataset to be processed, the size of $\boldsymbol{G}_{k\times l}$ (i.e., $k \times l$) gives the grid dimension (i.e., the total number of blocks per grid) and the size of $\boldsymbol{B}_{m\times n}$ (i.e., $m \times n$) gives the block dimension (i.e. the total number of elements per block). The relationship between these sizes is described by Eq. (1):

$$i = k \times m, j = l \times n, \tag{1}$$

To define a grid, the block dimension should be specified in the application. Thus, assuming that $\boldsymbol{D}_{i\times j}$ and $\boldsymbol{B}_{m\times n}$ are given, Eq. (1) can be rewritten as Eq. (2):

$$k = \frac{i}{m}, l = \frac{j}{n}, \tag{2}$$

Note that k and l are inversely proportional to m and n, respectively. This relationship reveals a trade-off between task-level and thread-level parallelism. The larger the block dimension, the smaller the grid dimension, which increases the task granularity (and, hence, thread parallelism, as now the block contains more elements to process) and reduces the number of generated tasks (and, hence, task parallelism). On the contrary, the smaller the block dimension, the larger the grid dimension, which minimizes the degree of thread parallelism and maximizes the degree of task parallelism. Consequently, these relationships are key to achieve a balanced degree of task-level and thread-level parallelism. However, there are two constraints. First, the memory size of a block must fit in the memory of a processor to avoid undesirable effects; e.g., out-of-memory (OOM) errors. Second, the block dimension ($k \times l$) cannot be greater than the input dataset dimension ($i \times j$).

4 OUR ANALYSIS METHOD

Increasing task granularity is not the only strategy to improve performance of distributed GPU-accelerated task-based workflows. We argue that additional factors should also be considered. To validate our hypothesis we conduct a performance analysis adopting the systematic method proposed by Jain [28].

4.1 Evaluated algorithms

We consider task-based workflows that represent data-intensive applications and run on a heterogeneous CPU-GPU cluster with dedicated GPUs. A task-based performance analysis relates to how the low-level tasks in each algorithm are processed by a distributed system. From this aspect, the algorithm's logic is orthogonal to how the underlying tasks implementing it (we don't measure how these tasks are produced) are being processed (this is what we measure in our analysis). In this setting, the parallel fraction of the user code is of paramount importance to task-based analysis (see also Section 5). Hence, our criterion to choose representative algorithms is the level of parallelism employed in each algorithm.

To this end, we consider two families of algorithms based on how they perform data computation in the task user code employing serial and/or parallel processing fractions (see Section 3.3): (a) *Fully parallelizable algorithms* involve fully parallel tasks whose user code can be fully parallelized in a GPU device (see Figure 4c). And (b) *Partially parallelizable algorithms* involve partially parallel tasks whose user code contains both serial and parallel fractions (see Figure 4b). In our analysis, we employ two representative algorithms of each family: an algorithm from family (a) that involves only parallel task execution vs. one from family (b) that has a low ratio of parallel / serial code in its task execution. For the first case, we use *Matrix Multiplication* (Matmul), which fulfills our criterion and is also a fundamental operation in many ML/DL techniques, including LLMs, PCA, SVD, linear regression, etc. For the second case, we use *K-means*, another widely used and easy to understand algorithm, whose ratio of parallel / serial code is low and hinders its parallel execution. Note that any other fully parallelizable (in the first case) or partially parallelizable (in the second case) algorithm would also be applicable to our analysis.

For these algorithms, we investigate when it is beneficial to use CPU and/or GPU acceleration taking advantage of thread and task parallelism yielded by key factors related to the algorithms, datasets, resources, and the distributed system employed.

4.2 Evaluated metrics

First, we consider metrics related to task execution according to its processing stages: task user code and (de-)serialization. These can be used to perform a head-to-head comparison between CPU-based and GPU-accelerated tasks leaving aside other overheads, such as storage I/O, network I/O, and task scheduling.

Task user code metrics. We report these metrics aggregated per task type; i.e., tasks running the same code are aggregated together.

- *Serial fraction execution time*: average time per task to process the serial fraction of the user code.
- *Parallel fraction execution time*: average time per task to process the parallel fraction of the user code.
- *CPU-GPU communication time*: average time per task to move data from CPU to GPU or vice versa.
- *User code execution time*: average time per task user code; i.e., summary of serial fraction, parallel fraction, and CPU-GPU communication times.

Data movement overheads. We report the data serialization and deserialization times grouped by CPU core for all task types.

- *Deserialization time*: average time per CPU core to read data from storage (e.g., disk) and load it to main memory.
- *Serialization time*: average time per CPU core to write data from main memory to storage.

Task level metrics. These metrics are computed per DAG level, i.e., for tasks that are in the same level in the DAG.

- *Parallel task execution time*: average time per algorithm iteration to run in parallel tasks placed in the same level in the DAG, considering all overheads related to data movement.

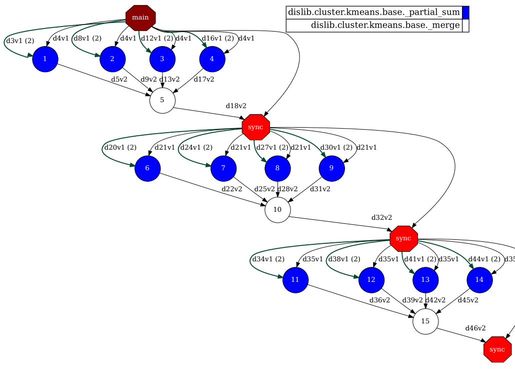
Figure 6 shows example DAGs generated by PyCOMPSs [68] for the two algorithms we consider: K-means and Matrix Multiplication (Matmul). For the task user code metrics we compute the average time per task of the same task type in the algorithm. Hence, for Matmul there are two tasks *matmul_func* (blue nodes in Figure 6b) and *add_func* (white nodes in Figure 6b). The data movement metrics are computed per CPU core involving all tasks (blue and white). And the parallel task execution time is computed per each level in the DAG (e.g., all blue nodes, the four white, the two white, etc.). This classification of metrics help us scrutinize thread and task parallelism in Section 5.

4.3 Evaluated factors

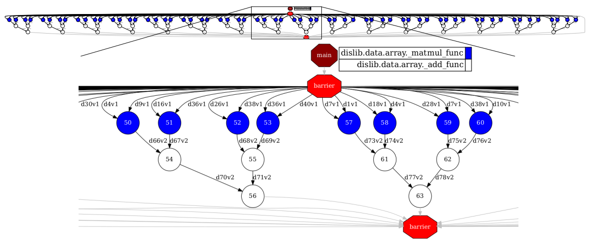
Each variable that affects measured performance (i.e., the outcome of an experiment) and has several alternatives is called a *factor* [28]. In our analysis, we measure the metrics detailed earlier by varying the factors explained in this section.

Table 1 presents a list of factors that affect the performance of task-based workflows. The list originates from experimentation, our experience, and our interviews with the team developed the distributed system COMPSs [4]. Some of these factors have been studied before (see Section 2), but never all together.

We classify the factors into four dimensions, namely task algorithm, dataset, resources, and system employed. Each factor further affects a number of parameters. For example, when we vary the block dimension, then parameters such as block size, grid dimension, and DAG shape are affected. We also consider an algorithm-specific parameter to study the computational complexity of tasks, e.g., the number of clusters in K-means. It is worth noting, that some presumably relevant parameters are determined from the factors and therefore, they are not directly included in our experiments; e.g., task dependencies may be extracted from the input algorithm, the block dimension determines the DAG shape, etc. Finally, we consider as future work other resource parameters, such as #GPU devices, RAM and GPU memory size, CPU-GPU bus throughput, and disk throughput. Our experiments indicate that the factors reported here are sufficient to detect relevant trends.


main
dislib.cluster.kmeans.base._partial_sum
dislib.cluster.kmeans.base._merge
sync
sync
sync

(a) DAG for K-means, grid dimension 4x1, 3 iterations



(b) DAG for Matmul, grid dimension 4x4

Figure 6: DAGs of partially & fully parallelizable algorithms

4.4 Experimental setup

4.4.1 Cluster configuration. We employed the Minotauro System, an HPC cluster hosted at BSC [13]. We used 8 out of the 38 nodes totally available. Each node has 16 CPU cores (Intel Xeon E5-2630 with 128 GB of RAM) and 4 GPU devices (NVIDIA K80, each with 12 GB of memory, using PCIe 3.0 for CPU-GPU interconnect). Hence, at most 128 tasks can be parallelized by CPUs and 32 by GPUs.

4.4.2 Distributed system. Our tested distribution system is PyCOMPSs (v.3.0) [68]. We experimented with two of its scheduling policies [12], one considering the task generation order and another one based on data locality. We also experimented with both local and shared disk storage running General Parallel File System (GPFS), which is commonly used in HPC environments.

4.4.3 Scripts. The software bits to run our analysis can be found in our public code repo[2]. We employ Python's performance counter [51] to measure serial fraction execution, parallel

[2]https://github.com/mnlcarv/Performance-Analysis-of-Distributed-GPU-Accelerated-Task-Based-Workflows.git

Table 1: Factors and parameters

Dimension	Factors	Parameters
Task algorithm	a) block dimension$^{*\|\dagger\ddagger\S}$, b) computational complexity$^{\|}$, c) parallel fraction$^{\|}$, and d) algorithm-specific parameter$^{\|}$	a) block size, grid dimension, and DAG shape b) - c) - d) -
Dataset	e) dataset dimension$^{*\|\dagger\ddagger\S}$	e) dataset size
Resources	f) processor type (i.e., CPU or GPU)$^{\|}$, and g) storage architecture†	f) maximum #CPU cores available depending on the processor type g) -
System	h) scheduling policy‡§	h) -

System functions affected: Device Speedup (‖); Storage I/O (†); Network I/O (‡); CPU-GPU Data Transfer (∗); Task Scheduling (§)

fraction execution (only for CPU-based tasks), CPU-GPU communication, and parallel task execution times. Since GPU executions run asynchronously with respect to CPU executions, we used CUDA events [21] to measure the parallel fraction execution for GPU-accelerated tasks. We used Paraver [48] to collect data deserialization and serialization times from traces automatically generated by PyCOMPSs runtime.

4.4.4 Algorithms. We used the K-means and Matmul implementations from dislib library (version 0.6.4) [17, 23], a distributed version of the scikit-learn library [47]. The algorithms used have the following GPU-accelerated tasks and computational complexities:

Matmul: *matmul_func*: $O(N^3)$ and *add_func*: $O(N)$, where N is the order of the block (these two tasks share a dependency as well).

K-means: *partial_sum*: $O(MNK^2)$, where M and N are the number of rows (i.e., samples) and columns (i.e., features) in a block, respectively, and K is the number of clusters.

The Matmul tasks *matmul_func* and *add_func* have fully parallel user code. The K-means task *partial_sum* has partially parallel user code. The two algorithms use different chunking strategies. Matmul chunks the datasets into rows and columns, while K-means chunks the dataset into rows. As result, they yield differently shaped DAGs as shown in Figure 6. The Matmul DAG is wide and shallow, which implies a high level of task parallelism. The K-means DAG is narrow and deep, resulting in a low degree of task parallelism and a high level of task dependencies. As discussed in Section 3.5, to control the number of tasks generated in all executions we assign exactly one block per grid per task. Matmul implementation guarantees such task granularity by default, but in K-means we enforce it by setting the number of grid columns to 1.

4.4.5 Datasets. We generated synthetic datasets in the form of NumPy array with random float64 (double precision) values. To ensure reproducibility across multiple executions, we used a fixed random state value. We varied the dataset sizes to ensure that each algorithm tested reaches the GPU memory limits. As we discussed, varying the block size leads to different grid dimensions and hence, different (but predicable) number of tasks and task granularities. Smaller blocks generate a large number of fine-grained tasks, whereas larger blocks generate a small number of coarse-grained tasks. And this allows us to stress the cluster resources for various scenarios of thread and task parallelism. Specifically, we used the following sizing scenarios:

- Matmul: two datasets: 8 GB, 32K x 32K (1024M elements) and 32 GB, 64K x 64K (4B elements), and five grid dimensions: 1x1, 2x2, 4x4, 8x8, 16x16.
- K-means: two datasets: 10 GB, 12.5M samples x 100 features (1250M elements) and 100 GB, 125M samples x 100 features (12.5B elements), and nine grid dimensions: 1x1, 2x1, 4x1, 8x1, 16x1, 32x1, 64x1, 128x1, 256x1.

5 EXPERIMENTS

Our analysis spans four main experiments: (a) an end-to-end performance analysis, (b) profiling task user code processing, (c) profiling parallel task processing, and (d) conducting a correlation analysis of all the factors considered.

Section 5.1 presents an end-to-end performance evaluation considering all system functions affected, i.e., CPU-GPU data transfer, limited device speedup, storage I/O, network I/O and scheduling overhead. Besides the costly CPU-GPU communication, task serial fraction may also cause a significant limitation in device speedup. As expected, data (de-)serialization dominates storage I/O and represents a critical bottleneck in distributed environments.

Next, we follow a bottom-up approach to analyze performance from thread-level to task-level parallelism. Section 5.2 discusses the gains obtained by GPUs due to thread parallelism by testing different task workloads with different computational complexities. For a focused evaluation, we study the overheads caused by CPU-GPU communication and serial fraction processing. Section 5.3 extends the previous analysis by considering storage I/O, network I/O, and scheduling overheads caused by task distribution. For this experiment, we compare the execution times between CPUs and GPUs over different combination of storage architectures and scheduling policies. Section 5.4 presents a correlation analysis of all relevant factors, summarizing our findings and discussing relevant trends identified in our study. Finally, a discussion about the generalizability of our approach is provided in Section 5.5.

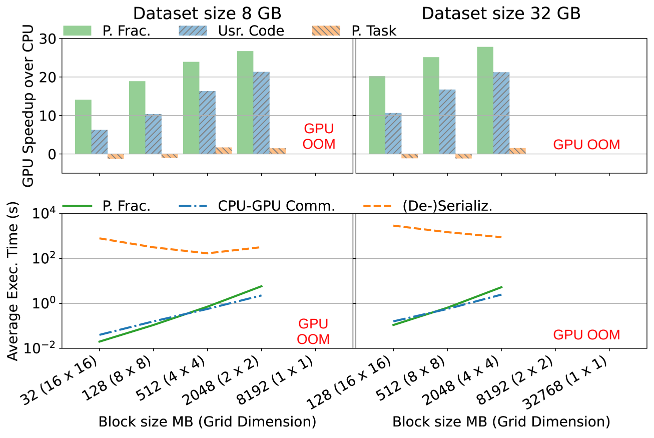
Unless otherwise stated, in the experiments we use the experimental setup presented in Section 4.4 with the 8 GB and 10 GB datasets for Matmul and K-means, respectively, shared disk as storage architecture, and task generation order as a scheduling policy. We ran each experiment six times and discarded the first run to avoid fluctuations due to warm up processing, such as loading required modules, compile the GPU kernel, etc.

5.1 End-to-end analysis

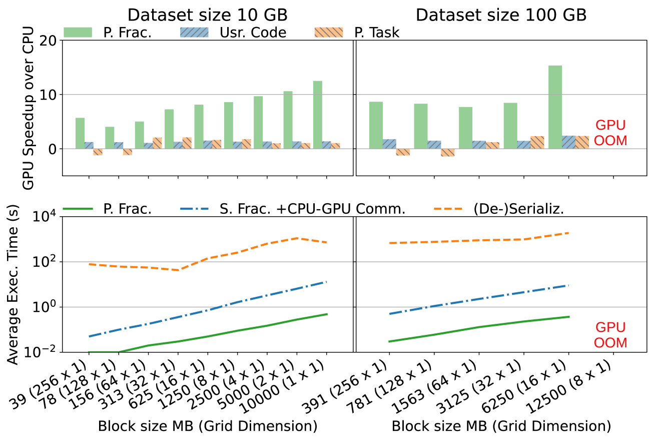
Figure 7 presents an end-to-end performance analysis for Matmul (Figure 7a) and K-means (Figure 7b). The analysis shows how factors, such as task computational complexity of each algorithm, block dimension (represented by block size and grid dimension in *X-axis*), and processor type (CPU or GPU) affect performance metrics, such as GPU speedup over CPU (top charts), execution times (bottom charts), parallel fraction (P. Frac - green line), CPU-GPU communication and serial fraction (blue line), and data serialization/deserialization (orange line). The left charts refer to the 8 GB and 10 GB datasets, and the right charts to the larger datasets 32 GB and 100 GB for Matmul and K-means, respectively.

(blue line), and data serialization/deserialitzation (red line).

Previous studies allude to CPU-GPU communication and (lack of) memory being limiting factors in GPU acceleration [60]. Our



(a) Matmul, 8GB left and 32GB right



(b) K-means, 10GB left and 100GB right

Figure 7: End-to-end performance analysis

experiments corroborate this hypothesis. Speedups obtained in the parallel fraction scale with the block size. However, Figure 7 shows that GPUs' memory limitations eventually lead to out-of-memory errors for large task granularities. Similarly, the user code speedup is affected by the block size. Consider for example Matmul on the 8 GB dataset. In fine-grained tasks, communication dominates the parallel fraction computation and the user code speedup decreases on average about 35% compared to the parallel fraction speedup. The decrease is smaller for coarse-grained tasks (e.g., 20% in block size 2048 MB), as computation dominates communication. The same trend holds for both algorithms in all datasets tested. This result is aligned with a typical performance optimization strategy that suggests increasing the volume of data to process for improving GPU throughput [32]. However, Figure 7 reveals that this strategy is not always effective when considering serial fraction processing and data (de-)serialization.

5.1.1 *Serial fraction processing.* Figure 7b shows that user code speedups do not change significantly with the block size. In partially parallelizable algorithms, these speedups suffer from serial processing and CPU-GPU communication that dominate the parallel fraction for all block sizes. Moreover, both serial and parallel fractions scale with the block size in similar proportions. Hence, the gain obtained by GPUs in the parallel fraction are

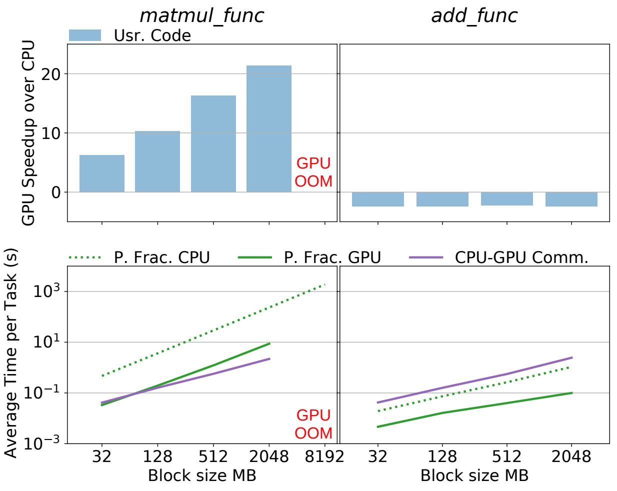


Figure 8: Task computational complexity in Matmul

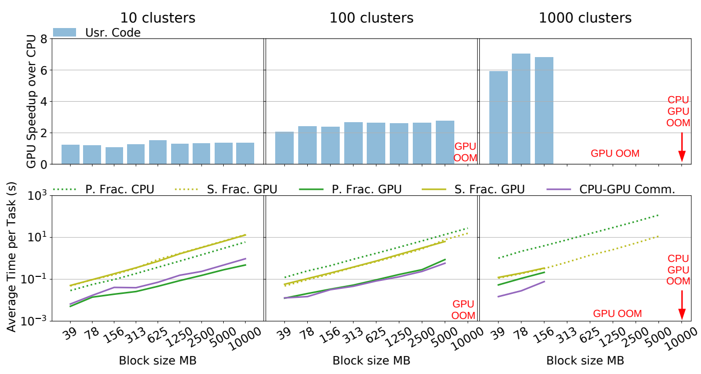
diminished by the cost of serial processing, resulting in marginal speedups for all block sizes.

5.1.2 *Data (de-)serialization.* Figure 7 shows that the speedups of parallel tasks are largely affected by data (de-)serialization. An excess of fine-grained tasks saturates the nodes with more tasks than available CPU cores and the disk with an abundance of read/write processes. On the other hand, a small number of coarse-grained tasks do not fully utilize the available CPU cores in the cluster and increase the data volume per read/write processes, thus, increasing the cost of (de-)serialization that cannot be parallelized. Figure 7 shows that the maximum GPU speedup is obtained when the maximum parallelism in data (de)-serialization is achieved; i.e., when the number of tasks is equal to the number of available CPU cores. It is also worth noting that for small block sizes the parallel task GPU speedup over CPU is negative due to the relatively considerable communication and data movement overheads. For larger block sizes, these overheads are amortized with the increased task parallelism and hence, GPU speedup turns positive as we reach the maximum task parallelism (32 tasks in our settings).

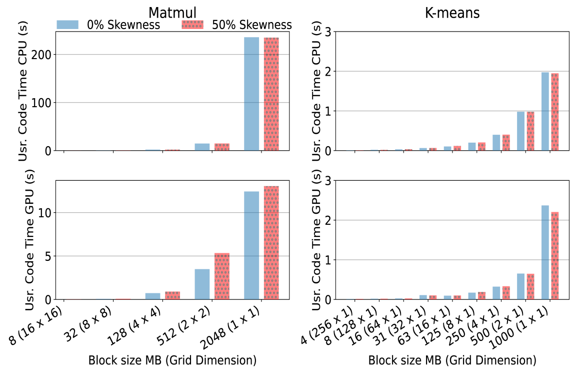
5.1.3 *Dataset size.* Figures 7 shows that the same trends hold for both smaller (left charts) and larger (right charts) datasets. In fact, as the dataset size increases, there is also an increase in GPU speedup (on avg ~5x) for parallel fraction and user code, whilst for parallel task the difference is negligible. Note that for the large datasets (32 GB for Matmul and 100 GB for K-means) our analysis is limited by the available GPU memory, which is 12 GB in our settings. We explain this in more detail in Section 5.3. This limitation does not allow testing block sizes larger than 2 MB (4x4 grid size) in Matmul and 6 MB (16x1 grid size) in K-means due to the GPU memory size needed to handle larger blocks along with the intermediate results produced in each algorithm.

Summary. Higher task granularity does not always achieve higher GPU speedups over CPU, mainly due to serial processing, CPU-GPU communication, and data (de-)serialization overheads. Our findings are as follows.

- **Observation O1**: User code speedups are not affected significantly by block size when parallel processing gains are diminished by the serial processing and CPU-GPU communication costs.



(a) Varying clusters, dataset size 10GB, K-means



(b) Varying data skew, Matmul 2GB, K-means 1GB, 10 clusters

Figure 9: The effect of (a) algorithm-specific parameter (#clusters) in K-means and (b) data skew in Matmul and K-means

- **Observation O2**: Parallel task speedups do not increase significantly for coarse-grained tasks, but can significantly improve when data (de-)serialization is fully parallelized using all available CPU cores.

5.2 Profiling task user code processing

In this experiment, we investigate how task algorithm factors affect the task user code metrics. In particular, we explore the effect of (a) the computational complexities of *matmul_func* and *add_func* tasks in Matmul, (b) the algorithm-specific parameter, which in this experiment is the number of clusters (#clusters) in K-means, and (c) data skew in Matmul and K-means.

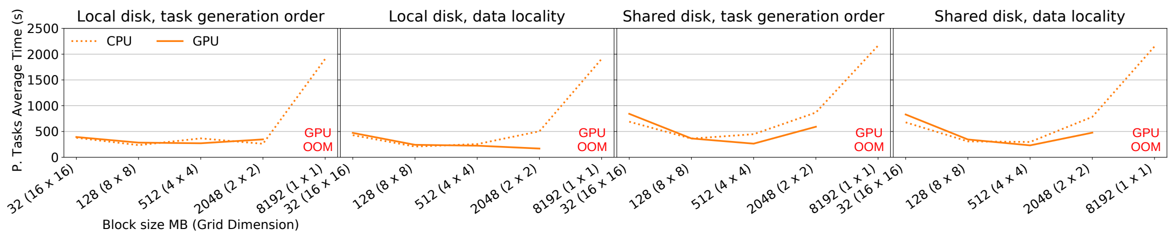
5.2.1 Computational complexity. We use Matmul that comprises two types of tasks with different computational complexities, namely *matmul_func* and *add_func*. Figure 8 shows how factors such as task computational complexity, block dimension (represented by block size in *X-axis*), and processor type affect performance metrics such as the user code GPU speedup over CPU (top charts), and execution times (bottom charts) related to the parallel fraction (green line) and CPU-GPU communication (purple line). Note that for the maximum task granularity (8192 MB) the matrix is multiplied in a single *matmul_func* task and no *add_func* task is needed (the reason we skip this value in Figure 8).

Figure 8 shows that the parallel fraction execution time in *matmul_func* dominates CPU-GPU communication times in most cases. Consequently, speedups scale with block size and increase as high as 21x. However, Figure 8 shows that this pattern does not repeat in *add_func* because communication dominates parallel fraction computation in all block sizes, resulting in performance degradation of GPUs compared to CPUs. This happens because the computational complexity of *add_func* is two orders of magnitude less than the computational complexity of *matmul_func*. Therefore, the parallel fraction in *add_func* is too small to benefit from the massive thread parallelism provided by GPUs.

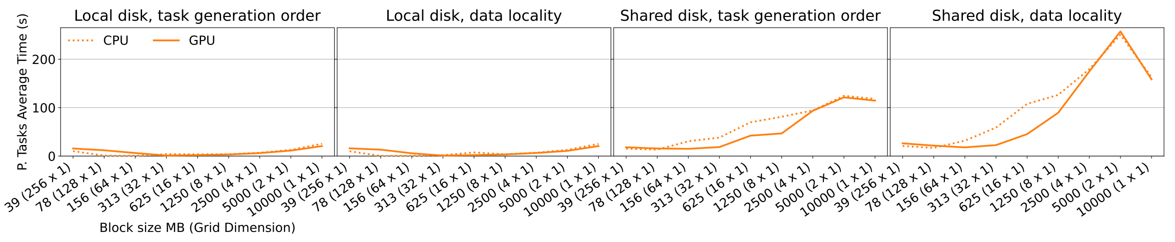
5.2.2 Algorithm-specific parameter. K-means has a single task, *partial_sum*, whose computational complexity is affected by an algorithm-specific parameter: #clusters. Figure 9a shows how factors such as task computational complexity with 10, 100, and 1000 clusters, respectively, block dimension (represented by block size in *X-axis*), parallel fraction and processor type affect performance metrics such as the user code speedup of GPU over CPU (top charts) and execution times (bottom charts) related to the parallel fraction (green line), serial fraction (yellow line), and CPU-GPU communication (purple line).

For 10 clusters, the computational complexity is so low that the parallel fraction execution time is less than the serial fraction and CPU-GPU communication times, resulting in marginal speedups (no more than 1.5x). For 100 clusters, the computational complexity is higher, making the parallel fraction execution time greater than the CPU-GPU communication time, but still less than the serial fraction execution time. In this case, the speedups are increased in about two times over the scenario with 10 clusters. Finally, for 1000 clusters, the parallel fraction keeps dominating the CPU-GPU communication, but the gap between the serial and parallel fraction execution times is reduced. Therefore, the speedups are up to 7x higher than in the scenario with 10 clusters. Note that the speedups keep scaling with #clusters until GPU's memory capacity is reached (OOM). Figure 9a shows that the speedups do not scale with the block size. This happens because the effect of #clusters dominates the effect of block dimension in the computational complexity of *partial_sum* task. And this is expected as #clusters has a quadratic impact in the computational complexity of *partial_sum*, while block dimension has a linear impact (see also Section 4.4.4).

5.2.3 Data Skew. Next, we investigate how data skew may affect our analysis. We generated two skewed datasets of size: 2 GB (16K x 16K, 256M elements) for Matmul, and 1 GB (1.25M samples x 100 features) for K-means. For doing so, we adapted the uniform distribution of the NumPy *random* routine [43] to move 50% of the elements to certain regions of the distribution forcing groups of elements in the dataset. Figure 9b compares the CPU (top charts) and GPU (bottom charts) task user code execution time in Matmul (left charts) and K-means (right charts) for the uniform (0% skew) and skewed datasets. We observe that data skew does not affect the task user code execution time. Our analysis also indicates similar performance for the fine-grained stages of parallel and serial fraction, CPU-GPU communication, and parallel task (the respective charts are omitted due to space considerations). This result is expected as the algorithms tested do not process differently tasks involving uniform or skewed data. In general, we believe that data skew might have an impact in certain pipelines that exploit data distribution in a special manner; how this could affect processing in GPUs is an excellent topic for future work.



(a) Matmul, dataset size 8GB



(b) K-means, dataset size 10GB, 10 clusters

Figure 10: The effects of storage architecture and scheduling policy on Matmul and K-means

Summary. The challenge at the task user code processing stage is to balance serial and parallel processing, and CPU-GPU communication to maximize GPU speedups. Our findings are as follows.

- **Observation O3**: In tasks with low computational complexity, increasing task granularity does not increase significantly GPU speedups over CPU.
- **Observation O4**: GPU speedups over CPU are largely affected by algorithm-specific parameters when the effect of these parameters dominates the effect of task computational complexity.

5.3 Profiling parallel task processing

So far, we focused on the task user code processing without considering the side-effects caused by data access and distribution, such as storage I/O, network I/O, and task scheduling overheads. Here, we expand the analysis to study task distribution on CPUs and GPUs across different combinations of storage architectures and schedulers. In this context, Figures 10a and 10b show how factors such as block dimension (represented by block size and grid dimension in *X-axis*), task computational complexity and parallel fraction of Matmul and K-means, processor type (CPU or GPU), storage architecture (local or shared disk), scheduling policy (data locality or task generation order) affect the parallel task execution time (*Y-axis*) performance metric.

Figures 10a and 10b show that changing the scheduling policy from task generation order to data locality does not result in significant changes in the execution times of CPUs and GPUs in local disk storage architecture. Since data access is simplified in local disk (i.e. data is accessed directly from the node's disks), task scheduling adds minimum overhead. And overall, execution on the local storage is faster than the shared storage (see Section 3.4).

However, the execution time in shared disk is more sensitive to changes in the scheduling policy than local disk, as depicted in Figures 10a and 10b. In this case, data access is more complex, as data is accessible from a shared file system across a shared network. As a result, changes in the scheduling policy are more evident in shared disk. This effect is more noticeable in K-means than in Matmul, as K-means tasks have lower computational complexity. Hence, K-means is affected more by scheduling overhead and the execution times gaps between CPUs and GPUs are more evident when changing the scheduling policy.

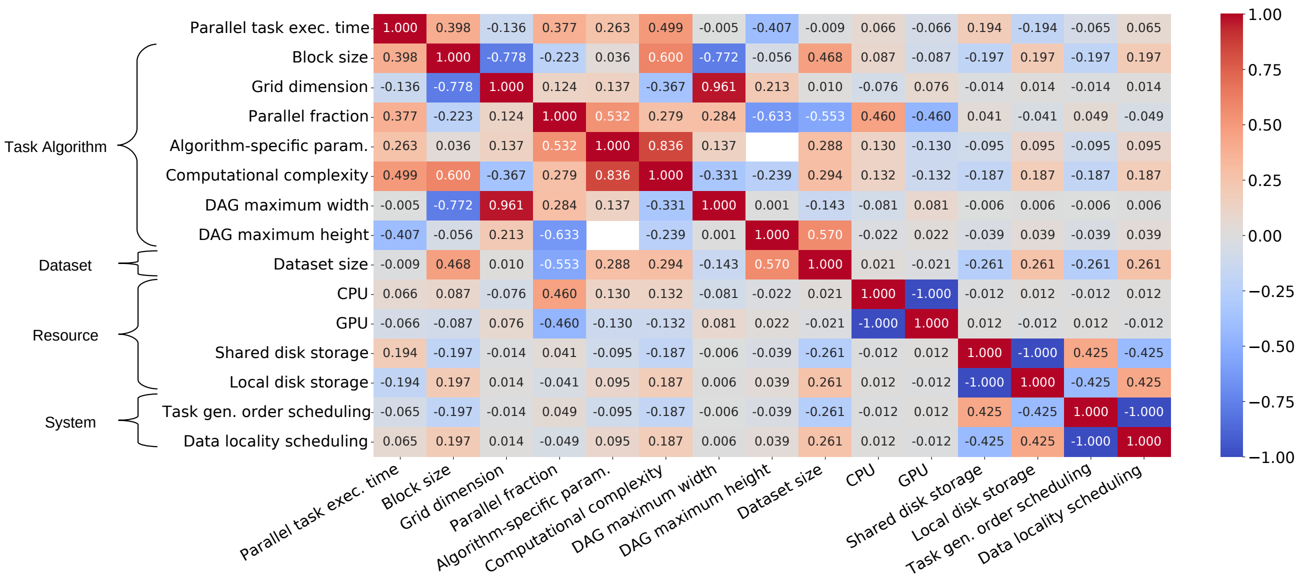
Two additional observations are worth noting in these charts. First, time increases for larger block sizes as task parallelism is not fully leveraged in coarse-grained tasks. However, it drops for the maximum block size, because in this case there is neither task distribution (only a single task is generated) nor any overhead caused by it. Second, Matmul requires memory equal to three times the block size (each task has two blocks inputs and one block output). Since the GPU device we use has 12 GB of memory, it runs out of memory for the maximum block size because then each task requires 24 GB (3 x 8192 MB) of memory.

Summary When tasks are distributed and processed in parallel, the execution time increases in both CPUs and GPUs due to the effects of storage I/O, network I/O, and scheduling overheads. The combination of different storage architectures and scheduling policies results in different execution performance gains between CPU and GPU. Our findings are as follows.

- **Observation O5**: When using local disks, variations in scheduling policy do not affect much the difference in execution time in CPUs and GPUs.
- **Observation O6**: When using shared disks, variations in scheduling policy affect differently the execution time in CPUs and GPUs for tasks with low computational complexity.

5.4 Correlation Analysis

In this section, we investigate how the parallel execution time metric and all the factors and parameters listed in Table 1 are correlated. Considering both factors and parameters as features, we have a mix of categorical and numerical data. The categorical features are processor type, storage architecture, and scheduling policy, and we use one-hot encoding to transform them into numerical data. We use the Spearman rank correlation [65] to measure the relationship between two features. We opted for this statistical measure, because of its robustness to potential non-linear relationships between the features. Naturally, other measures could be used as well.



	Parallel task exec. time	Block size	Grid dimension	Parallel fraction	Algorithm-specific param.	Computational complexity	DAG maximum width	DAG maximum height	Dataset size	CPU	GPU	Shared disk storage	Local disk storage	Task gen. order scheduling	Data locality scheduling
Parallel task exec. time	1.000	0.398	-0.136	0.377	0.263	0.499	-0.005	-0.407	-0.009	0.066	-0.066	0.194	-0.194	-0.065	0.065
Block size	0.398	1.000	-0.778	-0.223	0.036	0.600	-0.772	-0.056	0.468	0.087	-0.087	-0.197	0.197	-0.197	0.197
Grid dimension	-0.136	-0.778	1.000	0.124	0.137	-0.367	0.961	0.213	0.010	-0.076	0.076	-0.014	0.014	-0.014	0.014
Parallel fraction	0.377	-0.223	0.124	1.000	0.532	0.279	0.284	-0.633	-0.553	0.460	-0.460	0.041	-0.041	0.049	-0.049
Algorithm-specific param.	0.263	0.036	0.137	0.532	1.000	0.836	0.137		0.288	0.130	-0.130	-0.095	0.095	-0.095	0.095
Computational complexity	0.499	0.600	-0.367	0.279	0.836	1.000	-0.331	-0.239	0.294	0.132	-0.132	-0.187	0.187	-0.187	0.187
DAG maximum width	-0.005	-0.772	0.961	0.284	0.137	-0.331	1.000	0.001	-0.143	-0.081	0.081	-0.006	0.006	-0.006	0.006
DAG maximum height	-0.407	-0.056	0.213	-0.633		-0.239	0.001	1.000	0.570	-0.022	0.022	-0.039	0.039	-0.039	0.039
Dataset size	-0.009	0.468	0.010	-0.553	0.288	0.294	-0.143	0.570	1.000	0.021	-0.021	-0.261	0.261	-0.261	0.261
CPU	0.066	0.087	-0.076	0.460	0.130	0.132	-0.081	-0.022	0.021	1.000	-1.000	-0.012	0.012	-0.012	0.012
GPU	-0.066	-0.087	0.076	-0.460	-0.130	-0.132	0.081	0.022	-0.021	-1.000	1.000	0.012	-0.012	0.012	-0.012
Shared disk storage	0.194	-0.197	-0.014	0.041	-0.095	-0.187	-0.006	-0.039	-0.261	-0.012	0.012	1.000	-1.000	0.425	-0.425
Local disk storage	-0.194	0.197	0.014	-0.041	0.095	0.187	0.006	0.039	0.261	0.012	-0.012	-1.000	1.000	-0.425	0.425
Task gen. order scheduling	-0.065	-0.197	-0.014	0.049	-0.095	-0.187	-0.006	-0.039	-0.261	-0.012	0.012	0.425	-0.425	1.000	-1.000
Data locality scheduling	0.065	0.197	0.014	-0.049	0.095	0.187	0.006	0.039	0.261	0.012	-0.012	-0.425	0.425	-1.000	1.000

Figure 11: Correlation matrix of key features

To enrich the diversity of datasets sizes, we supplemented the experiments presented so far with additional experiments involving smaller datasets for each algorithm: 128 MB (4000 x 4000) for Matmul and 100 MB (125000 X 100) for K-means. For the new experiments, we follow the same analysis method presented in Section 4 using shared disk storage architecture and task generation order scheduling policy. Hence, in total the dataset used in the correlation analysis contains 192 samples, each having a unique combination of factors, parameters, and parallel task execution times.

Figure 11 shows a correlation matrix of the parallel task execution time and the features related to task algorithm, dataset, resource, and system employed. The values in the correlation matrix range from -1 to 1, where positive (negative) values mean that two features increase in the same (opposite) direction. The absolute value reveals the strength of the correlation, where 0 means no correlation and 1 means absolute correlation. Note that we do not compute correlation for the DAG maximum height and the algorithm-specific parameter, as in our experiments we always use the same values for these features.

5.4.1 *Verifying observations O1-O6.*

Figure 11 shows the following trends that reinforce the observations O1-O6.

O1. There is a positive correlation (about 0.4) between parallel task execution time and the task parallel fraction. As mentioned in O1, the task parallel fraction plays an important role in increasing the speedups of GPUs, being as important as block size, which also has a similar positive correlation value with the execution time.

O2. DAG maximum width has the weakest correlation with execution time (-0.005). This trend highlights the challenge to balance the levels of task and thread parallelism. As the DAG gets wider, the task-level parallelism increases and the thread-level parallelism decreases. O2 states that the non-linear behavior of data (de-)serialization is the main cause of this imbalance in parallelism.

O3. Task computational complexity is the feature with the strongest correlation with execution time (about 0.5), showing that the more complex the task is, the more time is required to process it. This trend is directly related to O3.

O4. The algorithm-specific parameter has significant positive correlation with task computational complexity (over 0.8) and execution time (about 0.3), reinforcing the observation O4. Additionally, we observe that algorithm-specific parameter has a correlation of about 0.5 with task parallel fraction. Hence, appropriately tuning the algorithm (application) is critical for parallelism and as a consequence for workflow performance.

O5 and O6. Storage architecture shows a correlation with the execution time: positive for shared disk (about 0.2) and negative for local disk (about -0.2). This is another indication that shared disk has higher latency than local disk. Changing the storage architecture results in different latency for data (de-)serialization. However, the correlation of scheduling policy with execution time is weaker (-0.065 for task generation order and 0.065 for data locality). Thus, changes in the scheduling policy do not result in significant variations in execution time. These trends are well aligned with observations O5 and O6.

5.4.2 *Additional findings.*

The correlation matrix presented in Figure 11 also reveals additional interesting trends.

(a) The block size has a much stronger correlation with execution time than dataset size. Thus, regardless of the dataset size, selecting an appropriate block size is a key step to process tasks efficiently.

(b) The high correlation of block size with grid dimension and DAG maximum width reinforces the trade-off between thread and task parallelism discussed exhaustively in this paper.

(c) Shared disk correlates positively with task generation order scheduler (over 0.4) and local disk correlates positively with data locality scheduler (over 0.4), revealing scheduling preferences depending on the storage architecture. This fits nicely the idea of the schedulers too; e.g., data locality for local disk.

(d) The negative correlation between processor type (i.e. CPU or GPU) and task parallel fraction shows that when GPUs are used, the parallel fraction is reduced because this fraction is executed considerably faster in GPUs than in CPUs.

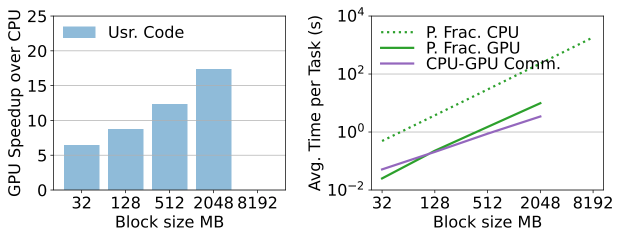


Figure 12: Analysis of task user code in Matmul FMA

(e) The weak correlation between processor type and execution time (0.066 for CPU and -0.066 for GPU) explains why the decision of choosing CPUs or GPUs to process tasks is non-obvious.

The additional insights provided by the correlation analysis confirm our hypothesis that to efficiently run GPU-accelerated task-based workflows, we should consider a combination of parameters related to task algorithm, dataset, resources, and system employed. However, selecting an appropriate combination of parameters that justifies the use of GPUs is a non-trivial task because of the complex non-linear relationships between these parameters. Overall, our experiments presented empirical evidence that naive heuristics and cost-based models do not suffice to tackle this problem.

5.4.3 Toward automated design. Our findings and observations could serve as guidelines toward designing an automated method to handle task-based workflows in modern, high-compute capacity, CPU-GPU engines. Tuning such a multiplicity of factors while taking into account our findings is a complex design problem, especially considering its non-linear nature. One potentially fruitful direction would be put learning models into play and investigate how they could identify and predict non-linear trends, as for example, the ideal block size to maximize the efficiency of each processor, the level of task computational complexity and parallel fraction that would make GPUs shine, etc.

5.5 Generalizability

Here, we discuss how our method could be generalized to additional algorithms, architectures, and processing systems.

5.5.1 Algorithms. Given the multitude of the factors considered in our analysis, working with two representative algorithms allows us to focus on the space of design options and identify the relevant factors and overheads involved. Extending the analysis to additional algorithms and implementations, which is part of our future work, would give us more data points between the two extreme cases considered here, namely fully and partially parallelizable algorithms, which presumably could help transition the focus of the work from "getting insights" to "propose actions", e.g., devise a method to decide when it is worth exploiting GPUs based on the ratio of parallel / serial code in an algorithm. Towards this end, we conducted an experiment with another implementation of Matmul, the Fused Multiply Add matrix multiplication (Matmul FMA) [18]. Figure 12 shows the GPU speedup and the average time per task employing the same parameters used in the experiment with the dislib implementation of Matmul (Figure 8). The results follow the same trends observed in Figure 8 with respect to user code speedup, parallel fraction, and CPU-GPU communication times.

5.5.2 Architectures. GPUs have a core execution model: several threads execute the same instructions with different data (i.e., Single Instruction, Multiple Thread – SIMT) [60]. CPU-GPU communication varies depending on the GPU integration architecture. Integrated GPUs share RAM with CPUs and therefore CPU-GPU communication is eliminated. Oppositely, the memory of dedicated GPUs is decoupled from RAM requiring CPU-GPU communication [60, 70]. Examples of hardware features available in modern GPUs to mitigate (but not eliminate) CPU-GPU communication include: faster bus interconnects (e.g. NVLink, CXL) [36, 60], shared memory between GPUs [24, 40], GPU direct memory access to disk [41, 52], and GPU cache [8, 10, 27, 37, 40, 41, 52, 58, 77]. As our work focuses on dedicated GPUs (see also Figure 2), considering a wider variety of GPU vendors and models would result into the same challenges (e.g., CPU-GPU communication overhead and GPU memory capacity), arguably, without changing our findings related to the key factors and parameters to consider. Hence, as our resources are limited, we chose to include in our analysis one of the most popular and well-adopted GPU architectures; i.e. dedicated NVIDIA GPUs using PCIe as bus interconnect [60]. For other GPU vendors, we could work similarly; for example, CuPy [50] supports AMD GPUs (ROCm) [1]. We believe that the trends we show are good indications of the practical overheads met in alternative architectures, at least at the task-level analysis.

5.5.3 Processing systems. In our analysis, we employed a very common distributed architecture used by data scientists, namely COMPSs, a highly scalable system that follows a typical master-worker architecture [19]. However, our process pipeline described in Section 3 is equivalent to how other popular distributed systems (e.g., Spark [67], Parla [31]) operate: work in batches, generate a DAG, assign tasks to multiple nodes, and then on each node deserialize data, execute the code in parallel or serial, serialize the data, etc. As a case in point, data (de-)serialization has been widely reported as a bottleneck in several systems including COMPSs [49] and Spark [5, 27, 38].

6 CONCLUSIONS AND FUTURE WORK

In our analysis, we investigated the wide range of performance gains obtained by GPUs in distributed task-based workflows running on HPC clusters. Understanding the key factors affecting the performance of task-based workflows is essential to understand not only how to maximize the gains of GPUs, but also when is worth using these accelerators. Our empirical analysis reveals that tuning task granularity alone is not sufficient to run task-based workflows efficiently. On the contrary, achieving reasonably good performance is a result of tuning a multiplicity of interrelated factors associated with task algorithm, dataset, resources, and system employed. Finally, we also discussed that our findings open up several interesting questions and directions for future research.

ACKNOWLEDGMENTS

This work has been partially supported by DEDS (H2020-MSCAITN-2020) with grant agreement No. 955895, the EU-HORIZON programme CREXDATA under GA.101092749, the EU-HORIZON programme FAIR-CORE4EOSC under GA.101057264, the EU-HORIZON programme EXTREMEXP under GA.101093164, the Spanish Government projects PID2019-107255GB and PID2020-117191RB-I00 / AEI/10.13039/501100011033 and MCIN/AEI /10.13039 /501100011033 (CEX2021-001148-S), and by the Departament de Recerca i Universitats de la Generalitat de Catalunya (2021 SGR 00412, MPiEDist).

REFERENCES

[1] AMD. 2023. *AMD ROCm Software.* https://www.amd.com/en/products/software/rocm.html

[2] Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, and Rosa M Badia. 2018. Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs. *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles* 73 (2018), 47.

[3] M Usman Ashraf, Fathy Alburaei Eassa, Aiiad Ahmad Albeshri, and Abdullah Algarni. 2018. Performance and power efficient massive parallel computational model for HPC heterogeneous exascale systems. *IEEE Access* 6 (2018), 23095–23107.

[4] Rosa M Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortes, and Raul Sirvent. 2015. Comp superscalar, an interoperable programming framework. *SoftwareX* 3 (2015), 32–36.

[5] Lorenzo Baldacci and Matteo Golfarelli. 2018. A cost model for SPARK SQL. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2018), 819–832.

[6] Tal Ben-Nun, Todd Gamblin, Daisy S Hollman, Hari Krishnan, and Chris J Newburn. 2020. Workflows are the new applications: Challenges in performance, portability, and productivity. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 57–69.

[7] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering*. 2091–2103.

[8] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*. 1891–1906.

[9] Ebubekir Buber and DIRI Banu. 2018. Performance analysis and CPU vs GPU comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*. IEEE, 1–6.

[10] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. Revisiting Query Performance in GPU Database Systems. *arXiv preprint arXiv:2302.00734* (2023).

[11] Juan M Cebrian, Baldomero Imbernón, Jesús Soto, and José M Cecilia. 2021. Evaluation of Clustering Algorithms on HPC Platforms. *Mathematics* 9, 17 (2021), 2156.

[12] Barcelona Supercomputing Center. 2023. *COMPSs Schedulers.* https://compss-doc.readthedocs.io/en/3.0/Sections/03_Execution_Environments/01_Scheduling.html

[13] Barcelona Supercomputing Center. 2023. *Minotauro System Overview.* https://bsc.es/supportkc/docs/Minotauro/overview/

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[15] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, and Charles Kamhoua. 2015. G-Storm: GPU-enabled high-throughput online data processing in Storm. In *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 307–312.

[16] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A Kwiat, Charles Alexandre Kamhoua, and Chonggang Wang. 2016. GPU-accelerated high-throughput online stream data processing. *IEEE Transactions on Big Data* 4, 2 (2016), 191–202.

[17] Javier Álvarez Cid-Fuentes, Salvi Solà, Pol Álvarez, Alfred Castro-Ginard, and Rosa M Badia. 2019. dislib: Large scale high performance machine learning in python. In *2019 15th International Conference on eScience (eScience)*. IEEE, 96–105.

[18] COMPSs. 2023. *Fused Multiply Add algorithm in COMPSs.* https://compss-doc.readthedocs.io/en/stable/Sections/07_Sample_Applications/02_Python/04_Matmul.html

[19] Javier Conejero, Sandra Corella, Rosa M Badia, and Jesus Labarta. 2018. Task-based programming in COMPSs to converge from HPC to big data. *The International Journal of High Performance Computing Applications* 32, 1 (2018), 45–60.

[20] CuPy. 2023. *NumPy/SciPy-compatible Array Library for GPU-accelerated Computing with Python.* https://cupy.dev/

[21] CuPy. 2023. *Performance Best Practices.* https://docs.cupy.dev/en/stable/user_guide/performance.html

[22] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

[23] Dislib. 2023. *Distributed computing library implemented over PyCOMPSs programming model for HPC.* https://github.com/bsc-wdc/dislib/tree/gpu-support

[24] Nitin A Gawande, Jeff A Daily, Charles Siegel, Nathan R Tallent, and Abhinav Vishnu. 2020. Scaling deep learning workloads: Nvidia dgx-1/pascal and intel knights landing. *Future Generation Computer Systems* 108 (2020), 1162–1172.

[25] Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux. 2021. Readys: A reinforcement learning based strategy for heterogeneous dynamic scheduling. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 70–81.

[26] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E Grant, and Ron Brightwell. 2017. sPIN: High-performance streaming Processing in the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16.

[27] Sumin Hong, Woohyuk Choi, and Won-Ki Jeong. 2017. GPU in-memory processing using spark for iterative computation. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 31–41.

[28] Raj Jain. 1990. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* john wiley & sons.

[29] Anusha Jayasimhan and P Pabitha. 2022. A comparison between CPU and GPU for image classification using Convolutional Neural Networks. In *2022 International Conference on Communication, Computing and Internet of Things (IC3IoT)*. IEEE, 1–4.

[30] Yasir Noman Khalid, Muhammad Aleem, Usman Ahmed, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. 2019. Troodon: A machine-learning based load-balancing application scheduler for CPU–GPU system. *J. Parallel and Distrib. Comput.* 132 (2019), 79–94.

[31] Hochan Lee, William Ruys, Ian Henriksen, Arthur Peters, Yineng Yan, Sean Stephens, Bozhi You, Henrique Fingler, Martin Burtscher, Milos Gligoric, et al. 2022. Parla: A python orchestration system for heterogeneous architectures. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[32] Suyeon Lee and Sungyong Park. 2021. Performance analysis of big data ETL process over CPU-GPU heterogeneous architectures. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 42–47.

[33] Jiesong Liu, Feng Zhang, Hourun Li, Dalin Wang, Weitao Wan, Xiaokun Fang, Jidong Zhai, and Xiaoyong Du. 2022. Exploring Query Processing on CPU-GPU Integrated Edge Device. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4057–4070.

[34] Fengshun Lu, Junqiang Song, Fukang Yin, and Xiaoqian Zhu. 2012. Performance evaluation of hybrid programming patterns for large CPU/GPU heterogeneous clusters. *Computer physics communications* 183, 6 (2012), 1172–1181.

[35] Hua Luan and Yan Fu. 2021. Accelerating group-by and aggregation on heterogeneous CPU-GPU platforms. In *The International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery*. Springer, 980–990.

[36] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1633–1649.

[37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton join: efficiently scaling to a large join state on GPUs with fast interconnects. In *Proceedings of the 2022 International Conference on Management of Data*. 1017–1032.

[38] Dieudonne Manzi and David Tompkins. 2016. Exploring GPU acceleration of apache spark. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 222–223.

[39] Marcelo K Moori, Hiago Mayk G de A Rocha, Matheus A Silva, Janaina Schwarzrock, Arthur F Lorenzon, and Antonio Carlos S Beck. 2023. Automatic CPU-GPU Allocation for Graph Execution. In *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 27–34.

[40] Shin Morishima and Hiroki Matsutani. 2017. Distributed in-GPU data cache for document-oriented data store via PCIe over 10 Gbit ethernet. In *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers 22*. Springer, 41–55.

[41] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU-acceleration for Memory-Efficient Analytics. *CIDR. www. cidrdb. org* (2023).

[42] S Nishanth, Manu S Rao, BM Sagar, T Padmashree, and NK Cauvery. 2022. Performance of CPUs and GPUs on Deep Learning Models For Heterogeneous Datasets. In *2022 6th International Conference on Electronics, Communication and Aerospace Technology*. IEEE, 978–985.

[43] NumPy. 2023. *NumPy Random routine.* https://numpy.org/doc/stable/reference/random/generated/numpy.random.random.html

[44] NVIDIA. 2023. *CUDA Toolkit.* https://developer.nvidia.com/cuda-toolkit

[45] NVIDIA. 2023. *GPU Performance Background User's Guide.* https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html

[46] OpenMP. 2023. *The OpenMP API specification for parallel programming.* https://www.openmp.org

[47] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[48] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, Vol. 44. 17–31.

[49] Lucas M Ponce, Daniele Lezzi, Rosa M Badia, and Dorgival Guedes. 2019. Extension of a Task-based model to Functional programming. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 64–71.

[50] PyPi.org. 2023. *CuPy ROCm.* https://pypi.org/project/cupy-rocm-5-0

[51] Python. 2023. *time — Time access and conversions.* https://docs.python.org/3/library/time.html

[52] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, CJ Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-Initiated On-Demand High-Throughput Storage Access in the BaM System Architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[53] Ashur Rafiev, Mohammed AN Al-Hayanni, Fei Xia, Rishad Shafik, Alexander Romanovsky, and Alex Yakovlev. 2018. Speedup and power scaling models for heterogeneous many-core systems. *IEEE Transactions on Multi-scale computing systems* 4, 3 (2018), 436–449.

[54] Guillem Ramirez-Gargallo, Marta Garcia-Gasulla, and Filippo Mantovani. 2019. TensorFlow on state-of-the-art HPC clusters: a machine learning use case. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 526–533.

[55] Cristian Ramon-Cortes, Pol Alvarez, Francesc Lordan, Javier Alvarez, Jorge Ejarque, and Rosa M Badia. 2021. A survey on the Distributed Computing stack. *Computer Science Review* 42 (2021), 100422.

[56] RAPIDS. 2023. *GPU Accelerated Data Science*. https://rapids.ai/

[57] M Mazhar Rathore, Hojae Son, Awais Ahmad, Anand Paul, and Gwanggil Jeon. 2018. Real-time big data stream processing using GPU with spark over hadoop ecosystem. *International Journal of Parallel Programming* 46 (2018), 630–646.

[58] Syed Mohammad Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[59] Christoph Riesinger, Arash Bakhtiari, Martin Schreiber, Philipp Neumann, and Hans-Joachim Bungartz. 2017. A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters. *Computation* 5, 4 (2017), 48.

[60] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–38.

[61] Maximilian E Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2022. Recursive SQL and GPU-support for in-database machine learning. *Distributed and Parallel Databases* 40, 2-3 (2022), 205–259.

[62] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[63] Alkis Simitsis, Spiros Skiadopoulos, and Panos Vassiliadis. 2023. The History, Present, and Future of ETL Technology. In *Proceedings of the 25th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP)*. 3–12.

[64] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 698–709.

[65] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).

[66] Young-Kyoon Suh, Junyoung An, Byungchul Tak, and Gap-Joo Na. 2022. A Comprehensive Empirical Study of Query Performance Across GPU DBMSes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–29.

[67] Shanjiang Tang, Bingsheng He, Ce Yu, Yusen Li, and Kun Li. 2020. A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 71–91.

[68] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

[69] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2018. Morpheus: Exploring the potential of near-data processing for creating application objects in heterogeneous computing. *ACM SIGOPS Operating Systems Review* 52, 1 (2018), 71–83.

[70] Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos Kotselidis. 2022. Enabling Transparent Acceleration of Big Data Frameworks Using Heterogeneous Hardware. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3869–3882.

[71] Bobbi W Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2491–2503.

[72] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 273–283.

[73] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freire. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.

[74] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[75] Feng Zhang, Chenyang Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2021. Fine-grained multi-query stream processing on integrated architectures. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2303–2320.

[76] Tao Zhang, Jingjie Zhang, Wei Shu, Min-You Wu, and Xiaoyao Liang. 2015. Efficient graph computation on hybrid CPU and GPU systems. *The Journal of Supercomputing* 71 (2015), 1563–1586.

[77] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with the GPU. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[78] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 88–100.