Machine Learning Techniques for Understanding and Predicting Memory Interference in CPU-GPU Embedded Systems

Alessio Masola, Nicola Capodieci, Benjamin Rouxel, Giorgia Franchini and Roberto Cavicchioli
University of Modena and Reggio Emilia, Italy
Department of Physics, Informatics and Mathematics
{*name.surname*}@unimore.it

***Abstract*—Nowadays, heterogeneous embedded platforms are extensively used in various low-latency applications, including the automotive industry, real-time IoT systems, and automated factories. These platforms utilize specific components, such as CPUs, GPUs, and neural network accelerators for efficient task processing and to solve specific problems with a lower power consumption compared to more traditional systems. However, since these accelerators share resources such as the global memory, it is crucial to understand how workloads behave under high computational loads to determine how parallel computational engines on modern platforms can interfere and adversely affect the system's predictability and performance. One area that remains unclear is the interference effect on shared memory resources between the CPU and GPU: more specifically, the latency degradation experienced by GPU kernels when memory-intensive CPU applications run concurrently. In this work, we first analyze the metrics that characterize the behavior of different kernels under various board conditions caused by CPU memory-intensive workloads on a Nvidia Jetson Xavier. Then, we exploit various machine learning methodologies aiming to estimate the latency degradation of kernels based on their metrics. As a result of this, we are able to identify the metrics that could potentially have the most significant impact when predicting the kernels completion latency degradation.**

***Index Terms*—GPU, memory interference, embedded, machine learning**

I. Introduction

Heterogeneous embedded boards are nowadays commonly used in widely deployed applications such as the automotive industry, Internet-of-Things (IoT) systems and industrial robotic applications. These platforms are characterized by multicore CPUs that work in concert with massively parallel accelerators, such as GPUs and/or other ASICs (Application Specific Integrated Circuits). There is a knowledge gap in research literature regarding the timing behavior that can be observed when multiple accelerators in the same board access shared memory hierarchies, such as the system DRAM. Previous work identified memory interference as a significant threat to both performance and predictability [1]–[3]. In addition, device vendors do not share the documentation containing the necessary architectural details that would allow a system engineer to understand and mitigate the effect of memory interference on real-time workloads.

In this paper, we aim to analyze the memory interference behavior by focusing on an embedded platform in which multiple CPUs compete with an integrated GPU to access the system DRAM. More specifically, we study the behavior of applications that execute on the GPU side, commonly called kernels, while memory-intensive applications are running on the CPU side, accessing the main memory shared by both processing unit type. We conduct our research using the well known Nvidia Jetson Xavier embedded board [4]. Such a board comes with a rich SDK for GPU-based application development in which several profiling tools are made available for the application developers. Such profiling tools are able to inspect hardware counters and kernel execution metrics that define the characteristics and behavior of GPU kernels. In this context, we aim to find correlations and useful insights about the relationship between these metrics and the slowdown in the execution time experienced by GPU kernels while varying the number of CPU co-running tasks that are accessing DRAM with variable memory requirements. This constitutes the starting point in defining methodologies able to account for the magnitude of CPU memory interference and the GPU kernel characteristics and provide predictions on the GPU kernel execution time degradation due to DRAM interference. Our contribution lies in the creation of methodologies to create predictive model that predicts the interference a GPU kernel can suffer from. To that end, we employ and combine known machine learning (ML) techniques: in this work we exploit Support Vector machines for Regression (SVR) [5], Random Forest Regressor (RFR) [6] and Deep Neural Network (DeepNN) [7]. Being able to understand, in a data driven fashion, the important factors that are involved in memory interference and thus predict kernel latency degradation are paramount to derive an accurate response time analysis [8], [9].

This paper is structured as follows:
In Section II, we present the related work. In Section III, we provide a brief summary of the architectural characteristics of a Jetson Xavier, the Nvidia CUDA API and the environment set for the experiments is described in Section IV. Subsequently, in Section V, we present the machine learning

methodologies that we used in our research to predict the experienced slowdown of a GPU kernel starting from its baseline metrics. In Section VI, we analyze which metrics are the most important as the result of the training phases for each individual ML technique. Finally, in Section VII we bring conclusive remarks and open future work.

II. Related Work

Interference on shared memory resources has been previously observed in both multicore CPUs [10]–[13], within the GPU computing clusters [14], [15] and between CPU and GPU in integrated SoCs [1], [3], [16]. Focusing on this latter topic, several approaches have been proposed in order to mitigate the effect of memory interference on both performance and predictability. Ali et al. [17], on an Nvidia TX2 implemented BWLOCK++, a mechanism that enables bandwidth reservation for the GPU kernels from CPU memory intensive activity by exploiting software throttling, hence reducing the CPU requested bandwidth. A similar, but specular mechanism has been proposed in [18]: in this work, software throttling to mitigate memory interference is used to protect CPU tasks by GPU memory activity. Houdek et al. [19] explored these ideas using hardware throttling: they analysed the memory controller behaviour in an Nvidia TX1 using hardware counters. They then implemented an hardware throttling mechanism for bandwidth regulation based on PREM [20] memory scheduling policies.

Both these characterization and mitigation research efforts allow us to both quantify memory interference and design schedulers accordingly; however, there is no work, to the best of our knowledge, that is specifically aimed at understanding and deriving a predictive model for CPU-GPU interference by focusing on the kernels characteristics. Moreover, previous research efforts assumed that the requested memory bandwidth is the best predictor for interference effects. We argue that a more in-depth characterization of GPU kernels can significantly widen our knowledge on interference. To this purpose, we exploit fine grained kernels profiling and machine learning (ML) methodologies. Previous work exists on exploiting ML approaches for deriving predictive models for latency deterioration caused by memory conflicts [21]–[25]. Saeed et al. in [21] proposed a mechanism that is able to predict the execution time of two co-running applications in a multicore processor; their predictive model is based on hardware performance events that have been previously selected using the Spearman correlation coefficient. Also regarding multicore CPU interference prediction and mitigation, Mishra et al. [22] chose relevant metrics using linear feature selection. Kim et al. [23] use a random forest regression model to predict interference starting from data collected in an offline profiling phase. As far GPU kernels are concerned, Ayub et al. [24] investigate interference detection during concurrent kernel execution on discrete GPU devices using different ML techniques. Their goal is to determine if two kernels are suitable for concurrent execution. Although effective, the dataset feature selection for their models is rather coarse-grained, as it is only based on four data items that refer to the kernels' launch configurations and register pressure. Since the profiling tools provided in Nvidia embedded boards allow the user to collect metrics, counters and other execution facts in a much finer granularity, we argue that those are paramount instruments to exploit. Such tools, although in a different context, have been extensively used in Bramley et al. [25]. The authors conducted a detailed analysis in the field of road vehicles and functional safety to determine whether bit faults in SDRAM cause computational errors in GPU kernels. They introduced the concept of *Liveness*, defined as the time in which data was last written and used during the entire kernel execution period, to evaluate the vulnerability of GPU memory and potentially compute its failure rate. For this purpose, they trained a neural network to predict the Liveness of a kernel, utilizing 147 performance metrics extracted on a kernel set using the `nvprof` tool. In our work, we take inspiration from this latter work in the different context of CPU-GPU memory interference prediction.

III. Background

In this section, we briefly summarize the most important architectural characteristics of a Nvidia Jetson Xavier, the GPU programming model and the selected environment for the experiments.

A. Nvidia Jetson Xavier Architecture

The Nvidia Jetson Xavier is a recently released embedded platform based on a custom System-on-Chip (SoC) architecture that combines multiple processing units and accelerators to provide high performance at a relatively low power consumption. Its internal components (Figure 1) includes a 64-bit ARM processor with 8-core CPU (Carmel) with a clock speed of up to 2.26 GHz, a Volta integrated GPU (iGPU) and other application specific accelerators. The CPU complex is a custom-designed processor based on the ARM-v8.2 specification. Internally, each pair of single cores is grouped together in a cluster that shares an L2 cache of 2 MB, meanwhile, each cluster shares a L3 victim cache of 4 MB. The iGPU, has access to a Last Level Cache (LLC) L2 of 512 KB. As visible in Figure 1, CPU and GPU share the same system RAM, which is a 16GB LPDDR4: that is the contention point and the platform subsystem on which we focus our experiments.

B. Programming Models and Tools

CUDA is a programming model and an API developed by Nvidia for general-purpose computing on GPUs. It allows developers to harness the power of GPUs to accelerate compute-intensive tasks, such as scientific simulations, image processing, and machine learning. The CUDA API is used to offload computations from the CPU to the GPU. This approach can result in significant speedups compared to traditional CPU-based computing, making it a popular choice for high-performance computing applications. The submitted work, referred to as kernels, is then executed on the GPU by

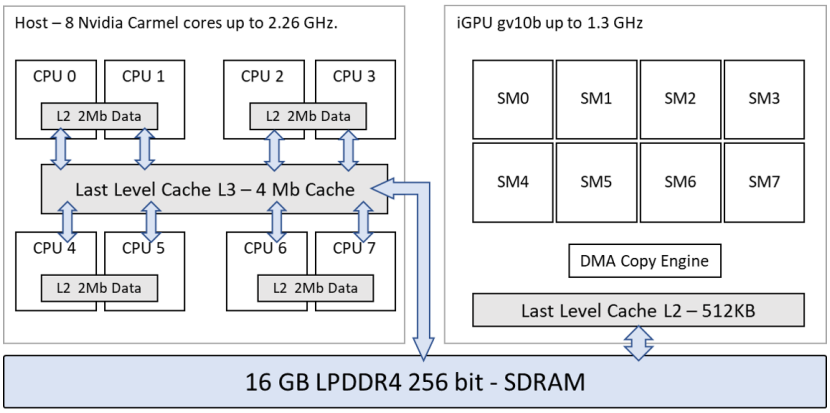


Fig. 1: Nvidia Jetson Xavier architecture. We focus on the upper part of this figure, in which the connection between the CPU complex and the integrated GPU is shown.

breaking down the computation into smaller tasks that can be executed in parallel. These tasks are then distributed across the Streaming Multiprocessors (SMs) of the GPU; these are the processing clusters able to execute block of threads which are themselves grouped into *warps*, which are the minimal scheduling entity of a GPU following the SIMD paradigm.

In this paper, we use the Nvidia Nsight Compute utility (`ncu`)[1], which is a powerful performance analysis tool designed for developers working on CUDA applications. It provides a comprehensive set of performance metrics and kernel execution statistics to help developers identifying performance bottlenecks and optimizing their code for maximum efficiency. With `ncu`, we can collect the performance metrics, have access to GPU hardware (HW) performance counter and other execution statistics able to define a complete profile of the individual kernel characteristics. `ncu` is a relatively novel addition to the CUDA SDK as its predecessor (the well known `nvprof`) has been recently deprecated [26]. Compared to its predecessor, `ncu` allows the system engineer to collect many more metrics, and it is able to autonomously derive for each metric the maximum, minimum and average values observed during an off-line profiling phase. Where applicable, the sum of the observed measurements or percentage are also automatically calculated by the tool. `ncu` provides close to 140000 metrics, however, the user is able to quickly select or filter the metrics he/she defines as interesting by interpreting the metrics names. The tool uses the following naming convention:

unit__(subunit?)_(pipestage?)_(interface)_qta_(qualifiers?)

While `unit` and `subunit` allow to specify the physical or logical part of the GPU being profiled, the `pipestage` refers to the pipeline stage. Additionally, the `qta` represents the dimensional units and `qualifiers` is used to apply additional filters or predicates to the counter. The primary unit metrics are defined by `counter_name.{sum, avg, min, max}`, where it is possible to have the sum, average, minimum, and maximum. Other metrics also include ratios, throughput, and other *roll-up quantifiers*. One relevant group of counters is the `Cycle Metrics`, which report the number of cycles within a defined unit clock domain. These metrics include cycles elapsed, active cycles (during which the unit was processing data), stalled cycles, and more.

[1]https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

IV. Setup and First Analysis

The initial profiling phase involves 14 different kernels with varying computational and memory requirements, ranging from basic operations to significantly more complex programs. These kernels are both internally implemented and sourced from known benchmark suites. We operated an initial classification on these kernels based on the amount of DRAM bandwidth demand they require, so that we can identify three categories: memory intensive, hybrid, and compute intensive. Kernels with a bandwidth demand exceeding 70% are classified as memory intensive (M), when below 10% are considered compute intensive (C), while the rest are classified as hybrid (H). Some additional information on the kernels are presented in Table I. This initial selection of kernels allow us to cover the largest input space: some of the kernels will suffer from interference as they require a lot of memory bandwidth to compute, while the impact of the interference will be limited on other kernels. This input space is therefore representative enough of real world applications, and enhance the accuracy of our model predicting the impact of interference on kernel execution even before the actual estimation of latency deterioration. We also understand that the more kernels are used to build the input space of the predictive models, the more confident our estimation will be. However, isolating a kernel from its context (i.e. its surrounding CPU application) is not a trivial task. Nonetheless, isolation is necessary in order to study the effect of memory interference on the actual kernel execution.

In this phase we analyze how metrics, read from hardware counters, vary during the execution of individual kernels as the intensity of CPU-generated interference changes. Such metrics

are obtained through the previously introduced `ncu`, that we launch via command line concurrently with our test application able to dispatch the previously described kernel set.

In order to generate variable CPU-side memory interference, an open-source software designed for heterogeneous SoCs has been set up, *HeSoC-mark* [16][2] (details in Section IV-B). The Nvidia board under test is configured for maximum performance (MAXN).

A. `ncu` and Kernels' Performance Metrics

The `ncu` tool allows us to delve into the details of the kernels' behavior through GPU-side hardware counters. From now on, readings from such counters will be referenced as metrics. A GPU has several different metrics that enable the analysis of memory and compute behavior in the context of kernels' completion latencies.

Due to the large number of available `ncu` profilable metrics, in this study we focused on only 45 metrics out of the close to 140000 available ones. The choice to profile only a limited number of metrics is to avoid influencing the kernels execution time with the profiler's overhead. Moreover, the board tends to crash when attempting to collect too many metrics during a single profiling run. We operated this initial filtering by following the naming convention explained in Section III-B; specifically, we focused on three macro categories: a first category that refers to GPU memory accesses (L2 and DRAM, starts with *lts*); a second category that refers to metrics related to compute instructions (starts with *gr, gpu, smsp__inst* or *__cycles*) and a final category that summarizes branch divergence (starts with *smsp__warp*). The reason for choosing these categories is because it has been shown to fit into what is known as principal components for GPGPU performance prediction and/or optimization [30], [31].

These profiled metrics are shown in Table II. Some of the observed metrics have been coalesced into a single one: this is because those metrics are combined to retrieve known `nvprof` old metrics for which we were more familiar with. For instance, metrics 11 and 13 from Table II map to `nvprof`'s *L2_read_throughput* and *L2_write_throughput* previous nomenclature. It is also important to note that the chosen metrics might be correlated, or sometimes a specific *rollup metric* (e.g. a sum rather than an average) carries more information than the other: since it is impossible to know these details in advance, we argue that the analysis performed in Section VI will clarify these issues.

B. Experimental Setup

As mentioned earlier, in order to generate interference from the CPU, we use the *HeSoC-mark* tool. Within the CPU complex, each individual core is numbered from 0 to 7. We designate core-0 as the core that submits work for individual kernels to the GPU, while we use the remaining seven cores (numbered from 1 to 7) to pin the processes that produce significant memory pressure through a specific application in *HeSoC-mark*. Such an application is named *meminterf*, which is able to generate high-memory traffic by iteratively executing memcpy or memset POSIX-defined functions over buffers of parametrized size. In our experiments, we set the buffer size to 50 MB and elect to use memset. This will effectively lead to the saturation the CPU-to-DRAM memory access traffic on the global memory shared with the integrated GPU, as already highlighted in previous research [1].

Tests of metrics and execution times for each individual kernel were performed without interference from the CPU side to generate a *baseline*. Then, we scaled up to a maximum of 7 tasks generating interference during the execution of individual kernels. Each CPU task is therefore an instance of *meminterf* pinned to a different core: this will lead to a linear scale of the magnitude of CPU-side memory interference. Previous research [16] measured that CPU-to-DRAM bandwidth for sequential accesses in the Nvidia Xavier is equally partitioned among sibling[3] cores.

The results of these experiments indicate that the hybrid (MVT) and memory-intensive (VADD) kernels experience slowdowns due to the high intensity of global memory accesses generated by the CPU, whereas the fully compute-intensive kernel (RAYTRACE) is not affected. As illustrated in Figure 2, which displays partial data for three kernels, the MVT kernel (Figure 2b) experiences the highest slowdown among the 14 kernels used in the experiments, with a factor of 2.19 when 7 *meminterf* processes are co-running. This confirms that M and H classified kernels experience a slowdown in the presence of high intensity of shared global memory accesses, and that DRAM bandwidth alone is not a good predictor on latency deterioration; in fact, MVT experiences a larger slowdown compared to VADD while requiring close to 20% less memory bandwidth.

V. Methodologies for Slowdown Prediction

We aim to create a method that allows us to predict the slowdown that a single kernel experiences as the number of interferents varies. We can find in the literature different examples of performance predictors that use ML technique to find the predicted performance of a certain problem [32], [33]. For this purpose, we use three different machine learning methodologies for solving this regression problem:

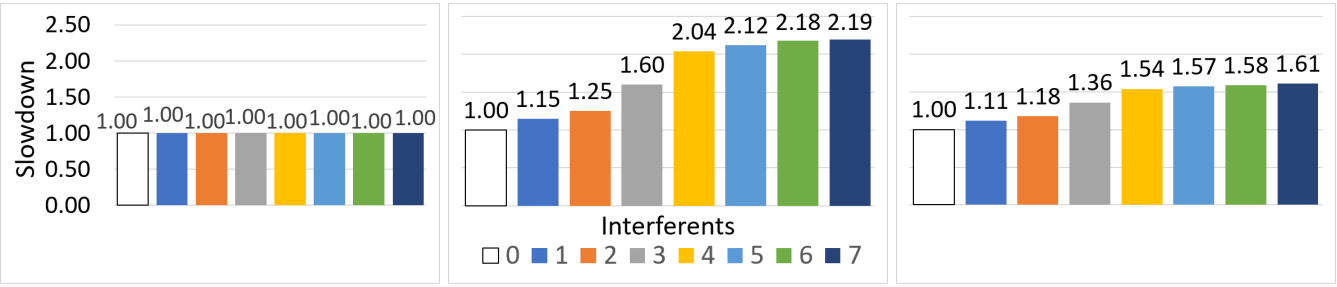
- **Support Vector machines for Regression (SVR)** is a machine learning algorithm used for predicting continuous numerical values. Unlike traditional regression models, which attempt to minimize the error between the predicted and actual values, SVR seeks to find an hyperplane, in higher dimensional space, that best fits the data while maximizing the margin between the predicted values and the boundary. This methodology is able to handle high-dimensional data with a small number of training samples and its robustness to outliers makes it an attractive choice for the interference prediction problem.

[2]https://git.hipert.unimore.it/mem-prof/hesoc-mark

[3]Sibling CPU cores are couples of cores sharing the same L2, as shown in Figure 1

TABLE I: Kernels used for analysis on Jetson Xavier architecture.

Kernel Name	Data Size In (MiB)	Data Size Out (MiB)	Bandwidth Demand (%)	Classification (Memory. Hybrid, Compute)	Origin
VADD	40 * 2	40	85.92	M	Synthetic / In House Implementation
SAXPY	40 * 2	40	83.55	M	Synthetic / In House Implementation
COPY	40	40	80.07	M	Synthetic / In House Implementation
RAYTRACE	-	1.5234	1.02	C	In House Implementation
DXTC	0.003 + 0.5	0.25	13.70	H	Cuda Samples
CONV	0.0001 + 40	40	21.34	H	In House Implementation
MVT	64 + 0.01 * 4	0.01 * 2	60.38	H	Polybench [27]
PF	0.5 + 255.5	0.5	50.73	H	Rodinia [28]
DOITGEN	16	16	85.12	H	Polybench
HOTSPOT	8 * 2	8	36.39	H	Rodinia
NBODY	4	2	27.09	H	Adapted from RoCm HIP Samples [29]
HISTO	24.3	0.003	55.93	H	In House Implementation
ATAX	262.14	0.032	76.61	M	Polybench
BFS	49.14	8.19	43.88	H	Rodinia



(a) RAYTRACE (b) MVT (c) VADD

Fig. 2: Slowdown experienced by different kernels as the number of interferents varies.

- **Random Forest Regression (RFR)** is a machine learning algorithm that combines ensembles of decision trees for solving regression problems. The RFR algorithm works by constructing an ensemble of decision trees, each of which is trained on a random subset of the input features and a random subset of the input data. Each tree is built and trained using a randomized version of the dataset. Once the trees are built, predictions are made by averaging the predictions of each tree. An advantage of RFR over SVR is its interpretability: each decision tree in the ensemble can be visualized and analyzed, hence allowing the system engineer to gain insights into the underlying relationships between the input variables and the output variable.
- **A Deep Neural Network (DNN)** for regression is a type of machine learning algorithm that is used to predict continuous numerical values. With DNN for regression the input data is passed through a series of layers, each consisting of interconnected neurons, to produce a prediction.

A. Training and Test

In order to train machine learning methods, a dataset of examples is used, consisting of X that defines the characteristics of an example, and Y that defines the values to be predicted. As a set of values X, we use all 45 metrics related to the baseline of the entire set of 14 kernels and the number of interferent processes (ranging from 0 to 7) co-running during the execution of each kernel. As a value Y to predict, we use the corresponding slowdown that each individual kernel experienced during the preliminary analysis (Section IV). Taking into account the small number of usable experiments (112 cases) available for training, we trained the models by splitting the examples into a training set of 85% and the remaining 15% as the test set. The best hyper-parameters for the models were determined using *Optuna* [34]. When we talk about hyper-parameters we are talking about all those quantities that are not changed during the training phase, but set upstream by the user.

During the initial training phase, sub-optimal results were produced due to the initial distribution of Y being unevenly distributed and with a large number (61) of slowdown cases $\leq$ 1.1. Therefore, to achieve good results during training, we removed 60% of slowdown values (36) that were below 1.1, resulting in a total of 76 cases with the corresponding distribution shown in Figure 3.

To evaluate the predictive models with never seen cases during the training phase, we created a parameterized kernel based on MVT, that we name PARKERNEL. We chose MVT as it was the kernel that experienced the highest slowdown during the profiling phase. PARKERNEL is governed by two parameters (p_0 and p_1), hence each couple $\langle p_0, p_1 \rangle$ allows us to consider kernels showing significantly different behaviors, as detailed in Section V-B. Subsequently, we use different models for predicting the slowdown by evaluating them on the

TABLE II: Metrics profiled for each kernel.

N°	Metrics	Description
1	gpu__cycles_active.avg	number of cycles where GPU was active
2	gpu__time_active.avg	total duration in nanoseconds
3	gr__cycles_active.avg	number of cycles where GR was active
4	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second	number of bytes requested for global loads (per seconds) - Efficiency
5	l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum	number of sectors requested for global loads
6	lts__t_sector_hit_rate.pct	proportion of L2 sector look-ups that hit (hit rate)
7	lts__t_sectors.avg.pct_of_peak_sustained_elapsed	number of LTS sectors (L2 utilization)
8	lts__t_sectors_aperture_sysmem_op_read.sum	number of LTS sectors accessing system memory for reads (system read bytes)
9	lts__t_sectors_aperture_sysmem_op_read.sum.per_second	number of LTS sectors accessing system memory for reads (system memory read throughput)
10	lts__t_sectors_op_read.sum.per_second	number of LTS sectors for reads
11	lts__t_sectors_op_read.sum.per_second + lts__t_sectors_op_atom.sum.per_second + lts__t_sectors_op_read.sum.per_second	L2 read throughput
12	lts__t_sectors_op_write.sum.per_second	number of LTS sectors for writes
13	lts__t_sectors_op_write.sum.per_second + lts__t_sectors_op_atom.sum.per_second + lts__t_sectors_op_read.sum.per_second	L2 write throughput
14	ltc__cycles_elapsed.{avg, sum}	number of cycles elapsed on LTC {average, sum}
15	lts__cycles_active.{avg, max, min, sum}	number of cycles where LTS was active {average, maximum, minimum, sum}
16	lts__cycles_elapsed.{avg, sum}	number of cycles elapsed on LTS {average, sum}
17	lts__d_sectors.{avg, max, min, sum}	number of sectors accessed in data banks {average, maximum, minimum, sum}
18	lts__d_sectors_fill_sysmem.{avg, max, min, sum}	number of sectors filled from system memory {average, sum, minimum, maximum}
19	lts__t_bytes.{avg, max, min, sum}	number of bytes requested {average, sum, minimum, maximum}
20	lts__t_request_hit_rate.pct	proportion of L2 requests that hit
21	lts__t_request_hit_rate.ratio	ratio of the proportion of L2 requests that hit
22	lts__t_requests.{avg, max, min, sum}	number of LTS requests {average, maximum, minimum, sum}
23	smsp__cycles_active.avg.pct_of_peak_sustained_elapsed	SM efficiency
24	smsp__inst_executed_op_global_ld.sum	instruction executed for global loads
25	smsp__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active	Load Store Unit (LDST) utilization
26	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct	proportion of warps per cycle
27	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct + smsp__warp_issue_stalled_misc_per_warp_active.pct	stalls for other reasons
28	smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct	stalls for memory dependency

generated kernels. For each models, we evaluate the minimum, maximum, average and absolute average of the percentage error (real-performance versus predicted-performance) observed during the prediction.

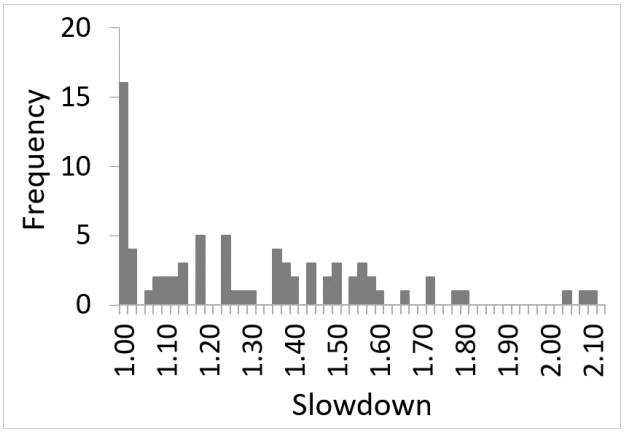


Fig. 3: Slowdown distribution of the 14 kernels.

B. A Parameterized Kernel: PARKERNEL

The parameterized kernel represents the core of the methodology we use to evaluate our predictive models. We could consider that, for each individual configuration of parameters p_0 and p_1, PARKERNEL generates kernels able to behave significantly differently from those generated with a different parameters couple.

Listing 1: PARKERNEL implementation

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
. . .
int j = 0;
for(j = 0; j < N; j++) {
    unsigned int f = 0;
    int m = 0;
    // compute/memory intensity
    for (f = 0 ; f < p0 ; f++)
        m += f + threadIdx.x;

    s[threadIdx.x] = m;
    // Branch divergence
    if(i % p1 == 0) {
       x1[i] += a[i*N+j] * y_1[j];
       x2[i] += a[j*N+i] * y_2[j];
    }
    else{
       x1[i] -= a[i*N+j] / y_1[j];
       x2[i] -= a[j*N+i] / y_2[j];
    }
}
x1[i] += s[threadIdx.x];
```

As shown in Listing 1, parameter p_0 is used to define the ratio of compute/memory instructions, while p_1 determines the

amount of branch divergence within a single warp. Setting p_0 and p_1 to 1 results in generating a kernel that behaves just like the MVT kernel; low values of p_0 generates more memory-intensive kernels, and conversely, larger values generates more compute-intensive ones. Similarly, low values of p_1 result in little to no branch divergence, while high values result in kernels that will be more frequently stalled due to branch divergence. This is because as shown in Listing 1, the conditional predicate in line 13 is based on the GPU thread ID, hence potentially breaking the GPU's lockstep model at warp level.

In order to understand whether or not this approach to generate validation kernels is able to reasonably cover the output domain (i.e. all the possible slowdown values, Y), we plot both how the slowdown changes as we vary the two parameters, keeping fixed at 7 the number of CPU interferents (Figure 4), and the total slowdown distribution among all the generated kernels on all the possible number of interferents (Figure 5). We set $p_0 \in$ [1, 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250, 300, 350, 400, 500, 1000] and $p_1 \in$ [1, 3, 7, 9, 10, 12, 14, 17, 23, 30]. Considering these steps and ranges and the number of interferents, we generate 180 different versions of PARKERNEL and 1440 individual experiments calculated by varying the number of interferents.

As shown in Figure 5, the distribution of slowdown cases for PARKERNEL is able to cover a wider range of possible slowdown values, and with less empty regions on the X axis compared to the slowdown distribution for the 14 kernels (Figure 3).

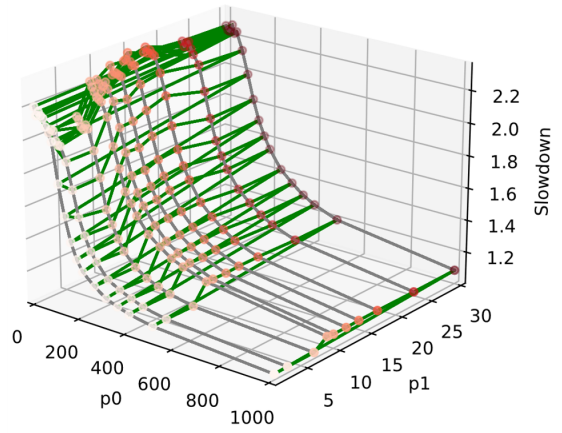


Fig. 4: Slowdown distribution of PARKERNEL depending on p_0 and p_1.

C. Methodologies and Results

When dealing with predictive models in real-time systems, a correctness margin has to be set. This is because it is unlikely that the prediction will match the reference value down to the very last significant digit. When presenting our results we will then set this margin to an arbitrary 20% as it is in line with both industrial and academic practice in many application

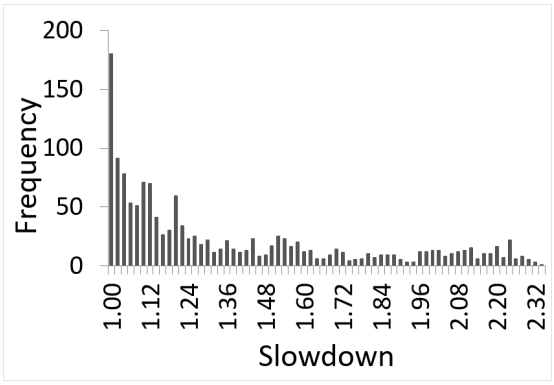


Fig. 5: Slowdown distribution of PARKERNEL.

domains for WCET estimation [35]–[37]. If a prediction error is below 20% then it is `in range` (Fig. 7 and 6).

1) Support Vector machines for Regression - SVR: As shown in Figure 6, we found the best configuration that has a good performance on validation, with an average percentage error of -4% and an absolute percentage error of 8.33%. The lowest underestimation was -35.15%, while the maximum overestimation was 16.29%. The SVR makes 770 overestimations and 670 underestimations (Figure 7).

2) Random Forest Regression - RFR: As shown in Figure 6, we found that the best configuration of Random Forest has an average percentage error of -2.83% and an absolute percentage error of 10.16%. The lowest underestimation was -43.09%, while the maximum overestimation was 22.32%. The RFR makes 866 overestimations and 574 underestimations (Figure 7).

3) Deep Neural Network - DeepNN: We found the optimal configuration during the training phase, shown in Figure 8, to be a neural network with 3 hidden layers, the first 2 of which have twice the size of the input (46 neurons), and the last hidden layer with a size equal to the input. As the final output neuron, since a real value needs to be predicted, a linear neuron is used to predict the value of slowdown. Dropout layers were used at each layer to ensure the reliability of the results and prevent the network from overfitting the training data. As shown in Figure 6, the best DeepNN performs an average percentage error of -3.30% and an absolute percentage error of 9.29%. The lowest underestimation was -34.24%, while the maximum overestimation was 13.46%. The DeepNN makes 777 overestimations and 663 underestimations (Figure 7).

4) Combined: The combined approach is a methodology that combines the single predicted slowdown of SVR, RFR and DeepNN by always taking the most pessimistic prediction among the aforementioned methods. As known in the literature combining different ML methods, as long as they are independent, guarantees improved overall performance. In our context, we can also introduce the bias related to interest versus pessimistic prediction. As shown in Figure 6, despite a 0.86% increase in the absolute average error of the best of the 3 methods (SVR), using a combined approach decreases the number of underestimations by 2% in the minimum percentage

underestimation values, and there is a normal distribution centered around 0, resulting in 913 overestimations, 527 underestimations and the `in range` is 1298 over 1440 (Figure 7), meaning that a 90% of the predictions fall within in the correctness range of 0-20%.

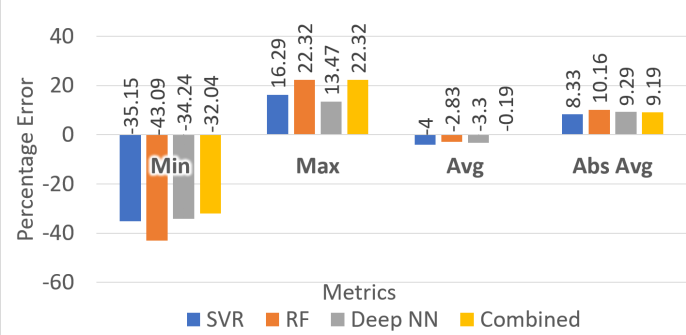


Fig. 6: Prediction results.

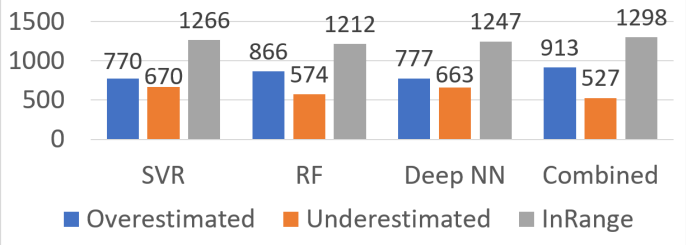


Fig. 7: Underestimation and overestimation count from the predictive models.

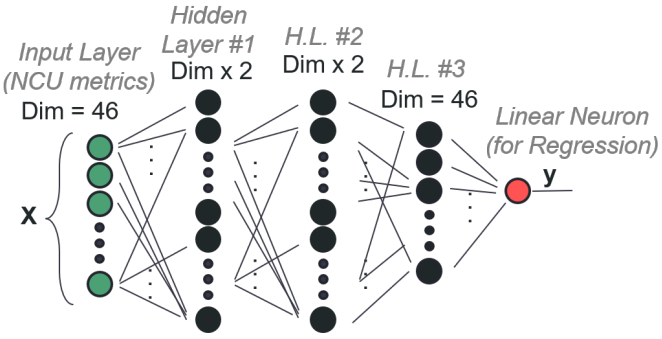


Fig. 8: The best configuration of the neural network .

VI. Metrics Analysis

Since the predictive model described in the previous section shows reasonable performance, we now move on to analyze and interpret the importance of metrics in relation to the kernels latency deterioration. For this purpose we used three methodologies:

- *Principal Component Analysis* (PCA) [38] is a dimensionality reduction technique that is used to transform high-dimensional data into a smaller set of linearly uncorrelated variables called principal components. The principal components are ranked in order of their contribution to the total variance of the data, with the first principal component explaining the most variance. PCA is commonly used to simplify complex datasets and to visualize patterns in data.
- *GINI importance* is a measure of the importance of each feature in a Random Forest model. It is based on the Gini index, a measure of impurity used in decision tree algorithms for the node splitting. The GINI importance of a feature is calculated by randomly permuting the values of that feature in the dataset and measuring the decrease in Gini index. The greater the decrease in Gini index, the more important the feature is considered. The GINI importance is often used to identify the most important features in a Random Forest model and to perform a feature selection for further analysis.
- *Spearman correlation factor*, also known as Spearman's rank correlation coefficient, is a statistical measure that quantifies the strength and direction of the relationship between two variables. It is calculated by ranking the values of each variable and then calculating the correlation coefficient between their ranks. The Spearman correlation factor is often used to measure the correlation among variables that do not have a linear relationship or that have non-normal distributions. It ranges from $[-1; +1]$, where a value of -1 indicates a perfect negative correlation, 0 indicates no correlation, and $+1$ indicates a perfect positive correlation. We use it to calculate the non-linear correlation between each single metric and the kernel slowdown caused by memory interference. For this analysis we flag as most important those metrics that present a Spearman coefficient $\geq 0,21$ or ≤ -0.21. 0.21 is the absolute average value calculated from the entire correlation set.

For each of these methodologies we highlighted the metrics with the highest correlations, i.e. by creating a set for each method, with each set containing the largest 10 values according to GINI, PCA and Spearman. After that, we trivially find the intersection set among the three. In this intersection set, we identified 5 metrics that are present in all the three sets, which are related to the number of CPU-side interferents, stalls for memory dependency, the proportion of warps per cycles, and the Load Store Unit utilization. Then, we find the union among the sets, by combining the individual top 10 of each analysis methodology. By doing so, we identified 18 metrics that were found to be the most influential using the different methods. The final 18 metrics and the values from the methodologies for analyzing the metric importance are reported in Table III.

To confirm the soundness of our approach in defining metrics importance, we retrained the three predictive methodologies and combined them using the pessimistic method by only considering the 18 metrics. Validation is then performed in the same way as before, i.e using PARKERNEL generated kernels. In this setting, the combined method produces a mean

TABLE III: Metrics that are more influential during slowdown prediction.

N° in Tab. II	Metric Name	PCA	GINI	Spearman
-	Interferents_nr (number of interferents by CPU side)	2.682	0.469	0.529
28	smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct	19.688	0.135	0.504
27	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct + smsp__warp_issue_stalled_misc_per_warp_active.pct	11.630	0.095	-0.378
26	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct	3.859	0.040	-0.378
25	smsp__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active	1.048	0.025	-0.343
7	lts__t_sectors.avg.pct_of_peak_sustained_elapsed	-	-	0.371
9	lts__t_sectors_aperture_sysmem_op_read.sum.per_second	-	-	0.304
10	lts__t_sectors_op_read.sum.per_second	-	-	0.301
4	l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second	-	-	0.286
12	lts__t_sectors_op_write.sum.per_second	-	-	0.285
24	smsp__inst_executed_op_global_ld.sum	2.168	0.022	-
22	lts__t_requests.sum	-	0.018	-
22	lts__t_requests.min	-	0.017	-
22	lts__t_requests.max	-	0.016	-
6	lts__t_sector_hit_rate.pct	0.737	-	-
19	lts__t_bytes.min	50.826	-	-
21	lts__t_request_hit_rate.ratio	5.439	-	-
23	smsp__cycles_active.avg.pct_of_peak_sustained_elapsed	1.458	0.022	-

and absolute error of 2.79% and 10.24%, respectively, with an estimate of the minimum and maximum slowdown of -32.21% and 21.95%, respectively, with 995 overestimates and 445 underestimates. Hence, the prediction quality compared to the previous system remained almost the same as what we reported in Section V. This result confirms and highlights that these 18 metrics are the most important for making decisions regarding the characterization of the behavior of GPU kernels and CPU-side memory interference.

VII. Conclusion

This paper presented an analysis of the behavior of GPU kernels under various memory-intensive CPU-side interference conditions. Different ML methodologies such as SVR, RFR, DeepNN, and a combined approach were used to predict the slowdown for GPU kernels caused by intensive CPU-side memory activity. The ML methodologies were trained using fined grained profiling, made possible using the Nvidia profiling tool `ncu`. Through the combined methodology, good results were achieved with a mean and absolute error percentage of -0.19% and 9.19%, respectively. Moreover, through the use of known dataset analysis methods such as PCA, GINI, and Spearman, 18 out of 45 initial `ncu` metrics were highlighted as the most influential predictors. This is then confirmed by re-training our proposed predictive model by restricting the feature set to only the 18 selected metrics. Such a predictive model trained with the restricted features produces an average and absolute percentage error of +2.79% and 10.24% respectively.

Such a combined predictive model can be used to analyze the behavior of boards under different computational loads between CPU and GPU, hence the potential to become a necessary tool for response time analysis and memory interference-aware scheduling.

As for future work we argue that the key metrics found with our proposed approach can be used to extend the current state-of-the-art heterogeneous task models in real time literature, e.g. DAG-based task models [39], [40]. These models are characterized by jobs as connected nodes of a directed acyclic graph: to each node, a set of timing parameters are associated, e.g. period, WCET, deadline, offset .... In heterogeneous DAGs each node can then be flagged to run on specific accelerators; this is where our findings come into play when extending these models: the key metrics detected as memory-interference predictors can be used to further enrich the associated parameters to each node, hence allowing the system engineer to account for memory interference when deploying tasksets on real HW.

Acknowledgment

This work was partly supported by ECSEL Joint Undertaking in H2020 project IMOCO4.E, grant agreement No.101007311.

References

[1] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–10.

[2] Q. Zhu, B. Wu, X. Shen, K. Shen, L. Shen, and Z. Wang, "Understanding co-run performance on cpu-gpu integrated processors: observations, insights, directions," *Frontiers of Computer Science*, vol. 11, pp. 130–146, 2017.

[3] D. Shingari, A. Arunkumar, and C.-J. Wu, "Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 22–33.

[4] M. Ditty, A. Karandikar, and D. Reed, "Nvidia's xavier soc," in *Hot chips: a symposium on high performance chips*, 2018.

[5] C. J. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery*, pp. 121–167, 1998.

[6] L. Breiman, "Random forests," *Machine Learning*, pp. 5–32, 2001.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Data Mining and Knowledge Discovery*, pp. 436–444, 2015.

[8] Z. Li and L. Wang, "Memory-aware scheduling for mixed-criticality systems," in *Computational Science and Its Applications–ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II 16*. Springer, 2016, pp. 140–156.

[9] Z. Lei, X. Lei, and J. Long, "Memory-aware scheduling parallel real-time tasks for multicore systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 04, pp. 613–634, 2021.

[10] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 1–10.

[11] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, 2011, pp. 374–385.

[12] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 155–166.

[13] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022.

[14] W. Zhao, Q. Chen, and M. Guo, "Ksm: Online application-level performance slowdown prediction for spatial multitasking gpgpu," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 187–191, 2018.

[15] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, "Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 653–663.

[16] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, "Contending memory in heterogeneous socs: Evolution in nvidia tegra embedded platforms," in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2020, pp. 1–10.

[17] W. Ali and H. Yun, "Protecting real-time gpu kernels on integrated cpu-gpu soc platforms," *arXiv preprint arXiv:1712.08738*, 2017.

[18] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "Sigamma: Server based integrated gpu arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pp. 48–57.

[19] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on arm-based heterogeneous platforms," in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*. IEEE, 2017, pp. 1297–1302.

[20] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 269–279.

[21] A. Saeed, D. Mueller-Gritschneder, F. Rehm, A. Hamann, D. Ziegenbein, U. Schlichtmann, and A. Gerstlauer, "Learning based memory interference prediction for co-running applications on multi-cores," in *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 2021, pp. 1–6.

[22] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 125–134.

[23] S. Kim and Y. Kim, "Co-scheml: interference-aware container co-scheduling scheme using machine learning application profiles for gpu clusters," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 104–108.

[24] M. Ayub and T. Helmy, "Concurrent kernel execution and interference analysis on gpus using deep learning approaches," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 10 193–10 204, 2022.

[25] R. Bramley, Y. Huang, G. Duan, N. Saxena, and P. Racunas, "On the measurement of safe fault failure rates in high-performance compute processors," in *2020 IEEE International Test Conference (ITC)*. IEEE, 2020, pp. 1–10.

[26] A. Saiz, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, "Top-down performance profiling on nvidia's gpus," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 179–189.

[27] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*. Ieee, 2012, pp. 1–10.

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[29] P. Bauman, N. Chalmers, N. Curtis, C. Freitag, J. Greathouse, N. Malaya, D. McDougall, S. Moe, R. van Oostrum, N. Wolfe *et al.*, "Introduction to amd gpu programming with hip," *Presentation at Oak Ridge National Laboratory.*, 2019.

[30] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, "Gpgpu benchmark suites: How well do they sample the performance spectrum?" in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 320–329.

[31] Y. Kim and A. Shrivastava, "Cumapz: A tool to analyze memory access patterns in cuda," in *Proceedings of the 48th design automation conference*, 2011, pp. 128–133.

[32] G. Franchini, V. Ruggiero, F. Porta, and L. Zanni, "Neural architecture search via standard machine learning methodologies," *Mathematics in Engineering*, pp. 1–21, 2023.

[33] S. Bonettini, G. Franchini, D. Pezzi, and M. Prato, "Explainable bilevel optimization: An application to the helsinki deblur challenge," *Inverse Problems in Immaging*, 2023.

[34] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[35] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovino, Z. Stephenson, B. Triquet, E. Quinones, C. Lo, E. Mezzetta, I. Broster *et al.*, "Timing analysis of an avionics case study on complex hardware/software platforms," in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015, pp. 397–402.

[36] K. P. Silva, L. F. Arcaro, and R. S. De Oliveira, "On using gev or gumbel models when applying evt for probabilistic wcet estimation," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 220–230.

[37] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, "Hiding communication delays in contention-free execution for spm-based multi-core architectures," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[38] M. Andrzej and R. Waldemar, "Principal components analysis (pca)," *Computers & Geosciences*, pp. 303–342, 1993.

[39] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The hpc-dag task model for heterogeneous real-time systems," *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1747–1761, 2020.

[40] J. Roeder, B. Rouxel, and C. Grelck, "Scheduling dags of multi-version multi-phase tasks on heterogeneous real-time systems," in *Proceedings of the 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-2021)*. IEEE, 2021.