Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel

10

GARGI ALAVANI and JINEET DESAI, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus, India
SNEHANSHU SAHA, APPCAIR, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus & Happy Monk AI, India
SANTONU SARKAR, APPCAIR, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus, India

The General Purpose Graphics Processing Unit has secured a prominent position in the High-Performance Computing world due to its performance gain and programmability. Understanding the relationship between Graphics Processing Unit (GPU) power consumption and program features can aid developers in building energy-efficient sustainable applications. In this work, we propose a static analysis-based power model built using machine learning techniques. We have investigated six machine learning models across three NVIDIA GPU architectures: Kepler, Maxwell, and Volta with Random Forest, Extra Trees, Gradient Boosting, CatBoost, and XGBoost reporting favorable results. We observed that the XGBoost technique-based prediction model is the most efficient technique with an R^2 value of 0.9646 on Volta Architecture. The dataset used for these techniques includes kernels from different benchmarks suits, sizes, nature (e.g., compute-bound, memory-bound), and complexity (e.g., control divergence, memory access patterns). Experimental results suggest that the proposed solution can help developers precisely predict GPU applications power consumption using program analysis across GPU architectures. Developers can use this approach to refactor their code to build energy-efficient GPU applications.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; **Machine learning**; **Machine learning algorithms**; **Ensemble methods**; • **Hardware** → **Power estimation and optimization;**

Additional Key Words and Phrases: GPGPU computing, XGBoost, CatBoost, CUDA, static analysis, sustainable computing

ACM Reference format:
Gargi Alavani, Jineet Desai, Snehanshu Saha, and Santonu Sarkar. 2023. Program Analysis and Machine Learning–based Approach to Predict Power Consumption of CUDA Kernel. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 8, 4, Article 10 (July 2023), 24 pages.
https://doi.org/10.1145/3603533

Authors' addresses: G. Alavani (corresponding author) and J. Desai, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus, Sancoale, Goa, India; emails: {p20160008, f20170168}@goa.bits-pilani.ac.in; S. Saha, APPCAIR, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus & Happy Monk AI, Sancoale, Goa, India; email: snehanshus@goa.bits-pilani.ac.in; S. Sarkar, APPCAIR, Department of CS&IS, BITS Pilani K. K. Birla Goa Campus, Sancoale, Goa, India; email: santonus@acm.org.

2376-3639/2023/07-ART10 $15.00
https://doi.org/10.1145/3603533

1 INTRODUCTION

General Purpose Graphics Processing Units (GPGPUs) are ubiquitously used in most of the compute-intensive, data-crunching applications where "performance at any cost" [15, 45] has been the key requirement from these applications. High throughput from a GPU comes at the cost of massive power consumption [10], which ultimately increases the total cost of ownership and limits the performance in the long run [15]. In fact, this became the trigger for the Green500 initiative[1] more than a decade ago.

Since then, the HPC community has been exploring various avenues for energy optimization in GPUs. One popular technique is to utilize **Dynamic voltage and frequency scaling– (DVFS)** based frequency tuning to control the power consumption in a CPU and a GPU. There have been attempts to balance the workload between CPUs and GPUs [39] to optimize the power/energy consumption. Researchers have attempted to monitor the power and energy consumed by a GPU program using direct methods such as external hardware and internal sensors for energy/power consumption profiling. There has been a significant interest in *indirect methods* such as the use of GPU performance counters [29, 40, 43], use of GPU simulators [13, 36], and **Parallel Thread Execution (PTX)** instruction-based power estimation [26, 48]. There have been in-depth investigations to understand the relationship between GPU hardware components and power consumption [18, 23, 36, 37]. While these approaches certainly have their own merits, it is equally important to revisit the power consumption issues of a GPGPU application from application design and deployment-time tuning perspectives. A recent study [16] demonstrates that fine-tuning a GPGPU program for performance actually hurts power consumption.

In the absence of any strong guiding principles, current software development practice performs only trial and error, where an application designer develops a GPGPU application using known best practices, runs it with extensive execution level profiling, and monitors the power/energy consumption. The process is iterated with alternative designs, different deployment configurations, and multiple datasets to optimize energy consumption. Such an approach has a high total cost of ownership and may not always be practical when the execution time is too large or the required GPU infrastructure needs to be hired on a rental basis.

Our work follows an *indirect* power modeling of GPGPUs, where we propose a machine learning–based prediction model using code features and minimal runtime information to estimate the power usage of a **Compute Unified Device Architecture (CUDA)** program without actually executing it on an NVIDIA GPGPU. Runtime information includes GPGPU program deployment, a.k.a. launch parameters, and other details such as latency and throughput, computed only once for a GPGPU by running a set of microbenchmarks. Our prediction approach is non-invasive, lightweight, and does not require any special hardware to run the prediction model. While hardware performance-counter-based power/energy estimation is tightly coupled with a specific architecture [10], our approach is not tightly coupled with any particular GPU architecture. Since our solution is primarily static analysis based, it can assist a GPU developer at the time of application design. For instance, a suitable product engineering of our solution can manifest as an Eclipse plugin,[2] which can act as an assistant for the developers to provide real-time feedback on power consumption during application development. Based on the predicted power and detection of code hotspots that contribute the most to power, the developer can refactor their code to build a power-efficient application. For instance, a developer can replace global memory instructions with shared memory instructions wherever feasible, since global memory instructions have proved to be a significant contributor to power consumption [26, 43]. Another example could be reducing

[1] https://en.wikipedia.org/wiki/Green500.
[2] https://www.eclipse.org/ide/.

the number of registers used for the application, since the usage of registers also impacts power consumption significantly [27]. The developer can also perform what-if analysis by changing the deployment configurations (launch parameters) and checking the impact on power consumption.

To the best of our knowledge, not much work has been done in this direction, though several attempts in the past have explored static input features [26, 53]. Also, we did not come across any algorithmic approach that establishes a relationship between power consumption and GPGPU program characteristics. As a result, we have employed a maching learning–based approach to model the complex interplay among the application characteristics, architecture features, and power consumption. We did stumble upon literature that used linear and logistic regression for the prediction of GPU power [31, 35], but these models are not able to predict the complexities in the dataset we have used. This observation has led us to use advanced ensemble machine learning algorithms, discussed in Section 5.

The main contributions of this article are as follows:

- We have created a dataset from a set of CUDA benchmark applications. We have defined a feature engineering methodology to identify a set of features that influences power consumption. This dataset and feature extraction code is made available.[3]
- We have performed an extensive empirical study across a set of machine learning approaches and demonstrated that the XGBOOST-based approach provides the best prediction model.
- We have reported how various CUDA program features impact power consumption in our study.

Though our work is based on NVIDIA GPUs, its outcome can be extended to other GPU architectures such as AMD, Asus, Intel, and so on. The remainder of the article is organized as follows: Section 2 discusses the related work. In Section 3, we provide details of the feature identification and acquisition. Feature selection techniques are presented in Section 4. Details of model techniques, their construction, and observed results are described in Section 5. We analyze the observed results in Section 6. We summarize our findings and possible future work in Section 7.

2 RELATED WORK

GPU power measurement and estimation can be briefly divided into (i) direct methods, including using external and internal hardware for monitoring power, and (ii) indirect techniques, such as modeling and simulation [10]. Utilizing external hardware sensors is costly for profiling large-scale distributed systems, and internal hardware sensors are still being verified for their accuracy and reliability. Hence researchers are also working extensively on indirect power modeling and measurements. Indirect power modeling includes models based on hardware performance counters, simulators, and PTX-based power models. DVFS is another popular approach for improving power efficiency. Researchers have shown how a combination of frequency tuning and task scheduling to multiple cores [33] and task allocation to heterogeneous devices [39] can help in achieving energy efficiency. Researchers have also studied the effects of frequency scaling on an NVIDIA K20 GPGPU [20]. There are more recent studies on the usage of DVFS to optimize power consumption [10, 18, 23]. However, DVFS is auto-enabled in NVIDIA GPUs, and unlike a CPU, there is no external control to fine-tune the frequency using DVFS. Moreover, the GPU frequencies have no relation (direct or latent) with the CUDA program structure or CUDA program deployment configurations. Thus, the DVFS-based power optimization strategy is orthogonal to our approach.

Ma et al. developed a statistical model for power prediction [41]. They utilized **Support Vector Machine Regressor (SVR)** and **Support Linear Machine (SLR)** to estimate the power used by

[3] https://github.com/santonus/energymodel.

the GPU over a small time window. SVR outperformed SLR emphasizing a complex model works efficiently over a linear model for power prediction. Nagasaka et al. [43] proposed a statistical method to estimate a CUDA kernel's power consumption using hardware performance counters. The approach uses GPU performance counters as the independent variable and power consumption as the dependent variable and trains a linear regression model. Song et al. [50] proposed the counter-based model built using a neural network. The disadvantage of the counter-based method is that some hardware counters may not be available, which affects the prediction model (only 13 counters were used in Reference [43]). Some architectures allow counters to a whole **streaming multiprocessors (SM)** that predicts asymmetric kernels inaccurately. Shuaiwen et al. [50] proposed a neural network model to estimate the average power using performance metrics. They utilize hardware performance counters with BP-**Artificial Neural Networks–(ANN)** based prediction models.

Although counter-based models are popular and effective, the execution of an application defeats its purpose. Many simulation-based power prediction tools are available that replace the need to execute an application [13, 46]. A work by Hong et al. [26] proposes an empirical runtime integrated power and performance prediction model for a GPGPU architecture. Its approach predicts the optimal number of active processors for a given application. However, their model fails to predict asymmetric and control-flow intensive applications. Singh et al. [48] applied non-linear functions on data collected using micro benchmarking to predict the power consumption of GPU. Zhao et al. [53] presented a PTX instruction count-based model to predict power prediction for the GPU, which is not suitable for a complex CUDA program.

Liang et al. [35] presented a generalized linear regression model to predict the GPU kernel performance by considering the control flow divergence. Though control flow divergence can have an impact on power consumption, we need to execute the kernel (whose power consumption we want to predict) and analyze its execution traces to derive the control flow divergence factor. There is no way to obtain an accurate estimate of the control flow divergence factor through a static, compile-time analysis. However, to include the impact of control flow divergence in our model, we did consider the number of branch instructions as a feature (refer to Table 1) to approximate control divergence using compile-time analysis of the CUDA program.

Kandiah et al. [31] proposed AccelWattch, a power modeling framework for GPUs. AccelWattch consists of four variants, of which three are based on simulation (two using Accel-Sim and one using GPGPU-Sim) and one uses hardware counters. Although AccelWattch performs reasonably, contrary to our approach, it needs a machine with GPU to collect the traces of a CUDA kernel that are then used in the simulator.

Although using a hardware counter-based modeling approach is popular, it needs multiple runs of an application to collect the data, unlike our approach. As seen in the work of Nagasaka et al. [43], some hardware counters may not be available, which affects the prediction model. Also, some architectures allow counters to a whole SM. In such a case, if there is an imbalanced number of cores utilized per SM, then the prediction may get affected [10].

The fact that the design of software indeed has an impact on the power the hardware consumes at runtime was investigated quite early [30]. Another study by Coplin et al. [16] empirically demonstrated how a CUDA source program could be optimized to improve energy consumption. Our work is based on the same principle. The approach by Moolchandani et al. [42] to use program features for power prediction using machine learning algorithms is similar to our work, except that the program features used in this study are collected using runtime analysis. Also, they have not utilized advanced ensemble learning algorithms (e.g., XGBoost, CatBoost), which have proved the most accurate in our experimentation.

Table 1. Features Considered for Feature Selection

Name	Description	Source
avg_comp_lat	Average computation instruction latency	PFEA
avg_glob_lat	Average global memory instruction latency	PFEA
avg_misc_lat	Average miscellaneous instruction latency	PFEA
avg_shar_lat	Average shared memory instruction latency	PFEA
branch	Number of branch instructions	PFEA
comp_inst_kernel	Number of computation instruction in kernel	PFEA
comp_inst_sm	Number of computation instructions per SM	PFEA
comp_lat_sm	Total Computation instruction latency per SM	PFEA
glob_inst_kernel	Number of global memory instruction in kernel	PFEA
glob_inst_sm	Number of global memory instructions per SM	PFEA
glob_lat_sm	Total global memory instruction latency per SM	PFEA
glob_load_sm	Number of global load instructions per SM	PFEA
glob_store_sm	Number of global store instructions per SM	PFEA
misc_inst_kernel	Number of miscellaneous instruction in kernel	PFEA
misc_inst_sm	Number of miscellaneous instructions per SM	PFEA
misc_lat_sm	Total miscellaneous instruction latency per SM	PFEA
shar_inst_sm	Number of shared memory instructions per SM	PFEA
shar_lat_sm	Total shared memory instruction latency per SM	PFEA
sm_active	Number of SMs activated	PFEA
n_warps	Number of warps per SM	PFEA
waves	Number of waves of blocks executed on SMs	PFEA
total_threads	Total number of threads launched	$grid\_size \times block\_size$
inst_issue_cycles	Instruction Issue Cycles	Equation (1)
cache_penalty	Time delay due to cache behavior	Equation (2)
glb_penalty	Time delay due to global memory instructions	Equation (3)
sh_penalty	Time delay due to shared memory instructions	Equation (4)
occupancy	The ratio of active warps on an SM to maximum number of active warps supported by the SM	CUDA Occupancy Calculator
reg_thread	Registers per thread	Cubin File
shmem_block	Shared memory per block	Cubin File
block_size	Number of threads per block	Provided by user
grid_size	Number of blocks	Provided by user

Highlighted rows indicate features that are finally selected by data-driven methods (Section 4).

In our previous work [2], we used some basic program features to train a prediction model. In the current work, we have added more complex features derived from scheduling a GPU application. We have also considered features that represent hardware-specific details such as Cache access, global as well as shared memory access, using dynamic analysis. We have employed a breadth of advanced machine-learning techniques to create an appropriate prediction model and demonstrated its efficacy.

3 FEATURE IDENTIFICATION AND ACQUISITION

While our modeling approach remains agnostic to a particular NVIDIA GPU architecture, and the approach can be generalized for multiple architectures, it is necessary to have an architectural overview before we introduce the model.

3.1 GPU Architecture Overview

Let us consider an NVIDIA Tesla K20 GPU for brevity. The GPU has 13 SMs. Each SM consists of fully pipelined 192 single-precision CUDA cores, 64 Double Precision cores, and 32 **special**

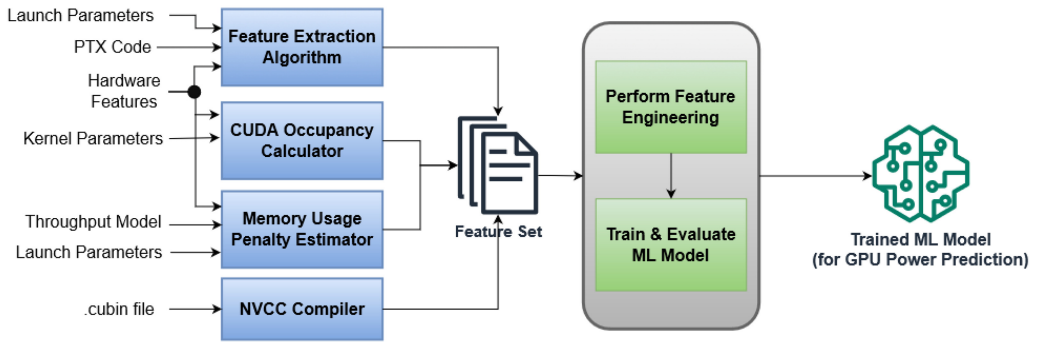

Fig. 1. Overview of power prediction modeling approach.

function units (SFUs).[4] While each core can execute one thread at a time, a set of 32 cores (synonymous to 32 threads for our purpose), called a *warp*, executes a single instruction (on multiple data). Each SM has four warp schedulers (denoted by nWS in our model) and eight instruction dispatch units (**number of dispatch units** (**nDU)** in our model), allowing four warps (of 32 threads each) to be issued and executed concurrently.

GPU architecture offers multiple memory options, such as global, shared, constant, and texture, that developers can access programmatically. While SMs run independently, they all access the data from global memory (main memory in DRAM). The programmer explicitly issues a global memory access instruction that causes all threads in a warp to fetch data from global memory simultaneously. The cores (threads) in an SM share a data cache called shared memory. A shared memory access instruction can cause a warp to simultaneously access 32 words in a shared memory. A global access time can be optimized if threads in a warp access consecutive locations in the global memory. Similarly, a shared memory access time is most optimal when the data requested by all threads in a warp are in different memory banks. The CUDA cores have access to a private set of registers allocated from a register file.

Power consumed by a GPGPU is influenced by a complex interplay among the (i) GPGPU architectural characteristics, (ii) CUDA program characteristics, and its runtime behavior such as instruction execution and data access, and (iii) launch strategy (determined by launch parameters). Features identified for power prediction in this work are listed in Table 1.

3.2 Generic Model of a GPU

Publicly available information on NVIDIA GPUs does not disclose crucial hardware execution details such as instruction latencies, cache behavior, memory access details, scheduling strategy, resource allocation, and so forth that influence GPU power consumption. To circumvent this issue, we propose a machine learning–based power prediction model that uses features derived from (i) GPU architectural features and (ii) CUDA program characteristics obtained from program analysis. Machine learning algorithms are being effectively used to predict architecture nuances for modern architectures [13, 43]. An overview of the approach is shown in Figure 1.

We first define a simplified structure of a GPU to express its architectural features succinctly. A GPU can be represented as $\mathcal{GPU} = \langle R, \mathcal{A}, \mathcal{TP}, \mathcal{PTX}, \mathcal{T}, \mathcal{L}, \mathcal{P} \rangle$, where

[4] https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

- R denotes the count of the following set of GPU resource types for each SM (i) Single precision cores, (ii) **Warp schedulers (WS)**, (iii) SFUs, (iv) Double precision units, and (v) **Load-store units (LSU)**. For instance, $R[LSU]$ denotes the number of load-store units present in an SM.
- $\mathcal{A}$ denotes the following set of attributes of a GPU:
(i) $access_{sz}$: This represents the number of bytes a GPU accesses for memory instruction, i.e., 128 bytes for global memory and 4 bytes for shared memory; (ii) **number of SMs (nSM);** (iii) nDU per SM; (iv) **Number of WSs (nWS)** per SM; and (v) nTh_m: Maximum number of threads per SM (vi) $L2\_sz$: L2 cache size
- $\mathcal{T}$: This is the set of all instruction types, namely {*C (Compute), GM (Global Memory), Sm (Shared Memory), M (Miscellaneous)*}.
- $\mathcal{PTX}$: The set of all PTX instructions (e.g., add, branch, shared load)
- $\mathcal{L}: \mathcal{PTX} \rightarrow \mathbb{N}$ is the latency function that denotes the number of clock cycles taken by an instruction to complete its execution.
- $\mathcal{TP}$: Throughput for global and shared memory instructions, denoted by $\mathcal{TP}(gm)$ and $\mathcal{TP}(sm)$ respectively.
- $\mathcal{P}$: Denotes a set of penalties $\mathcal{P} = \{cache\_penalty, glb\_penalty, shm\_penalty\}$.

Latency computation. Since NVIDIA does not publish latency values of a PTX instruction, a micro-benchmarking-based approach to derive latency values has been shown to be an effective solution [52]. Latency values can vary across different GPU architectures. We have used a set of microbenchmarks [4, 5] to derive the latency function $\mathcal{L}$ for a particular GPU architecture. Similarly to popular micro-benchmarking approaches [34, 52], we disabled compiler optimization to zero levels to avoid quantification errors in micro-benchmarking. The latency data collected are available in Section IV of the supplementary file.

We now discuss how various features are derived in the following subsection.

3.3 Registers per Thread, Shared Memory per Block, and Occupancy

Registers per thread and shared memory per block impact the occupancy of SM, which in turn influences hardware utilization. Past studies [26, 28] have shown that these attributes and occupancy play a crucial role in power consumption. We use .cubin files to generate these features.[5]

Studies have demonstrated that GPU power consumption is affected by occupancy [6, 50]. Occupancy also measures the thread-level parallelism in a CUDA program. Since our objective is to avoid runtime analysis, we utilize NVIDIA's CUDA Occupancy Calculator[6] to compute a theoretical occupancy for a CUDA program. Several architecture features are possible to derive without any detailed internal knowledge of a GPU. We describe these features in detail.

3.4 Instruction Issue Cycles

An NVIDIA GPU uses a **Fetch Decode Schedule (FDS)** unit to issue instructions to threads for execution. Naturally, an FDS unit contributes to the energy consumed by a GPU. To model the role of an FDS in power consumption, we have proposed a feature called Instruction issue cycles (*inst_issue_cycle*) in our previous work [2], which characterizes the number of batches of instructions issued. This is computed by dividing the total instructions issued for a kernel with a hardware limit of the number of instructions allowed per cycle. We use publicly available hardware details such as the nWS and nDU per SM to compute *inst_issue_cycle*. The dispatch unit is responsible

[5] https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.
[6] https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

ALGORITHM 1: Collect features of basic Block

```
1:  procedure CollectBBFeatures(GPU, G = ⟨V, E⟩, nTh_wave)
2:      c_bb = [n − 1], f = 0;
3:      for v ∈ V do
4:          Read( T(v) )                                  ▷ Read instruction type
5:          I(T(v)).count++;
6:          L(T(v)).total = L(T(v)).total+L(N(v));
7:          Read( N(v) )                                  ▷ Read instruction name
8:          N(v).count++;
9:      end for
10:     for each T(i) do
11:         c_bb[f] = I(T(i)).count; f++;
12:         c_bb[f] = L(T(i)).total; f++;
13:         if Iϵ(branch, globalload, globalstore) then
14:             c_bb[f] = N(i).count; f++;
15:         end if
16:     end for
17:     return c_bb;                                      ▷ Feature count for Basic Block
18: end procedure
```

for taking a given instruction, its warp ID, and the thread mask and calculating the thread IDs that will actually execute the instruction. nDU then sends each thread instruction to the applicable functional unit by checking its availability. Issuance of instructions is dependent on the total threads launched, total instructions, and warp scheduling policy by the GPU scheduler. We can extract total threads launched (*total_threads*) and total instructions (*total_inst*) using user-defined parameters and the feature extraction algorithm (Algorithms 1 and 2), respectively. Since the warp scheduling and idle cycles of the warp scheduler are difficult to predict, we assume that the warp is always full and doing useful work. The warp size (*warp_size*), which is typically 32 threads per warp, can be obtained by querying the GPU only once. We calculate *inst_issue_cycle* using the following formula:

$$inst\_issue\_cycle = \frac{total\_threads}{nWS \cdot warp\_size} \cdot \frac{total\_inst}{nDU}, \tag{1}$$

where $\frac{total\_threads}{nWS \cdot warp\_size}$ computes the number of batches of warps scheduled on GPU. We multiply this value by the number of batches of instruction scheduled ($\frac{total\_inst}{nDU}$). Since GPU is a pipelined architecture, we assume each batch of instruction is issued in one cycle.

3.5 Memory Access Penalties

The work by Nagasaka et al. [43] and Shuaiwen et al. [50] have demonstrated that GPU memory utilization is a significant contributor to power consumption. We model memory access overheads (that eventually contribute to power consumption) as penalties. To compute these penalty values, we require $\mathcal{TP}(gm)$ and $\mathcal{TP}(sm)$ throughput values. We have created empirical models for these two throughputs using micro-benchmarking data. Details of these model creations and validations are available in our previous work [3].

3.5.1 Cache Penalty. Global memory accesses are always routed through the L2 caches, and it is shared by all SMs. There is a significant delay in instruction execution if there is a cache miss.

ALGORITHM 2: Program Feature Extraction Algorithm

procedure FeatureExtraction($\mathcal{GPU}, G_{cfg} = \langle \mathbb{C}, \mathbb{E} \rangle, nB, nT, nLoop, n$)
  $f_{kernel} = [n]$; ▷ Feature array for Kernel of size n
  $nTh\_schd = \left\lceil \frac{nB}{nSM} * nT \right\rceil$;
  waves = 0;
  while $nTh\_schd \geq 0$ **do**
    $nTh\_wave = \min\{nTh\_schd, nTh_m\}$;
    BB' = GenerateCFG($\mathcal{GPU}, G_{cfg}, nTh\_wave$);
    for $G \in BB'$ **do**
      c_{BB} = CollectBBFeatures($\mathcal{GPU}, G, nTh\_wave$);
      if G has back-edge **then**
        for $f = 0$ to $n - 1$ **do**
          $c_{BB}[f] = c_{BB}[f] * nLoop$
        end for
      end if
    end for
    for $f = 0$ to $n - 1$ **do**
      $f_{kernel}[f] = f_{kernel}[f] + c_{BB}[f]; f++;$
    end for
    $nTh\_schd = nTh\_schd - nTh\_wave$;
    waves = waves + 1;
  end while
  total_inst = $\sum_{i=0}^{3} I(T(i)).count$;
  $f_{kernel}[n-1]$ = waves;
  return ($f_{kernel}, total\_inst$); ▷ Feature array collected for Kernel
end procedure

Cache penalty describes the delay after a cache miss occurs. Cache usage can be characterized using multiple cache performance metrics [49] such as cache misses, cache hits, average access time, and so on. Modeling accurate cache behavior using program analysis is an impossible task, since it depends on multiple runtime factors. We approximate this behavior by computing an L2 cache miss penalty using available architecture and program details to approximate the behavior of a GPU cache for global memory data quantitatively. L2_sz describes the L2 cache size collected using device query. We use the waves and *glob_inst_sm* computed in feature extraction Algorithm 1 and 2. Using the global memory latency (L(gm)), we calculate cache penalty as

$$cache\_penalty = \frac{total\_threads \cdot glob\_inst\_sm}{\frac{waves \cdot L2\_sz}{access_{sz}}} \times \mathcal{L}(gm). \tag{2}$$

3.5.2 Global Memory Penalty. Since global memory usage plays a crucial role in power consumption, we create a global memory penalty as a feature that quantitatively determines global memory usage. To do so, we first count the number of memory accesses using the number of load-store units per SM (R[LSU]), i.e., $\frac{total\_threads}{R[LSU]}$. Then we calculate the time required by each such access ($\frac{access\_sz}{\mathcal{TP}(gm)}$). We multiply the number of accesses by the time required to get a penalty for one global memory instruction. This penalty is then multiplied to global memory instruction per

SM (*glob_inst_sm*) to compute the global memory penalty by the following equation:

$$glb\_penalty = \frac{total\_threads}{R[LSU]} \times \frac{access\_sz}{\mathcal{TP}(gm)} \times glob\_inst\_sm. \tag{3}$$

3.5.3 Shared Memory Penalty. Shared memory instructions also significantly affect the power consumption of GPU, especially when kernel execution experiences a significant number of bank conflicts [50]. Hence we also consider the shared memory penalty feature to model this effect on power. Then we compute the penalty for one shared memory instruction by counting the number of times shared memory instructions are occurring on SM ($\frac{total\_threads}{R[LSU] \cdot nSM}$). Here nSM denotes the number of SMs in a GPU architecture and can be derived from a device query. We then multiply this penalty value for one instruction to the time taken for each of these shared memory instructions ($\frac{access_{sz}}{\mathcal{TP}(sm)}$) where *access_sz* denotes the number of bytes per access of shared memory. In the final computing shared memory penalty step, we multiply the number of shared memory instructions per SM generated using a feature extraction algorithm. The *shm_penalty* is computed as follows:

$$shm\_penalty = \frac{total\_threads}{R[LSU] \cdot nSM} \times \frac{access\_sz}{\mathcal{TP}(sm)} \times shar\_inst\_sm. \tag{4}$$

3.6 Program Feature Extraction Algorithm

The **Program Feature Extraction Algorithm (PFEA)** described in Algorithms 1 and 2 enacts GPU execution behavior based on static analysis of a CUDA program and generates additional features for the dataset. Depending upon the number of times the kernel code is run based on launch configuration, these algorithms compute the number of executed instructions, their total latency, the number of execution batches, and their average latency.

The number of threads ready for execution can be more than the maximum number of threads (per SM) that can execute in parallel on a GPU. In such a case, the GPU scheduler executes the threads in batches. We wanted to observe the effect of the number of batches on power consumption; hence it was included in the initial feature set. This batched execution is called "waves" in this work and is computed in Algorithm 2. Algorithm 2 calculates the total instructions executed (total_inst) by summing up the number of executed instructions of all instruction types in line 22. Since we take the number of loop iterations (nLoop) as input from the user, the total_inst includes the increase in the number of instructions if a loop is encountered. Table 1 shows all the features collected using the algorithm.

3.6.1 Illustration of PFEA. We illustrate how the Program Feature Extraction Algorithm extracts features with an example of neural network (nn kernel) shown in Figure 2. We consider a Tesla K20 GPU comprising 13 SMs for this illustration. To analyze a CUDA application such as nn kernel, we take its PTX code as the input (Figure 2(a)). We have highlighted branch and global memory load/store instructions in different colors. This code is converted into a **control flow graph (CFG)** of CFG (G_{cfg}) of basic blocks in Figure 2(b) as presented in line 7 of Algorithm 2. A CFG is popularly used to model the kernel application [24].

Recall that $\mathcal{T}(v)$ provides the type of instruction (e.g., computing, global memory) associated with the vertex v. Line 5 of Algorithm 1 accumulates the frequency of each instruction type in $\mathcal{I}(\mathcal{T}(v))$. Similarly, line 6 sums the total latency of an instruction type and stores it in $\mathcal{L}(\mathcal{T}(v))$. As explained in Section 3.2, $\mathcal{L}$ provides latency of particular instruction in clock cycles derived from micro-benchmarking. The count of global loads, global store instructions, and branch instructions are stored in $\mathcal{N}(\mathcal{T}(v))$ (line 14). The notation $\mathcal{N}$ indicates the name of the instruction (e.g., global load and global store). We include the global load and store instructions separately,

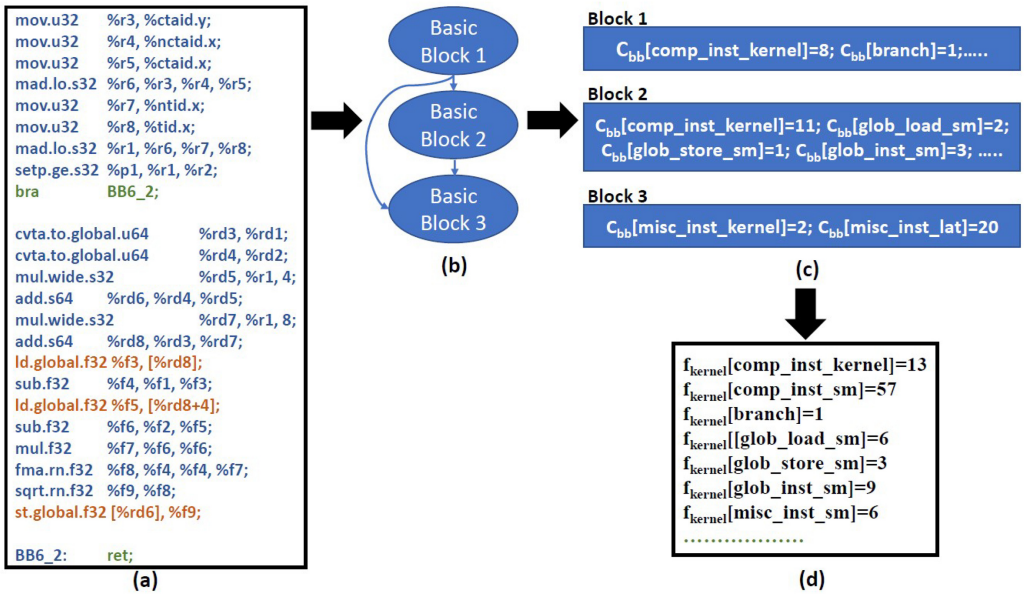

Fig. 2. Example of program feature extraction algorithm.

since many researchers suggest that global memory instruction impacts power consumption significantly [43, 50].

Algorithm 2 uses a feature array (f_kernel) to store the number of features for the CUDA kernel being analyzed. The user provides PTX code corresponding to a CUDA kernel along with launch parameters *block_size* (nT), *grid_size* (nB), and the number of loop iterations (nLoop) as inputs to FeatureExtraction Algorithm 2. Let us consider that the number of blocks (nB) is 78, and the number of threads per block (nT) is 1,024. In line 3 of Algorithm 2, we calculate the number of threads to be scheduled per SM, i.e., $nTh\_schd = \frac{nt \cdot nB}{SM} = 6144$. We then calculate the number of threads per wave (nTh_wave) based on the maximum threads per SM limit (2,048 threads for Tesla K20). In this case, $nTh\_wave = 2{,}048$, (line 6 of Algorithm 2).

The algorithm iterates to $nTh\_schd = 0$, ensuring all the threads are scheduled and complete their execution. In this example, the algorithm iterates three times ($\lceil \frac{nTh\_schd}{nTH\_wave} \rceil$) to cover all the threads. In each iteration, Algorithm 1 is called repeatedly for each basic block to generate basic block features. If the basic block has a back-edge, which means it is a loop, then the feature count of this basic block is multiplied by the number of loop iterations *nLoop* (supplied by the user), as shown in line 12 of Algorithm 2.

In the illustrated example, there is no loop. In the final step shown in Figure 2(d), we add the features collected during each wave and calculate the total features for a CUDA kernel. Since there were three waves as per the launch parameters, $f_{kernel}[comp\_inst\_sm]$ is the addition of computing instructions in the kernel($f_{kernel}[comp\_inst\_kernel]$) thrice (once during each wave) and hence equals 57. Similarly, $\mathcal{L}(\mathcal{T}(v))$ in Algorithm 1 represents the latency of each instruction type (which has been calculated for a given architecture using microbenchmarking). The algorithm keeps accumulating the latency for each instruction and stores it in ($f_{kernel}[total\_comp\_latency]$). Average computing instruction latency (($f_{kernel}[avg\_comp\_lat]$)) is then computed as $f_{kernel}[total\_comp\_latency]/f_{kernel}[comp\_inst\_kernel]$.

Table 2. Standard Deviation of Power Values Collected

Benchmark	Data Points	Average Power	Standard Deviation
vecADD	194	60.57	1.1830
clock	18	100.29	1.7796
NN	53	48.12	0.2668
Particle filter	630	62.22	1.5285
Cifarnet	206176	64.31	0.1359

3.7 Dataset Creation

We have created the dataset from three popular benchmark suites: The Rodinia [12], CUDA SDK,[7] and the Tango GPU [32]. We have created this dataset for three GPU architectures: Tesla K20, Tesla M60, and Tesla V100. Hardware specifications of all three GPUs are mentioned in Section V of the Supplementary File.

To create the dataset, we ran these benchmarks with different launch configurations (grid size and block size) and measured the power consumption of each benchmark using **NVIDIA Management Library (NVML)**[8] and **Unified Power Profiling Application Programming Interface (UPPAPI)** [11]. NVML is a C-based programmatic interface for monitoring and managing various hardware states within NVIDIA Quadro and Tesla Line GPUs. Both NVML and UPPAPI libraries use on-chip sensors for power measurement. NVML is an underlying library for the NVIDIA-supported `nvidia-smi` tool, which is thread-safe to make simultaneous NVML calls from multiple threads. You call your CUDA kernel within a function call to NVML and UPPAPI. When the kernel code executes, the GPU power consumed is stored in a text file. From the collected power values, an average value is recorded as measured power for the kernel. The standard deviation of data collected for some popular benchmarks is presented in Table 2.

Dataset creation can be susceptible to noise. We believe that the likelihood of attribute noise in the dataset is minimal and does not impact the prediction model. Since the architecture-related features are obtained using reliable/certified tools, these feature values are highly unlikely to have any noise. Attributes related to CUDA kernel program characteristics are essentially structural properties of the code. Therefore, it is unlikely that there can be noise in extracting these characteristics. We have used a set of regression-based models to derive various penalty values. These values can have some noise related to erroneous information. If this introduces a noise, then the noise will be present in the complete dataset, and as such, the noise is assumed to be Gaussian. We have measured the power consumed by a CUDA kernel from onboard sensors via APIs developed by NVIDIA. This is not a likely source of the noise. Also, we observe that the XGBoost algorithm, which offers the most accurate prediction on this dataset, is generally found to be relatively robust to noise [22].

4 FEATURE ENGINEERING

Of all the features considered in this study (refer Table 1), we systematically select the features that are likely to predict power consumption accurately. While choosing the final features for the model, we followed the following approach.

(1) Derived Attributes: We preferred to use derived attributes, as this is one of the ways to inject expert knowledge into the model. For instance, we used average latency computed from these two attributes instead of using total latency and total instructions per SM. Consequently, we

[7] https://docs.nvidia.com/cuda/cuda-samples/index.html.
[8] https://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html.

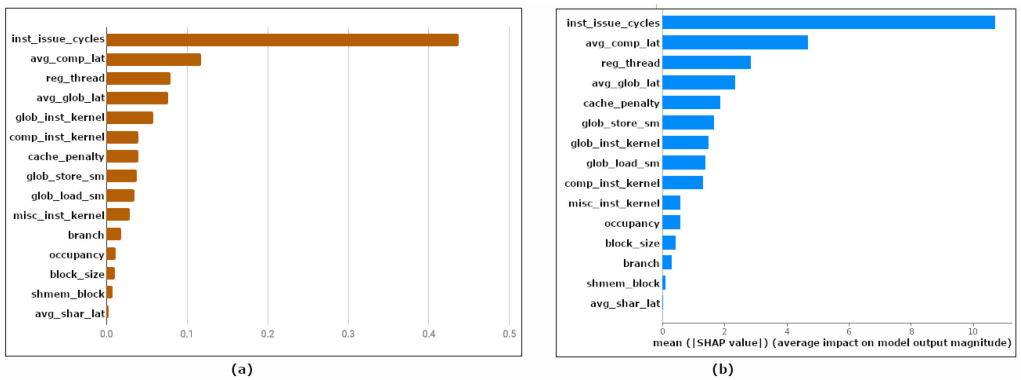

Fig. 3. Feature importance plot using (a) random forest and (b) SHAP.

dropped *comp_inst_sm*, *glob_inst_sm*, *misc_inst_sm*, *shar_inst_sm* (total instructions per SM) and *comp_lat_sm*, *glob_lat_sm*, *misc_lat_sm*, *shar_lat_sm* (total latency per SM) features and included only the average latencies.

(2) Non-correlated Attributes: Having highly correlated features in our model does not improve the model's performance; however, they may mask the interactions among various features. This suggests we include only one of the highly correlated features in our model. We use the correlation analysis on the remaining 24 features to observe the relationship of attributes among themselves. We consider two types of correlation coefficients: the Pearson coefficient and the Kendall coefficient. We considered a correlation coefficient 0.85 to remove the most highly correlated features. Using the resulting heatmap of Pearson correlation coefficient, we drop *sm_active*, as it is strongly correlated to *block_size* with correlation coefficients 1.0. Similarly, we observe that *inst_issue_cycles* is highly correlated with *glb_penalty* (0.914), *n_warps* (0.85), *waves* (0.85), *grid_size* (0.85), and *total_threads* (0.92).

The Kendall coefficient is more robust to outliers than Pearson. Also, the Pearson coefficient has an assumption that the variables are normally distributed. However, this may not always hold. So we also utilize the Kendall coefficient for our analysis, which helps us find further monotonic relationships among the features apart from linear relationships highlighted by Pearson analysis. After discarding the features based on the Pearson coefficient, we perform the Kendall correlation analysis on the new feature set (18 features). Based on the heatmap generated using the Kendall correlation coefficient, we observe that *avg_shar_lat* is highly correlated with *shar_inst_kernel* (0.89) and *sh_penalty* (0.94). Therefore we drop the two features and keep *avg_shar_lat* in the final feature set of 16 features. The Heatmap of the Pearson and Kendall coefficients is included in Section II of the supplementary file.

(3) Impactful Attributes: We have used Random Forest Regressor feature importance with 16 features and then systematically removed the features that have no impact on the prediction accuracy. In this process, we have removed the *avg_misc_lat* feature. We also utilized **Shapley Additive explanations (SHAP)** [38] to verify the features that contribute the most to power prediction. Comparison of Random Forest and SHAP feature importance results for Tesla K20 GPU are presented in Figure 3. As seen in the figure, these results are closely consistent, since the top four features and two least important features are the same, with a minor difference in order for other features. Pearson coefficient results for these features are included in Section II of the Supplementary file, which confirms a very low correlation between these final 15 features. The final set of features is highlighted in Table 1.

Table 3. ANN Model Experiment Results

Model Parameters		Smaller dataset		Larger dataset	
ANN Architecture (3 hidden layers)	Learning rate	Epochs	R^2 score	Epochs	R^2 score
15 features selected using feature engineering, uniform learning rate					
5,5,5	0.01	130	0.6226	135	0.5003
64,64,128	0.01	150	0.7011	120	0.6791
15 features selected using feature engineering, Exponential decay learning rate					
64,64,128	0.9	175	0.7246	150	0.7057
15 features selected using feature engineering, Inverse time decay learning rate					
64,64,128	0.5	100	0.7418	50	0.7737
VAE for feature extraction (latent features:15), Inverse time decay learning rate					
64,64,128	step decay		0.3270		0.00802
Auto feature selection (without VAE), uniform learning rate					
5,5,5	0.01	150	0.6081	150	0.53970
64,64,128	0.01	150	0.7084	120	0.6484

5 MODEL SELECTION AND EVALUATION

We undertook a systematic study of classical ML and Neural Network models to arrive at the final model based on the model mechanics and performance. We first investigated whether a linear relationship between predictor variables (pairwise) and between the response and predictor variables can be established. As evident in the plots of the predictor features versus response, i.e., power (Section III of Supplementary file), there is no linear relationship between input features and GPU power consumption. The R^2 score turned out to be -0.1919 for the multi-linear regression model, an indicator of poor model fitting on the experimental data. This observation led us to explore more complex, non-linear regression techniques, as presented in the following section. As shown later in the article, *CatBoost and XGBoost* are the two state-of-the-art models that are eventually chosen based on the performance and relevance with respect to selected features.

5.1 ANN Experiments and Analysis

Among the wide variety of available methods and algorithms, ANN [25] is one of the most popular methods while modeling non-linear relationship models on complex patterns in data, especially in prediction problems. We conducted extensive experimentation with ANN by changing the ANN architecture, dataset size, and learning rates. We have used multiple feature selection methods:

(i) Feature engineering, described in Section 4 and Table 1
(ii) **Variational autoencoder (VAE)** with 15 latent features and
(iii) Auto-feature selection (without VAE).

We trained ANN with three hidden layers for two different layer configurations. To investigate whether ANN performs better with a larger dataset, we ran the experiments for a smaller dataset (990 tuples) and a larger dataset (1,810 tuples). Details of ANN experiments are presented in Table 3. Our findings are as follows:

(1) We observed that ANN performs comparatively better with a three-layer MLP structure with 64, 64, and 128 neurons, among a few other architectures (varying neurons in hidden layers) we attempted.

(2) When training an ANN model, it is often useful to lower the learning rate as the training progresses and observe its effect on model performance. In addition to the uniform learning rate

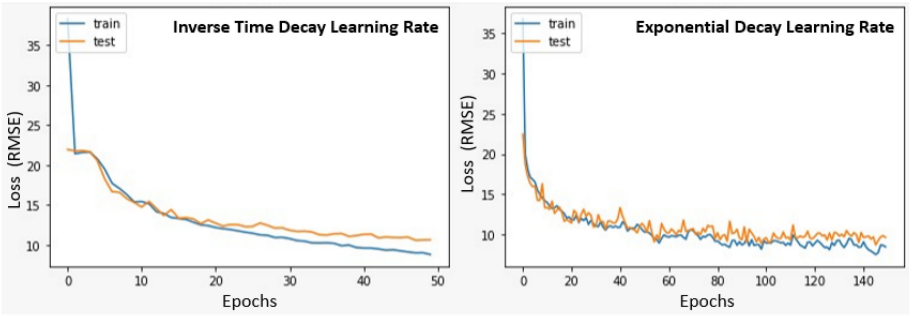

Fig. 4. Loss function plot for adaptive learning rate ANN.

(first two result rows of Table 3), we used the adaptive learning rate method with (i) exponential decay and (ii) inverse time decay learning rate schedules. For all three learning rate methods, we have used the features selected from the feature engineering method. The inverse time decay model was the most precise among all the ANN models trained, and it was the only method that performed better on the larger dataset.

(3) From the loss function plot for both the methods shown in Figure 4, validation loss is seen to be greater than the training loss as the number of epochs increases. This is evidence of over-fitting. Moreover, the training plots do not exhibit uniform smoothness and monotonicity. So the loss is likely to experience non-smooth behavior.

(4) VAE with 15 latent features (number of features are kept the same as the feature engineering approach) with *inverse time decay* shows an unacceptably low R^2 score, as shown in Table 3. Increasing the number of latent features did not yield any significant improvement. The auto feature selection approach performed better than VAE-based approach, as shown in Table 3, and the R^2 values are comparable to the feature-engineering-based approach.

5.2 SVR-based Model

SVR [17] considers the presence of non-linearity in the data and provides a proficient prediction mode. After experimenting with popular kernels for SVR, we observed that the **Radial Basis Function (RBF)** kernel gives the most precise model. However, an R^2 score of 0.7041 for the RBF kernel for the Tesla K20 GPU is still not acceptable for a power prediction model.

Since the multi-linear regression model such as ANN and SVR cannot accurately map the power consumption to input features and are shown to over-fit, we conclude that these models are not sufficient for the power prediction problem. Additionally, Neural network regressors are data hungry and, therefore, usually fail to model reasonably well in the absence of abundant data [47].

5.3 Random Forest–based Regressors

When the data are limited and has a significant number of feature attributes, tree-based regression models have proved to be highly efficient [8, 9, 13]. As the next step, we consider tree-based methods like Decision Trees, which model well on non-linear relationships. The resultant R^2 score of 0.8447 for decision trees for the Tesla K20 GPU shows promising results for tree-based approaches. We further explore advanced tree-based algorithms, which are known to perform better than decision trees.

Random forests (RF) offer higher resolution in the feature space, since the trees are not pruned and split the feature space into smaller regions. RF utilizes a random subsample of rows at each node, and a distinct sample of features is selected for splitting. Since RFs are learned on diverse samples, they are endowed with greater diversity. Each tree in RF produces an individual prediction,

Table 4. Machine Learning Model and Its Hyperparameters

Random Forest	
n_estimators	500
max_depth	14
min_sample_split	2
max_features	auto
min_samples_leaves	1

XGBoost	
n_estimators	500
max_depth	6
learning_rate	0.05
min_child_weight	1
colsample_bytree	0.7
subsample	0.7
silent	1

Extra Trees	
n_estimators	220
max_depth	16
min_sample_split	2
max_features	auto
min_sample_leaves	1

CatBoost	
depth	8
iterations	270
loss_function	RMSE
logging_level	Silent

Decision Tree	
max_depth	14
sample_split	5
criterion	mse

SVR	
C	200
epsilon	1
gamma	0.5
kernel	rbf

Gradient Boosting	
n_estimators	250
max_depth	5
learning_rate	0.17
max_features	auto
criterion	friedman_mse

and the final prediction is an average prediction of all trees. The averaging ensures smoothness in training and validation loss and, along with RF's randomness, improves its accuracy and reduces overfitting.

RF has some advantages compared to ANN in specific cases, including their robustness and benefits in cost and time. RF is also particularly advantageous in terms of interpretability [1] in energy consumption data. Random forest is found to be a better choice than a decision tree or Line-fit regressors, as empirically observed from the results seen in Table 5. For Tesla K20, the R^2 score of RF is 0.8899. We considered other popular tree-based ensemble machine-learning algorithms to ascertain the most accurate and efficient algorithm in GPU power prediction. These algorithms include ExtraTrees Regressor [21], Gradient Boosting Regressor [19], XGBoost Regressor [14], and CatBoost [44] to predict GPU power consumption. These algorithms cover a breadth of ensemble approaches in predicting continuous variables. XGBoost turned out to be the most efficient and accurate predictor model for power consumption. Experiments, results, and analysis of these approaches validate the superior performance of XGBoost and are discussed further in Section 6.

5.4 Model Building and Training: Implementation Details

Machine learning models work much better if the values of the features in the dataset are relatively on a similar scale. The tree-based models are scale invariant. However, we scaled the data, since it is a useful technique to accelerate the calculations. We use the MinMaxScalar technique here, since it preserves the shape of the distribution of the features, which, in turn, preserves the information embedded in the original data. The obtained dataset is shuffled to remove skewness from the dataset.

The tree-based bagging and boosting models are constructed using the Tree Regression method provided by the `scikit-learn` (Version: 0.24.1). Simple cross-validation uses the same data for hyperparameter tuning and evaluates the model's performance. This may lead to a biased evaluation of the model. So we consider using nested cross-validation where the scores of each hyperparameter combination are computed on all the splits, and the best parameter combination is thus obtained. This best parameter combination is used to calculate scores on all the splits again.

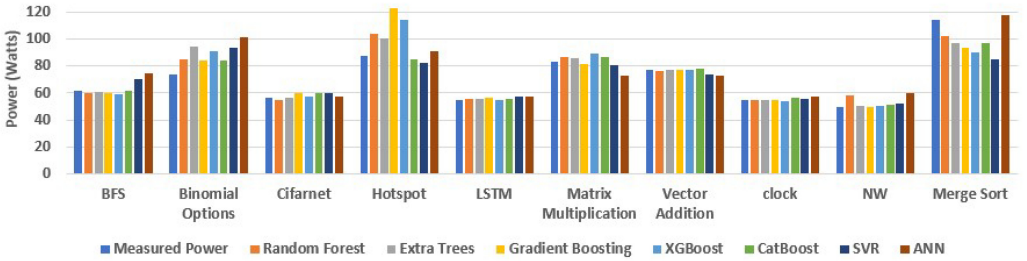

Fig. 5. Measured vs predicted power consumption for popular benchmarks.

The parameters tuned for these regression models are mentioned in Table 4. *n_estimators* denotes the number of trees used in the model, *max_depth* represents the maximum depth a tree can attain, *min_samples_split* indicates the minimum number of samples to split an internal node, max_features denotes the number of features to consider for best split, and the split criterion is used to measure the quality of split (can be MSE or **Mean Absolute Error (MAE)**). The Support Vector Regression model with the Radial Basis Function kernel is built using the `scikit-learn` library. We use the nested cross-validation technique mentioned earlier to tune parameters like kernel co-efficient, regularization parameter, and the epsilon distance from the actual value for which no penalty is associated.

The Keras library (Version: 2.4.3) is utilized to build the ANN model. For ANN model results used for comparison with other ML algorithms, we employ the incremental approach for constructing the ANN model [51], and finally, we come up with three fully connected hidden layers of 64, 64, and 128 neurons each. It uses the Adam optimizer for performing the gradient descent. We utilize the dropout technique and regularization to prevent overfitting.

Model Validation. We use nested cross-validation for optimal parameters like the number of epochs, batch size, and learning rate by observing the training and validation loss. We use sklearn's k-fold **cross-validation (CV)** method[9] In fivefold cross-validation, we split the dataset into five folds. For each unique fold, we considered one fold as a test dataset and took the remaining folds as the training dataset. Then our model is fit into the training set and evaluated on the test set. Moreover, we ran our model 5 times and reported the mean and standard deviation of the performance. As we observe from Section 7, the performance is robust, as the standard deviation from the mean performance is low on all five runs.

6 FINAL MODEL ANALYSIS

We present here the measured versus predicted power consumption of some of the popular benchmarks belonging to all three benchmark suites in Figure 5 for all the machine learning models considered in this study. For a better presentation, we chose a small set to present in Figure 5 and not all the benchmarks considered in this study.

Analysis of Results: Experimental results suggest that the XGBoost model outperforms all the other algorithms for all three GPU architectures under study. CatBoost loss plot follows its

[9]https://scikit-learn.org/stable/modules/cross_validation.html for splitting training and testing data. A k-fold CV ensures that prior knowledge about the test set may not "leak" into the model and evaluation metrics report generalization performance. A test set is held out for final evaluation, but the validation set is no longer needed when doing a CV. The training set is split into k smaller sets, and for each of the k folds, a model is trained using $k-1$ of the folds as training data, and, subsequently, the resulting model is validated on the remaining part of the data implying the remaining part is used as a test set to compute a performance measure such as accuracy. We set $k = 5$ for fivefold cross-validation.

Table 5. Validation Score for Machine Learning Algorithms across GPU Architectures

Architecture	Regression Model	R^2 score	RMSE	MAE
Tesla K20	Random Forest	0.8899	7.9265	4.4177
	Extra Trees	0.8893	7.9438	3.9346
	Gradient Boosting	0.9025	7.4657	3.7849
	XGBoost	**0.9101**	**7.1592**	**3.4816**
	CatBoost	**0.9067**	**7.3001**	**3.8308**
	Decision Tree	0.8447	9.3899	5.007
	SVR	0.7041	13.0078	8.4063
	ANN	0.7011	12.1048	7.4597
Tesla M60	Random Forest	0.9398	4.5547	2.2427
	Extra Trees	0.9557	3.9610	1.9362
	Gradient Boosting	0.9415	4.3645	2.0169
	XGBoost	**0.9544**	**3.9172**	**1.7588**
	CatBoost	**0.9548**	**3.89382**	**1.8252**
	Decision Tree	0.8762	6.6781	3.7151
	SVR	0.6021	12.0735	8.7283
	ANN	0.7314	8.8217	5.9993
Tesla V100	Random Forest	0.9415	8.8127	3.7447
	Extra Trees	0.9526	7.9428	3.1989
	Gradient Boosting	0.9595	7.4883	3.2475
	XGBoost	**0.9646**	**6.9968**	**2.9028**
	CatBoost	**0.9543**	**7.8954**	**3.1766**
	Decision Tree	0.8978	10.9053	4.4655
	SVR	0.6435	22.6876	13.5012
	ANN	0.7134	18.3762	11.0164

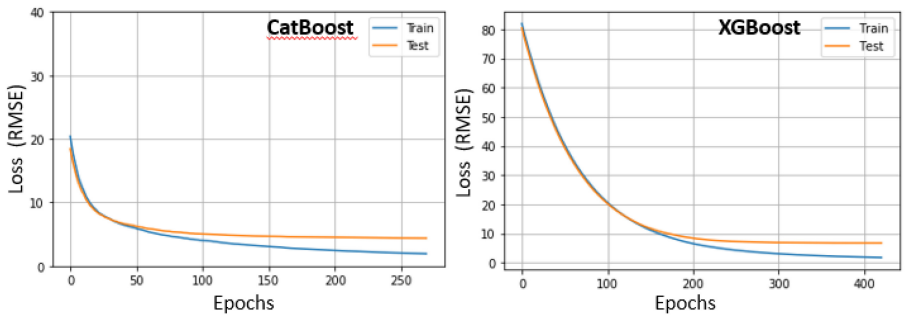

Fig. 6. Loss function plot for most precise tree-based methods.

XGBoost counterpart in terms of a monotonically decreasing trend. Loss function plots for the models assure that the two models are training well, since both the models exhibit smooth, monotonic descent, as seen in Figure 6. The test loss plot follows the training loss plot closely; hence, the model generalizes well on new data. This is in sharp contrast with ANN and other non-tree regressors (see Figure 4). The numerical difference between training and test loss is small enough to justify the better fitting capability of XGBoost compared to any other regressor, including CatBoost. Hence, we evaluate and compare the performance of these two tree-based models in Table 7 (shown for Tesla K20 GPU for brevity).

Table 6. XGBoost Results for All Three Architectures after Five Runs

Architecture	R^2 score	RMSE	MAE	MAPE
Kepler	0.901 ± 0.0083	7.4919 ± 0.0.3194	3.6740 ± 0.1126	4.5784 ± 0.1809
Maxwell	0.9418 ± 0.0.0056	4.4000 ± 0.1702	1.8780 ± 0.0634	2.9278 ± 0.0857
Volta	0.9553 ± 0.0056	7.7712 ± 0.4081	3.0521 ± 0.1365	3.9900 ± 0.0732

We further analyze the CatBoost and XGBoost model vs. ANN-based models on the basis of loss plots (Figures 4 and 6). ANN overfits very early, around 20 epochs. At 20 epochs, the validation R^2 score for inverse decay ANN is 0.6943 (train loss: 12.51, validation loss: 13.13), and for exponential decay ANN, it is 0.5839 (train loss: 11.77, validation loss: 12.97). It gets worse beyond 20 epochs with fluctuating loss curve (final train loss: 8.82, validation loss: 11.656). The CatBoost model starts overfitting around 60 epochs, with $R^2 = 0.839$ (train loss: 7.397, validation loss: 7.55), much higher than ANN. XGBoost clearly outperforms all the models. It achieves the final R^2 score of ANN (0.773) at 165 epochs (train loss: 8.802, validation loss: 8.86) without overfitting. When XGBoost starts to slightly overfit around 200 epochs, its validation R^2 score reaches 0.848 (train loss: 6.04, validation loss: 7.54), which is significantly higher than the ANN model's final R^2 score. Among tree-based techniques, the Decision tree also overfits early, i.e., at depth = 4, with validation $R^2 =$ 0.641 (train loss: 11.79, validation loss: 14.21). Furthermore, please notice that the loss is oscillating and does not decay smoothly. It also saturates after a certain number of epochs (50 for inverse decay and 140 for exponential decay), unlike in Figure 6, where tree-based methods demonstrate smooth, monotonically decaying loss. We ran the XGBoost model for Tesla V100 five times to test its robustness, and the observed results are presented in Table 6. This proves that our model is performing well, not by accident, and the prediction accuracy is consistent irrespective of changes in testing and training data.

We relate the response, i.e., the power consumed by the CUDA kernel and program features and understand its features contribution using feature importance. As seen in Table 7, we can conclude that *inst_issue_cycles*, *avg_comp_lat*, *reg_thread*, *glob_inst_kernel*, and *glob_store_sm* are the *top five* features, having high feature importance in both the models. Since we have used Shapley during the feature selection process, described in Section 4, these features are marked important by Shapley as well.

We now consider these important features that influence the response and aim to provide an explanation for the variation observed in the response variable corresponding to changes in the predictor variables, one at a time. It is also assumed that the important features have little pairwise interaction between them. In the nonlinear model fitted by XGBoost, we froze values of all (important) features except for one feature under consideration. As the values of that feature are varied and other features are kept constant, the change in the response variable is plotted. This will measure the impact of each crucial feature against predicted power in Figure 7. As expected, these top-ranked features are indeed influential in explaining the variations in the response variable. Note that since there is no closed-form regression model available, this is intuitively a reasonable check of the behavior of the response variable against the top-ranked predictor variables.

6.1 Analyzing Influential Features from Architecture Viewpoint

The feature importance analysis found that *inst_issue_cycle* contributes the most to power consumption across all three architectures. Since the FDS unit is employed in issuing instructions, inst_issue_cycle reasonably represents the number of times FDS is activated. Hong et al. [26] noted that the FDS unit consumes higher power than other components, because all instructions utilize this unit. Thus, our finding that this feature is the most significant feature certainly conforms to

Table 7. XGBoost vs. CatBoost Evaluation

Evaluation Metric		**XGBoost**	**CatBoost**
R^2 **score**		0.9101	0.9067
RMSE		7.1592	7.3001
MAE		3.4816	3.8308
Smaller dataset	**Memory consumption**	323.23 MiB	241.52 MiB
Larger dataset		162.68 MiB	214.86 MiB
Smaller dataset	**Execution time**	3.29 s	10.3 s
Larger dataset		6.4 s	11.8 s
Feature Importance Ranking (Consistent across all the three GPU architectures)		1. glob_store_sm 2. inst_issue_cycles 3. avg_comp_lat 4. misc_inst_kernel 5. glob_inst_kernel 6. reg_thread 7. comp_inst_kernel 8. avg_glob_lat 9. cache_penalty 10. avg_shar_lat 11. occupancy 12. shmem_block 13. branch 14. glob_load_sm 15. block_size	1. avg_comp_lat 2. inst_issue_cycles 3. reg_thread 4. glob_inst_kernel 5. comp_inst_kernel 6. avg_glob_lat 7. misc_inst_kernel 8. glob_store_sm 9. branch 10. glob_load_sm 11. cache_penalty 12. occupancy 13. avg_shar_lat 14. block_size 15. shmem_block

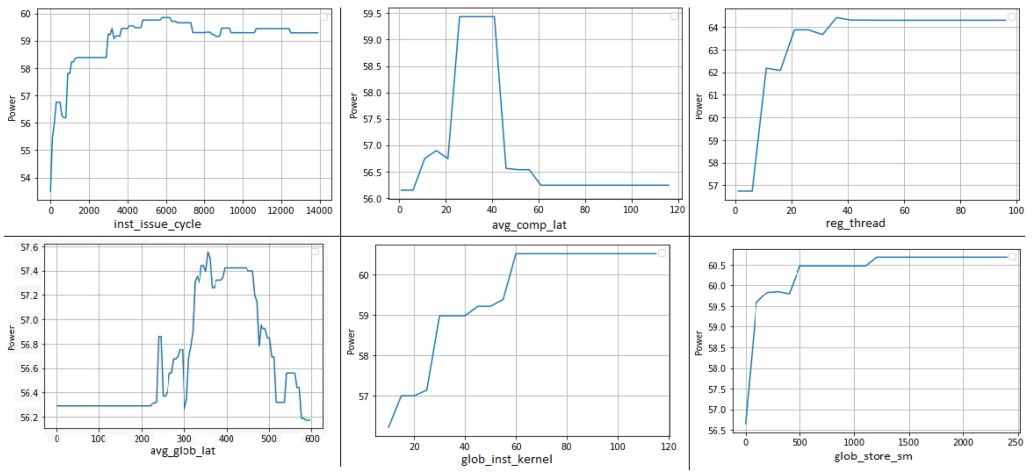

Fig. 7. Plots for crucial program features dependency on the power consumption.

the findings [26]. Since it is difficult for developers to control the number of instructions issued for a particular task, a plausible option would be to focus on writing efficient code with a lesser number of instructions to do the task.

The feature *avg_comp_lat* is the average latency of all computing instructions executed, including floating-point instructions such as `maddf`, `divf`. It is observed from Figure 7, *avg_comp_lat*

impacts power prediction when its value exceeds 10 cycles. This is the case when an application constitutes floating-point instruction, since the latency of other integer computing instructions (add, sub) is significantly lower (Section IV in the supplementary file). Our results *differ with* that of Hong et al. [26]'s finding, which concluded that there is no considerable difference between instruction and floating-point unit's power consumption.

The next important feature denotes the number of registers per thread (*reg_thread*). This is in concurrence with the findings reported by Greener [28]. As register file size increases, the leakage power increases; hence, limited use of registers is advisable for building power-efficient applications.

The number of global memory instructions per kernel (*glob_inst_kernel*) and the average global memory latency (*avg_glob_lat*) also contribute to power prediction–particularly global store instruction (*glob_store_sm*), which follows *glob_inst_kernel*. Nagasaka et al. [43] concluded that the acquisition of global memory access counts is crucial for accurate GPU power modeling. According to runtime hardware unit analysis by Hong et al. [26], global memory instructions consume a significant amount of power, much more than floating-point benchmarks. This supports our finding that global memory contributes significantly to power consumption. To reduce the increase in power consumption due to global memory, one can replace global memory instructions by using shared memory as frequently as possible. Observing the difference between the feature importance of *glob_store_sm* over *glob_load_sm* suggests that writing to the global memory is more power-consuming than reading from it.

The *block_size* and *shmem_block* are the least important features consistently across all the feature importance techniques employed. Observing feature importance of features related to shared memory (*shmem_block, avg_shar_lat*) suggests that its utilization has the most negligible impact on power prediction compared to all other instruction types. With respect to *block_size*, we can conclude that the *block_size* is not as crucial as long as total threads ($grid\_size \times block\_size$) launched is higher, which is used to compute *inst_issue_cycle*.

6.2 Analysing Prediction Model Using Parallel Dwarfs

Parallel dwarfs represent the communication and computation pattern of a parallel application [7]. To further analyze the predicted power results of the model built, we choose a set of benchmarks for each type of dwarf under study. We ran these benchmarks for multiple launch configurations and predicted their power consumption using the machine learning models trained. The observed MAE for each dwarf for the Tesla K20 machine is presented in Table 8.

Best cases: It is observed that the proposed model works best for predicting the power of benchmarks belonging to dense linear algebra, dynamic programming, and unstructured grid dwarfs. Dense linear algebra benchmarks are compute-bound applications, whereas dynamic programming and unstructured grid are memory latency bound. Hence we claim that our model works for both compute and memory-bound kernels.

Worst cases: We observe that benchmarks belonging to Graph Traversal, Branch and Bound, and Structured Grid dwarfs have higher MAE. The branch and bound benchmark's performance limit is problem specific. Structured grid benchmarks have high spatial locality. Benchmarks belonging to graph traversal dwarf include indirect table lookup and less computation. We conclude that our model may not produce precise results for benchmarks having these characteristics.

7 CONCLUSION

This article presents a GPU power prediction model based on program analysis employing machine learning algorithms. We utilize hardware details to generate the features considered in this study without running the application. Machine learning models considered in this study include

Table 8. Mean Absolute Error across Parallel Dwarfs

Model	Branch and Bound	Graph Traversal	Dense Linear Algebra	Structured Grid	Unstructured Grid	Dynamic Programming
Random Forest	11.31	13.17	4.45	14.31	0.33	8.33
Extra Trees	16.17	19.06	3.66	13.52	0.18	0.3
Gradient Boosting	19.51	14.52	4.6	25.74	0.43	0.05
XGBoost	21.14	14.85	3.91	24.03	0.3	0.96
CatBoost	14.42	23.4	7.22	6.29	3.28	2.65
SVR	29.34	25.86	5.89	3.69	1.37	1.96
ANN	15.91	32.91	4.74	4.8	5.96	10.04

Decision Tree, Random Forest Regressor, ExtraTrees Regressor, Gradient Boosting Regressor, XGBoost Regressor, CatBoost Regressor, SVR, and ANN. We tested these models for our hypothesis across three GPU machines: Tesla K20, Tesla M60, and Tesla V100. We observed that tree-based regression methods produce the most accurate power prediction model than SVR and ANN results. We observed that *inst_issue_cycle* is the most crucial feature across models, since it represents the FDS unit that is utilized by all instructions; hence, it is a constant contributor irrespective of other benchmark parameters. Benchmarks with floating-point instructions and global memory instructions also consume a significant amount of power; therefore, their usage has to be limited to build more energy-efficient applications. The use of registers in an application is also crucial and needs to be minimally utilized for energy efficiency. Thus, the final set of "influential features" obtained through a "data-driven" approach explains the variations in the response (power) well and is consistent with the domain expertise. This study has given rise to further GPU power modeling field investigations. We can utilize this study to predict the energy consumption of GPU. One can also dig deeper to understand how a program can be refactored using the presented observations in this work. This refactoring should be effective in such a way that it consumes lower power without compromising performance. Future work is also to analyze and improve prediction for benchmarks belonging to Graph Traversal, Branch and Bound, and Structured Grid dwarfs. In this work, we did not explore how cache performance characteristics can be included within the realm of static analysis. We aim to study these characteristics and their contributions to power consumption in the future.

REFERENCES

[1] Muhammad Waseem Ahmad, Monjur Mourshed, and Yacine Rezgui. 2017. Trees vs neurons: Comparison between random forest and ann for high-resolution prediction of building energy consumption. *Energy Build.* 147, 7 (2017), 77–89.

[2] Gargi Alavani, Jineet Desai, and Santonu Sarkar. 2020. An approach to estimate power consumption of a CUDA kernel. In *Proceedings of the IEEE International Conference on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom'20)*. IEEE, Exeter, 984–991.

[3] Gargi Alavani and Santonu Sarkar. 2022. Performance modeling of graphics processing unit application using static and dynamic analysis. *Concurr. Comput.: Pract. Exp.* 34, 3 (2022), e6602.

[4] Gargi Alavani and Santonu Sarkar. 2023. Inspect-GPU: A software to evaluate performance characteristics of CUDA kernels using microbenchmarks and regression models. In *Proceedings of the International Conference on Software Technologies*. SCITEPRESS, Rome.

[5] Gargi Alavani and Santonu Sarkar. 2023. Prediction of performance and power consumption of GPGPU applications. arXiv:2305.01886 [cs.DC]. Retrieved from https://arxiv.org/abs/2305.01886.

[6] Abdulaziz Alnori and Karim Djemame. 2018. A holistic resource management for graphics processing units in cloud computing. *Electr. Not. Theor. Comput. Sci.* 340 (2018), 3–22.

[7] Krste Asanovic, Rastislav Bodik, et al. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (2009), 56–67.

[8] Suryoday Basak, Saibal Kar, Snehanshu Saha, Luckyson Khaidem, and Sudeepa Roy Dey. 2019. Predicting the direction of stock market prices using tree-based classifiers. *N. Am. J. Econ. Financ.* 47, 1 (2019), 552–567.
[9] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. 2022. Deep neural networks and tabular data: A survey. In *IEEE Trans. Neural Netw. Learn. Syst.* IEEE, 1–21.
[10] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. 2016. Understanding gpu power: A survey of profiling, modeling, and simulation methods. *Comput. Surv.* 49 (2016), 41:1–41:27.
[11] M. Chadha, A. Srivastava, and S. Sarkar. 2016. Unified power and energy measurement API for HPC co-processors. In *Proceedings of the International Performance Computing and Communications Conference (IPCCC'16)*. IEEE, Las Vegas, NV, 1–8.
[12] 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 *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, Austin, TX, 44–54.
[13] J. Chen, B. Li, Y. Zhang, L. Peng, and J. Peir. 2011. Tree structured analysis on GPU power study. In *Proceedings of the International Conference on Computer Design (ICCD)*. IEEE, Amherst, MA, 57–64.
[14] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 785–794.
[15] Wu chun Feng and Kirk Cameron. 2007. The green500 list: Encouraging sustainable supercomputing. *Computer* 40, 12 (2007), 50–55.
[16] Jared Coplin and Martin Burtscher. 2015. Effects of source-code optimizations on GPU performance and energy consumption. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. ACM, New York, NY, 48–58.
[17] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. 1996. Support vector regression machines. In *Proceedings of the 9th International Conference on Neural Information Processing Systems (NIPS'96)*. MIT Press, Cambridge, MA, 155–161.
[18] Kaijie Fan, Biagio Cosenza, and Ben Juurlink. 2019. Predictable GPUs frequency scaling for energy and performance. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP'19)*. Association for Computing Machinery, New York, NY, Article 52, 10 pages.
[19] Jerome H. Friedman. 2000. Greedy function approximation: A gradient boosting machine. *Ann. Stat.* 29 (2000), 1189–1232.
[20] Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. 2013. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP'13)*. IEEE, 826–833.
[21] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Mach. Learn.* 63 (2006), 3–42.
[22] Anabel Gómez-Ríos, Julián Luengo, and Francisco Herrera. 2017. A study on the noise label influence in boosting algorithms: AdaBoost, GBM and XGBoost. In *Hybrid Artificial Intelligent Systems*, Francisco Javier Martínez de Pisón, Rubén Urraca, Héctor Quintián, and Emilio Corchado (Eds.). Springer International Publishing, Cham, 268–280.
[23] Joao Guerreiro, Aleksandar Ilic, Nuno Roma, and Pedro Tomas. 2018. GPGPU power modeling for multi-domain voltage-frequency scaling. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, Vienna, 789–800.
[24] M. S. Hecht and J. D. Ullman. 1974. Characterizations of reducible flow graphs. *J. ACM* 21 (1974), 367–375.
[25] Tim Hill, Leorey Marquez, Marcus O'Connor, and William Remus. 1994. Artificial neural network models for forecasting and decision making. *Int. J. Forecast.* 10 (1994), 5–15.
[26] Sunpyo Hong and Hyesoon Kim. 2010. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, 280–289.
[27] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. 2016. Improving GPU performance through resource sharing. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*. Association for Computing Machinery, New York, NY, 203–214.
[28] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. 2018. GREENER: A tool for improving energy efficiency of register files (2018). arXiv:1709.04697 [cs.AR].
[29] Wenhao Jia, Elba Garza, Kelly A. Shaw, and Margaret Martonosi. 2015. GPU performance and power tuning using regression trees. *ACM Trans. Arch. Code Optim.* 12 (2015), 1–26.
[30] Krishna Kandalla, Emilio P. Mancini, Sayantan Sur, and Dhabaleswar K. Panda. 2010. Designing power-aware collective communication algorithms for infiniband clusters. In *Proceedings of the 39th International Conference on Parallel Processing*. IEEE, 218–227.
[31] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A power modeling framework for modern GPUs. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)*. ACM, New York, NY, 738–753.

[32] A. Karki et al. 2019. Tango: A deep neural network benchmark suite for various accelerators. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 137–138.
[33] Vijay Anand Korthikanti and Gul Agha. 2010. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. Association for Computing Machinery, New York, NY, 157–165.
[34] J. Lemeire, J. G. Cornelis, and L. Segers. 2016. Microbenchmarks for GPU characteristics: The occupancy roofline and the pipeline model. In *Proceedings of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'16)*. IEEE, 456–463.
[35] Yun Liang, Muhammad Teguh Satria, Kyle Rupnow, and Deming Chen. 2016. An accurate GPU performance model for effective control flow divergence optimization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 7 (2016), 1165–1178.
[36] Jieun Lim, Nagesh B. Lakshminarayana, Hyesoon Kim, William Song, Sudhakar Yalamanchili, and Wonyong Sung. 2014. Power modeling for GPU architecture using McPAT. *ACM Trans. Des. Autom. Electr. Syst.* 19, 3 (2014), 1–24.
[37] Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez-Mesa, and Ben Juurlink. 2013. How a single chip causes massive power bills. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. IEEE, 97–106.
[38] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., Red Hook, NY, 4768–4777.
[39] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Wang Xiaorui. 2012. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Proceedings of the 41st International Conference on Parallel Processing (ICPP'12)*. IEEE, New York, NY, 48–57.
[40] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. 2009. Statistical power consumption analysis and modeling for GPU-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower'09)*, Vol. 1. ACM.
[41] Xiaohan Ma, Marion Rincon, and Zhigang Deng. 2011. Improving energy efficiency of GPU based general-purpose scientific computing through automated selection of near optimal configurations. Technical Report. https://uh.edu/nsm/_docs/cosc/technical-reports/2011/11_08.pdf.
[42] Diksha Moolchandani, Anshul Kumar, and Smruti R. Sarangi. 2022. Performance and power prediction for concurrent execution on GPUs. *ACM Trans. Archit. Code Optim.* 19, 3 (May 2022), 27 pages. https://doi.org/10.1145/3522712
[43] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. 2010. Statistical power modeling of GPU kernels using performance counters. In *Proceedings of the International Green Computing Conference (GREENCOMP'10)*. IEEE, 115–122.
[44] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: Unbiased boosting with categorical features. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems. Advances in Neural Information Processing Systems*, 6639–6649.
[45] S. Sharma, Chung-Hsing Hsu, and Wu chun Feng. 2006. Making a case for a green500 list. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 8 pp.
[46] J. W. Sheaffer, D. Luebke, and K. Skadron. 2004. A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*. Association for Computing Machinery, New York, NY, 85–94.
[47] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular data: Deep learning is not all you need. *Inf. Fus.* 81, 5 (2022), 84–90.
[48] Karan Singh, Major Bhadauria, and Sally A. McKee. 2009. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Comput. Arch. News* 37, 5 (2009), 46–55.
[49] Yan Solihin. 2015. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC.
[50] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'13)*. IEEE, 673–686.
[51] R. Tagliaferri and M. Marinaro. 1997. Neural nets, WIRN Vietri-96. In *Proceedings of the 8th Italian Workshop on Neural Nets, Vietri Sul Mare*. Springer-Verlag, Berlin, Heidelberg, XI, 346.
[52] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. IEEE, 235–246.
[53] Qi Zhao, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2013. POIGEM: A programming-oriented instruction level gpu energy model for CUDA program. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*. Springer, Berlin, 129–142.

Received 17 July 2022; revised 2 April 2023; accepted 24 May 2023