RTLB_Sched: Real Time Load Balancing Scheduler for CPU-GPU Heterogeneous Systems

Taha Abdelaziz Rahmani
dept. computer science, LIO laboratory
Ahmed Ben Bella Oran 1 University
Oran, Algeria
rahmani.taha@edu.univ-oran1.dz

Ghalem Belalem
dept. computer science, LIO laboratory
Ahmed Ben Bella Oran 1 University
Oran, Algeria
ghalem1dz@gmail.com

Sidi Ahmed Mahmoudi
dept. computer science, faculty of engineering
University of Mons
Mons, Belgium
Sidi.mahmoudi@umons.ac.be

***Abstract*— Heterogeneous computing systems consisting of Central Processing Units CPUs and Graphical Processing Units GPUs are found everywhere, from mobile phones, laptops to cloud clusters, due to their low cost/performance ratio and their impressive peak computational performance.**

One of the main and most complex problems in these systems is the load imbalance. This imbalance is caused by the large computing power difference between its computing nodes.

We present RTLB_Sched, a Real Time Load Balancing Scheduler that minimizes the load difference between the CPU and GPU nodes. We use execution time prediction to define a new metric called load difference. We describe the method and provide mathematical formulas to calculate this metric.

Experiments of scheduling multiple OpenCL applications on a heterogeneous system show that RTLB_Sched outperforms the Device Suitability and Round Robin approaches in load balancing.

Keywords—Heterogeneous Systems, Machine Learning, Scheduling, Load Balancing, Pycaret.

I. INTRODUCTION

CPU/GPU heterogeneous systems are systems that incorporate both CPUs and GPUs. Several frameworks like OpenCL were developed to allow software programmers to exploit the characteristics of these systems to the most.

OpenCL is a standard for parallel cross-platform programming of modern processors like GPUs and CPUs [1].

In OpenCL, the application is divided into two parts, the serial part called “host”, it is executed on the CPU and the parallel part called “kernel”, it can be executed on any supported device (CPU, GPU, ...).

Most applications developed for heterogeneous systems are more suitable to the GPU than to the CPU resulting in overloading the GPU and under-utilizing the CPU. This causes longer execution time, waste of computing power and more energy consumption.

Balancing the load of heterogeneous systems is one of the most complex problems in this field. The scheduling of the applications needs to be done according to each device computing power. The computing power may largely change form device to another.

Several works have tackled this problem with different approaches. Works like [2] resorted to machine learning to balance the load of an application pool. However, this solution cannot function in real time where applications are continuously submitted at different times.

The execution times of an application on the different computing devices of the system is important in determining the devices’ loads. When a device is overloaded, the applications are mapped to a slower device to balance the load of the system.

In this paper we present RTLB_Sched, a real time scheduler that minimizes the imbalance of the system using CPU and GPU predicted execution times.

The rest of this paper is organized as follows: in Section 2, we present previous related works. In Sections 3 and 4 we describe the whole process to build a machine learning execution time predictor. In Section 5 we describe the architecture of RTLB_Sched. In Section 6 we present the results of the experimentation conducted on RTLB_Sched and a discussion of the results. Finally, we conclude this work and present possible ameliorations of the solution in Section 7.

II. RELATED WORKS

CPU/GPU heterogeneous systems have gained wide popularity among researchers and developers of different domains:

A. Digital image processing

The implementation of the applications on GPU speeds up the calculations. However, these accelerations become less significant in multiple image processing or if the amount of work required is very low [3]. Authors in [3] propose two solutions to efficiently assign the applications of multiple image and individual image processing.

To distribute multiple image processing tasks between CPU or GPU, the authors use both “static scheduling” and “dynamic scheduling”. “Dynamic scheduling” assigns tasks to the most suitable resources according to the estimation of the computation duration [3].

As for individual image processing, when the complexity of the task is high enough and requires lot of parallel computational power, it is executed to the GPU, otherwise, it is executed on the CPU [3]. The authors proposed a method to calculate a complexity factor of image processing algorithms and a threshold. If the complexity of a task is smaller than the threshold, the task is better executed on the CPU, otherwise, the application is executed on the GPU.

B. Mobile computing

In [4] the authors propose a heterogeneous implementation of the PCA method using the NIPALS algorithm.

The authors proposed to split the instructions of the algorithm to two parts. The part that requires high precision, will be executed on the CPU and computation intensive part will be accelerated by the GPU [4].

C. Cloud computing

In a Cloud environment, clients submit their jobs to the cloud to be executed. The Cloud service provider tries to execute users' jobs using a minimum number of resources in the minimum time possible.

In this context, the authors of [5] proposed a machine learning based device suitability scheduler "KubeSCRTP" in a Kubernetes infrastructure. "KubeSCRTP" assigns Docker containerized applications to CPU or GPU. Using a trained machine learning model (Decision Tree), the scheduler classifies the applications to "fast execution" or "slow execution". The applications labeled as "fast execution" are executed on the CPU while those classified as "slow execution" are executed on the GPU.

Other works like [6] proposed E-OSched, a load balancing scheduler. E-OSched executes the highly intensive applications on the GPU and the less intensive ones on the CPU. The computational requirements of all the applications are determined before starting the scheduling. The job pool is sorted in an ascending order (shortest sized jobs first) of the processing requirement. After that the scheduler starts by assigning the applications with high processing requirement in the top of the pool to the GPU, while the applications with low processing requirement in the bottom of the pool are assigned to the CPU.

Troodon [2] is an amelioration of the E-OSched [6], it takes into consideration the device suitability and relative speedup of the applications.

Troodon classifies the applications into CPU-suitable pool and GPU-suitable pool. It sorts the CPU suited applications in descending order and the GPU suited applications in ascending order of predicted speedup. Troodon then combines the two pools. Applications at the top of the pool are mapped to the CPU while those at the bottom are mapped to the GPU until each device reaches its estimated load computed beforehand. when all GPUs in the system have reached their estimated load, the remaining GPU-suitable applications are then assigned to the CPUs and vice versa [2].

In summary, the presented works in the cloud computing domain did not provide a generic solution to the load balancing problem. In [5] the authors did not take into consideration the load balance of the system, whereas in [6] and [2] both schedulers require knowing all the applications that will be submitted and their processing requirements before starting the scheduling. They don't respect the order of execution of the applications. In real time cloud systems, users continuously submit their applications any time. It is impossible to know all the applications before they are even submitted to the system. In this case, their solutions are infeasible.

III. Dataset preparation

To build CPU and GPU execution time predictors we use a supervised machine learning approach based on three steps: dataset preparation, prediction models' training, prediction models' validation and deployment.

We create two different datasets one for the CPU and the other for the GPU. We used 10 linear algebra OpenCL applications from Polybench [7] benchmark and an own developed application to create the datasets. The applications are listed in *Table 1*.

Table 1: Benchmark OpenCL applications[1]

Application	Benchmark	Description
2mm	Polybench	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	Polybench	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
atax	Polybench	Matrix Transpose and Vector Multiplication
bicg	Polybench	BiCG Sub Kernel of BiCGStab Linear Solver
doitgen	Polybench	Multiresolution analysis kernel (MADNESS)
gemm	Polybench	Matrix-multiply C=alpha.A.B+beta.C
gemver	Polybench	Vector Multiplication and Matrix Addition
mvt	Polybench	Matrix Vector Product and Transpose
syr2k	Polybench	Symmetric rank-2k operations
syrk	Polybench	Symmetric rank-k operations
transpose	Own developed	Matrix transpose

Each dataset consists of a 23 OpenCL kernel code features used by [2] and the execution time as a target. The features are listed in *Table 2*.

We executed each application 2000 times in both CPU and GPU, in each execution we change the input data size of the application and we record its CPU execution time in the CPU dataset and the GPU execution time in the GPU dataset.

[1] http://web.cs.ucla.edu/~pouchet/software/polybench/

Table 2 OpenCL's code features [2]

No.	Features
1	Data Size
2	Number of Return Statements
3	Number of Control Statements
4	Number of Allocation Instructions
5	Number of Load Instructions
6	Number of Store Instructions
7	Number of Multiplication (Float Datatype) Operations
8	Number of Multiplication (Integer Datatype) Operations
9	Number of Division (Float Datatype) Operations
10	Number of Division (Integer Datatype) Operations
11	Number of Condition Check Instructions
12	Number of Addition (Float Datatype) Operations
13	Number of Addition (Integer Datatype) Operations
14	Number of Subtraction (Float Datatype) Operations
15	Number of Subtraction (Integer Datatype) Operations
16	Number of Function Call Instructions
17	Number of Functions
18	Number of Blocks
19	Number of Instructions
20	Number of Float Operations
21	Number of Integer Operations
22	Number of Loop Operations
23	Number of Loops

A. *Feature extraction*

We use an LLVM (Low Level Virtual Machine) pass to extract 20 features from LLVM intermediate representation and regular expressions to extract the rest. The steps of the code features extraction are depicted in *Figure 1*.

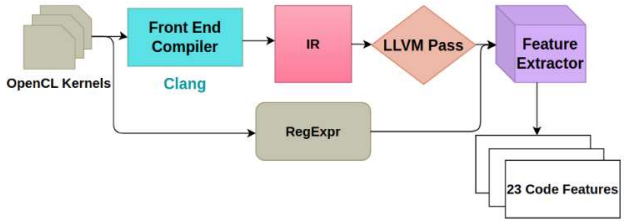

Figure 1. Code features extraction

We first compile each kernel with a front-end compiler (Clang) to verify its correctness. The kernel then is transformed to the LLVM intermediate representation (IR). The feature extractor uses Regular Expressions and an LLVM pass to obtain the code features of each kernel.

IV. Execution Time Prediction Models' Training

To train, validate and deploy CPU and GPU execution time prediction models, we use PyCaret library [8].

A. *PyCaret[8]*

PyCaret is an open-source, low-code machine learning library in Python that automates machine learning workflows. It can be used to replace hundreds of lines of code with few words only [8].

PyCaret's Regression module (pycaret.regression) is a supervised machine learning module that is used for predicting continuous values/outcomes using various techniques and algorithms [2].

PyCaret's workflow can be described as in *Figure 2*:

Figure 2. PyCaret's workflow

Each step can be realized by simple function calls (one or two calls)[3].

B. *Training of execution time models*

After creating the CPU and GPU datasets, we divide each one into two subsets: the train test set which represents 90% of the data and the validation set (10%). For each model (CPU/GPU), we first setup the environment by providing the setup function with the train test dataset and the target variable (execution time). We set the feature selection and multicollinearity removing parameters to "True",

We compare the machine learning models provided by PyCaret. *Figure 3 and Figure 4* represent the results of training and testing various models for CPU and GPU execution time prediction

The best models obtained for CPU and GPU execution time predictions are "Extra trees Regressor" for the CPU and "Light Gradient Boosting Machine" for the GPU.

Model	MAE	MSE	RMSE	R2	RMSLE	MAPE	TT (Sec
Extra Trees Regressor	0.0557	0.0608	0.2424	0.9241	0.0980	0.3698	1.5550
Random Forest Regressor	0.0609	0.0613	0.2440	0.9239	0.1031	0.4111	1.4280
Gradient Boosting Regressor	0.0960	0.0676	0.2556	0.9155	0.1098	2.2630	0.5800
Decision Tree Regressor	0.0585	0.0702	0.2613	0.9145	0.1055	0.3724	0.0270
Light Gradient Boosting Machine	0.0911	0.0699	0.2599	0.9128	0.1099	1.0211	0.1080
AdaBoost Regressor	0.2126	0.2021	0.4474	0.7472	0.1924	8.1432	0.1020

Figure 3.CPU execution time prediction model training + test

Model	MAE	MSE	RMSE	R2	RMSLE	MAPE	TT (Sec)
Light Gradient Boosting Machine	0.0956	0.0685	0.2561	0.9463	0.1244	1.1467	0.1410
Gradient Boosting Regressor	0.1123	0.0742	0.2670	0.9419	0.1267	2.4921	0.6360
Extra Trees Regressor	0.0658	0.0757	0.2710	0.9410	0.1186	0.4343	1.6200
Random Forest Regressor	0.0735	0.0765	0.2727	0.9399	0.1269	0.5214	1.5100
Decision Tree Regressor	0.0694	0.0882	0.2936	0.9308	0.1364	0.4506	0.0300
AdaBoost Regressor	0.3234	0.2993	0.5436	0.7641	0.2836	14.9207	0.1100

Figure 4.GPU execution time prediction model training + test

C. *Validation and depolyment*

We finalize the models by retraining them on their whole datasets including the test sets. Then we validate them using the unseen data of the validation set. Finally, we save them for later use.

Figure 5 and *Figure 6* present the results of the validation of the finalized CPU and GPU time prediction model.

	Model	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	Extra Trees Regressor	0.0483	0.0295	0.1719	0.9679	0.0886	0.3391

Figure 5.CPU execution time prediction model validation

[2] https://towardsdatascience.com/introduction-to-regression-in-python-with-pycaret d6150b540fc4

[3] https://pycaret.gitbook.io/docs/get-started/functions

	Model	MAE	MSE	RMSE	R2	RMSLE	MAPE
0	Light Gradient Boosting Machine	0.1045	0.0696	0.2638	0.9465	0.1306	1.1540

Figure 6.GPU execution time prediction model validation

V. SYSTEM ARCHITECTURE

Figure 7 presents the global architecture RTLB_Sched

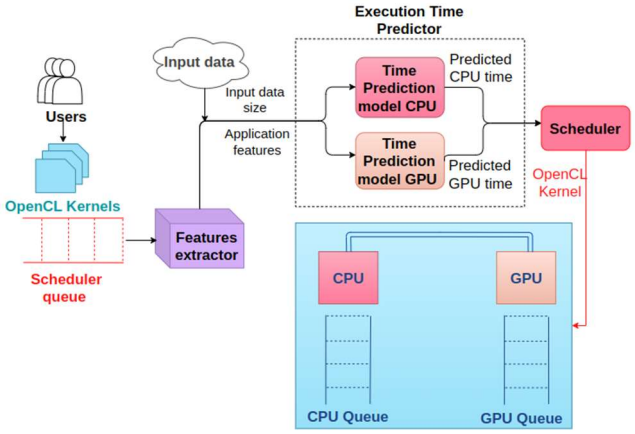

Figure 7.RTLB_Sched architecture

Users submit their applications in the scheduler queue. The code features of the application at the top of the queue will be extracted by the "Features extractor" module. The code features along with the input data size of the application will be used by the "Execution time predictor". The "Execution time predictor" uses the CPU and GPU models to predict the execution time of the applications.

With the predicted execution times and considering the state of the system, the scheduler assigns the application the most appropriate device.

VI. RTLB_SCHED

The objective of RTLB_Sched, is to assign the submitted applications one by one in the order of submission to the devices in a way to reduce the different of the loads between the CPU and the GPU.

We define the load of a device by the time needed to execute all the application on the device.

Figure 8 and *Figure 9* represent two heterogeneous systems consisting of a CPU and a GPU, each computing device (CPU or GPU) has its own queue of applications.

The system in *Figure 8* is perfectly balanced. All the load of its nodes are equal. The system in *Figure 9* is an unbalanced system.

To choose the most appropriate device to execute an application on. RTLB_Sched calculates the load differences between the CPU and the GPU in the two possible scenarios:

- The application will be mapped to the CPU.
- The application will be mapped to the GPU.

The scheduler then chooses the device that provide the minimum load difference.

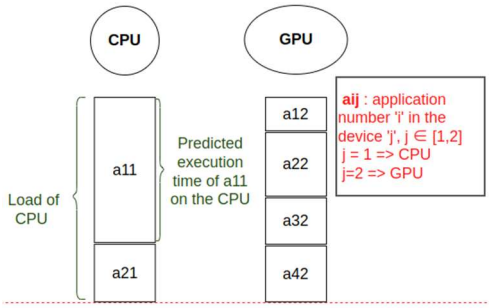

Figure 8.Balanced System

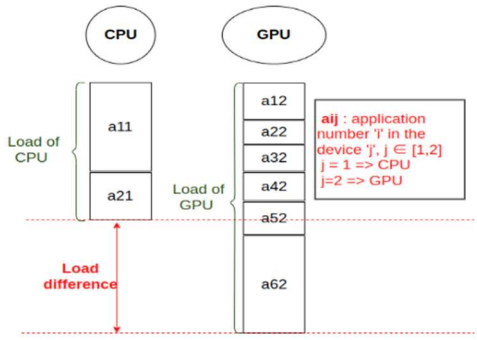

Figure 9. Imbalanced System

A. Mathematical model

In *Table 3* we list the terminologies used in describing the mathematical model:

Table 3 Notations

Notation	Description
A	List of OpenCL kernels submitted by users.
S	Heterogeneous system CPU/GPU.
SQ	Scheduler's queue.
$time_{ij}$	Predicted execution time of the application number 'i' in the queue of the node j (1 => CPU, 2 => GPU).
F_i	List of OpenCL kernel code features of an application 'i'.
Q_j	Load of node j (queue time).
$\|Q_j\|$	The number of applications in the queue of the node 'j'.
diff	Difference between CPU and GPU loads (queue times).
$difference_j$	Predicted load difference if an application is mapped to the device 'j' (j = 1 => CPU, j= 2 => GPU).

- The heterogeneous system 'S' consists of a CPU and a GPU.
- 'A' = $\{a_1, a_2, \ldots, a_n\}$ is the list of OpenCL applications submitted by users, these applications are placed in the scheduler's Queue 'SQ'.

- The Kernel code of the application 'a_1'in the head of the scheduler's queue 'SQ' is provided as input to the "Features extractor" to extract its code features 'F_1'.
- "Execution Time Predictor» uses the 'F_1' to predict GPU '$time_{i1}$' and '$time_{k2}$', the execution times of 'a_1' in the CPU and GPU respectively. ('i' is the order of a_1 if it will be mapped it the CPU and 'k' is the order of a_1 if it will be mapped to the GPU)
- The scheduler then calculates the load difference between the CPU and the GPU in two different scenarios: the first when the 'a_1' is mapped to the CPU: '$difference_1$' and the second when 'a_1' is mapped to the GPU: '$difference_2$'.

We define 'diff' as the subtraction between the loads of the CPU 'Q_1' and the GPU 'Q_2'. The load of a node 'j' is the sum of predicted execution times '$time_{ij}$' of all the applications placed in the node 'j' and defined as follows:

$$Q_j = \sum_{i=1}^{|Q_j|} time_{ij} \quad (1)$$

When a system 'S' is perfectly balanced as in *Figure 8*:

$$Q_1 = Q_2 \Leftrightarrow Q_1 - Q_2 = 0 \quad (2)$$

When 'S' is unbalanced, there is two possibilities, either:

$$Q_1 < Q_2 \Leftrightarrow Q_1 - Q_2 < 0 \quad (3)$$

or:

$$Q_1 > Q_2 \Leftrightarrow Q_1 - Q_2 > 0 \quad (4)$$

To generalize:

$$diff = |Q_1 - Q_2| > 0 \quad (5)$$

When *Equation 5* is True, the system is not perfectly balanced. The closer 'diff' is to '0' the more balanced the system. '$difference_1$' and '$difference_2$' are calculated as follows:

$$difference_1 = |Q_1 + time_{i1} - Q_2| \quad (6)$$

$$difference_2 = |Q_1 - (Q_2 + time_{i2})| \quad (7)$$

- After calculating '$difference_1$' and '$difference_2$', the scheduler selects the minimum of the differences and assigns the application to the corresponding device.

VII. Experimentation and results

To evaluate RTLB_Sched we developed it on a CPU/GPU heterogeneous system and evaluated it using multiple OpenCL applications with multiple data sizes.

A. Experimental setup

- *System*

The system consists of a host with Linux Ubuntu 20.04.1 LTS operating system with a multi-core CPU and a GPU.

CPU: Intel® Core™ i5-6300U with 2 cores and 4 threads with base frequency of 2.30 GHZ and maximum turbo frequency of 2.80 GHZ and 3 MB Intel® Smart Cache.

GPU: Intel HD Graphics 520 graphics card, it has 24 execution units, with 192 shading units, with maximum frequency of 1.1 GHz.

- *Workload*

11 benchmark applications listed in *Table 1* from "PolyBench" [7] benchmark suit are used to run the experimentation. The applications are executed using different input data sizes. All the experiments were repeated 10 times and mean values were reported.

- *Baselines*

To compare RTLB_Sched approach, we implemented 2 different approaches:

RR: (Round Robin) The applications are alternatively assigned to the computing nodes, the first application to the CPU, the second to the GPU and so on.

DS: (Device Suitability) The applications are assigned to the suitable device. The suitable device is the device that executes the application in a shorter time. This approach was used by [5].

- *Metrics*

To evaluate the three methods, we used the proposed metric in *Equation 5*. The metric "diff" represents the difference between the CPU and the GPU loads in a single moment of the experimentation.

To evaluate the three approaches during the whole experimentation, we calculated the total load difference. The total load difference is the sum of the differences in different moments of the experimentation, the smaller the sum the better the performance of the approach.

B. Experimentation results

Figure 10, Figure 11 represents respectively the load differences and the total load differences of the system when using the 3 approaches.

The results in *Figure 10* show that the curve of RTLB_Sched is the closest to zero and always converges to it unlike the other curves that diverge when more applications are mapped. The total load differences obtained were: 18.81 seconds, 30.40 seconds and 45.89 seconds for RTLB_Sched, Device Suitability and Round Robin respectively.

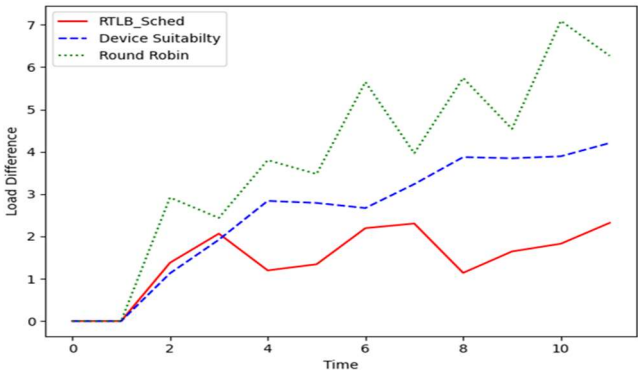

Figure 10. Load difference curve

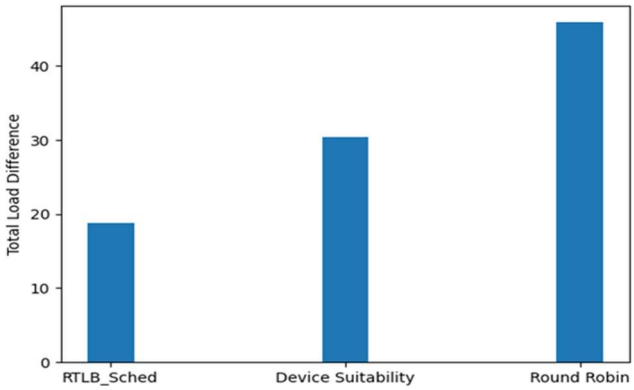

Figure 11.Total load difference

C. Results Discussion

Experimental results (presented in *Figures 10,11*) show that the proposed scheduling mechanism outperforms all the other scheduling schemes according to load balance performance.

RTLB_Sched outperforms the Device suitability and Round Robin by more than 1.5 times and 2,4 times respectively. It reduces the total load difference with 38.13% less than Device Suitability and 59% less than Round Robin.

In order to improve the load balance of the system, RTLB_Sched minimizes the difference between the CPU and the GPU Loads. Each newly submitted application will be assigned to either the CPU or the GPU, the scheduler calculates the load of the nodes in both cases before assigning the application. The scheduler assigns the application to the node that causes the minimum imbalance to the system even if it is the slower device which justify the performance of RTLB_Sched.

VIII. Conclusion

RTLB_Sched is a CPU-GPU Heterogeneous system resource manager that minimizes the load difference between the CPU and the GPU with no prior knowledge of the number or computation requirement of the applications. The key idea in RTLB_Sched is that for each submitted application, it has only two possibilities, either it will be assigned to the CPU or to the GPU, RTLB_Sched chooses to assign the application the device that provides the less load difference.

To calculate the node loads and the load difference, we provided mathematical formulas (*Equations 1, 5, 6 and 7*) that use the execution times of the submitted application. We resorted to machine learning techniques to predict the execution time of these applications.

Compared to other scheduling schemes like device suitability and round robin, RTLB_Sched shows significant improvements in the balance of the system as it provides the minimum load difference out of all the other scheduling schemes.

RTLB_Sched was developed for a single node heterogeneous system and is platform-dependent. It requires the applications' source code. Its performance is directly impacted by the precision of the machine learning models.

We intend to improve RTLB_Sched by:

- Scaling it to multi-node systems and compare it to recent works like [9].
- Generalizing the prediction models to different platforms.

References

[1] Vella F., Neri I., Gervasi O., Tasso S., "A simulation framework for scheduling performance evaluation on CPU-GPU Heterogeneous System", International Conference on Computational Science and Its Applications, Springer, Berlin, Heidelberg, 2012, https://doi.org/10.1007/978-3-642-31128-4_34

[2] Khalid Y.N., Aleem M., Usman A., Muhammad A. I., Islam M. A., Iqbal M. A., "Troodon A machine-learning based load-balancing application scheduler for CPU–GPU system", Journal of Parallel and Distributed Computing, Vol. 132, pp. 79-94, 2019, https://doi.org/10.1016/j.jpdc.2019.05.015

[3] Mahmoudi Sidi., Manneback P., Augonnet C., Thibault S., "Traitements d'images sur architectures parallèles et hétérogènes", Techniques et sciences informatiques (Computer science and technology), Vol. 31, pp. 1183-1203, 2012 https://doi.org/10.3166/tsi.31.1183-1203

[4] Olivier V., Pangfeng L., Jan-Jan W., "A collaborative CPU–GPU approach for principal component analysis on mobile heterogeneous platforms", Journal of Parallel and Distributed Computing, Vol. 120, pp. 44-61, 2018, https://doi.org/10.1016/j.jpdc.2018.05.006

[5] Harichane I., Makhlouf SA., Belalem G., "KubeSC-RTP: Smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems". Concurrency and Computation Practice and Experience e7108, 2022, https://doi.org/10.1002/cpe.7108

[6] Khalid Y.N., Aleem M., Prodan R., Iqbal M.A, Islam M. A., "E-OSched: a load balancing scheduler for heterogeneous multicores". Journal of Supercomputing, Vol. 74, pp. 5399–5431, 2018, https://doi.org/10.1007/s11227-018-2435-1

[7] Grauer-Gray S., Xu L., Searles R., Ayalasomayajula S., Cavazos J., "Auto-tuning a high-level language targeted to GPU codes, Innovative Parallel Computing, pp. 1-10, 2012, https://doi.org/10.1109/InPar.2012.6339595.

[8] Moez Ali. (2020). PyCaret: An open source, low-code machine learning library in Python. https://www.pycaret.org/.

[9] Usman A., Jerry C.W. L., Gautam S., Aleem M., "A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster", Soft Computing - A Fusion of Foundations, Methodologies and Applications, Vol. 25, pp. 407–420, 2021 https://doi.org/10.1007/s00500-020-05152-8