Predicting the energy consumption of CUDA kernels using SimGrid

Dorra Boughzala, Laurent Lefèvre, Anne-Cécile Orgerie

Predicting the Energy consumption of CUDA kernels using SimGrid

Dorra Boughzala*†, Laurent Lefèvre* and Anne-Cécile Orgerie†
*Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, Lyon, France - Email: {dorra.boughzala,laurent.lefevre}@inria.fr
†Univ. Rennes, Inria, CNRS, IRISA, Rennes, France - Email: anne-cecile.orgerie@irisa.fr

***Abstract*—Building a sustainable Exascale machine is a very promising target in High Performance Computing (HPC). To tackle the energy consumption challenge while continuing to provide tremendous performance, the HPC community have rapidly adopted GPU-based systems. Today, GPUs have became the most prevailing components in the massively parallel HPC landscape thanks to their high computational power and energy efficiency. Modeling the energy consumption of applications running on GPUs has gained a lot of attention for the last years. Alas, the HPC community lacks simple yet accurate simulators to predict the energy consumption of general purpose GPU applications. In this work, we address the prediction of the energy consumption of CUDA kernels via simulation. We propose in this paper a simple and lightweight energy model that we implemented using the open-source framework SimGrid. Our proposed model is validated across a diverse set of CUDA kernels and on two different NVIDIA GPUs (Tesla M2075 and Kepler K20Xm). As our modeling approach is not based on performance counters or detailed-architecture parameters, we believe that our model can be easily approved by users who take care of the energy consumption of their GPGPU applications.**

***Index Terms*—GPGPU computing, CUDA kernels, Energy modeling, Simulation.**

I. Introduction

Many technical reports on building an Exascale machine recognize the energy consumption as one of the major challenges to achieving those systems. Actually, to reach an Exascale performance, we should provide approximately 3-fold increase in energy efficiency [1]. Meanwhile, to address this challenge, extreme-scale computing systems such as Top500 [2] and Green500 [3] lists, that rank the world's fastest and most energy-efficient supercomputers, have combined multi-core CPUs with general purpose GPUs. Hence, CPU-GPU computing has become a mainstream trend in these systems: 6 systems from the top 10 of the last updated Green500 list adopt the CPU-GPU heterogeneous architecture [3].

For many years, GPUs have been dedicated only for graphics. However, since a decade now, they are continually evolving to support a wide range of massively parallel applications in scientific computing such as weather forecasts, simulations of molecular dynamics and real-time analytics. Today, GPUs are at heart of Deep Learning (DL) applications, accelerating breakthroughs in Artificial Intelligence. Even though, GPU-based systems have made tremendous progress in delivering high performance and energy efficency, these systems still draw great amounts of electrical power.

A lot of research efforts have been made to predict the performance and energy consumption of GPUs. Indeed, accurate predictions of performance and energy consumption of GPGPU applications could be very helpful for the study of energy-related policies and optimization techniques. For example, Hong et al. [4] proposed the first integrated power and performance (IPP) system that aims to predict the optimal number of cores needed to achieve the highest energy efficiency. Similar to other proposed GPU power simulators [5], [6], the IPP model is a product-specific model that requires expertise of the underlying architecture. Moreover, performance-counters models, very popular among the CPU power modeling community, are rapidly adopted in the research area of GPUs. Early works [10], [11] used linear-based approaches to characterize the relationship between performance counters and power. Recently, [12], [13] rely on more sophisticated non-linear approaches. However, counter-based models lack portability as some performance counters may be not present with existing or future architectures. Plus, those models lack flexibility as their number is increasing with every new GPU generation, making the selection a tedious task [9]. Therefore, the HPC community lacks simple yet accurate simulators to predict the energy consumption of CUDA kernels.

In this article, we address the problem of predicting the energy consumption of CUDA kernels through simulation. In particular, we propose a simple and lightweight energy model that relies only on the number of blocks as the independent variable. Based on the CUDA programming and execution model, we notice the strong correlation between the energy consumed and the number of blocks. Unlike many works that rely on performance counters or architectural parameters, our modeling approach is totally coarse-grained. For that, we believe that our model can be easily adopted by programmers who take care of the energy consumption of their GPGPU applications. Hence, we implement our model using the open-source simulation framework SimGrid [23]. Then, we validate it across a wide range of CUDA kernels retrieved from CUDA SDK samples [8] and the Rodinia benchmark suite [7] on two commercial Tesla NVIDIA GPUs: M2075 and K20Xm.

The remainder of this paper is organized as follows. Section II introduces an overview of the GPU architecture and CUDA execution model. Then, in Section III, we present our measurement methodology, our experimental setup and a description of CUDA kernels used for the study. Section IV presents our analytical study to model the performance and energy

consumption of CUDA kernels . In Section V, we introduce the SimGrid simulation toolkit and we explain how to implement and calibrate our proposed energy model . Last but not least, we discuss in Section VI results obtained via the comparison of simulation to measurements on real hardware. Finally, we conclude in Section VII.

II. Background

In this section, we introduce some fundamental information about the NVIDIA GPU architecture and its execution model.

A. NVIDIA GPU architecture

Each GPU has several streaming multiprocessors (SMs). Hundred to thousands of Stream Processors (SPs), known as Compute Unified Device Architecture (CUDA) cores, are organized into an SM, along with a few special function units (SFUs) and load/store units. CUDA cores execute the same instruction on different data, that is why we call them Single Instruction on Multiple data (SIMD) cores. For more recent architectures, NVIDIA started to scale the number of its CUDA cores by adding more SMs in the GPU card. This design has resulted in significant overall GPU performance improvements. The organization of SMs changes from a generation to another, either by adding new units like Double Precision (DP) units and Tensor cores, or by partitioning the SM into blocks to facilitate the scheduling process. From the memory hardware perspective, each SM includes on-chip register files used by all thread blocks and a (read/write) shared/L1 cache memory. GPUs also contain an L2 cache memory, which is accessible by all SMs. Finally, the DRAM can be accessed through several different abstractions in CUDA, including global memory (off-chip), and read-only texture memory (on-chip) and constant memory (on-chip). The access to the global memory by all threads is very expensive, this explains why GPUs have multiple levels of memory hierarchy.

B. GPU execution model

CUDA is both the platform and the programming model built by NVIDIA for developing applications on NVIDIA GPUs cards. The CUDA programming model is an extension to the C programming language, which makes it easy for programmers to port their applications to GPUs. The heart of CUDA performance and scalability lies in its execution model. CUDA exposes a high abstract view of the GPU parallel architecture in order to simplify the mapping of the parallelism within an application to the underlying hardware.
Therefore, CUDA proposes three key programming abstractions: threads, blocks and grids. Actually, when a kernel is launched, a grid of blocks is generated and threads are grouped into blocks, refer to Figure 1. A thread block is a set of threads that can cooperate among themselves through barrier synchronization and shared memory. Each thread block has a block ID within its grid. A thread within a thread block executes an instance of the kernel, and has a thread ID that can be calculated uniquely from the indexes of the block and the grid it belongs to. Further, instructions are issued

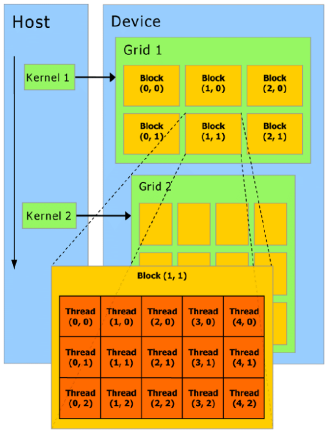

Fig. 1. Grids of blocks in the GPU programming model [18]

to groups of CUDA threads, called warps. A warp, a group of 32 consecutive threads, is the basic unit for scheduling work on the GPU. Indeed, to hide memory latencies, the GPU implements multi-level Thread Level Parallelism (TLP). Memory pending warps are replaced by other ready warps to maintain GPU resources busy. To handle all this, CUDA provides two-level of scheduling. Actually, the programmer has no direct control on the scheduling. Indeed, a *global* scheduling is assured by the *GigaThread* scheduler, which assigns one or more thread blocks to each SM. And a *local* scheduling is assured by the SM warp schedulers, which select every clock cycle active warps and dispatch them to execution units.

III. Experimental setup and Methodology

In this section, we explain our measurement methodology of the execution time, power and energy proposed in this study. Also, we introduce in detail our system configuration and evaluated CUDA kernels.

A. Measurement Methodology

In this work, we rely on CUDA events for accurate timing of the kernel execution time (T) and on the built-in GPU power sensors for power measurements. Indeed, we use the NVIDIA system management interface (NVSMI) [15] to monitor the power consumption (P) and other relevant metrics. NVSMI is based on the C-based NVIDIA Management Library (NVML), which is mainly dedicated to help users in the management and monitoring of their NVIDIA GPU devices [16]. Unlike some related works [19], [21] in which authors create their own tool based on NVML to monitor power consumption, we rely directly on NVSMI for its ease of use (command-line utility) and availability to everyone owning a GPU.

In our experiments, we collect the last measured power draw (P) for the entire board in Watts. This reading is accurate to within +/- 5 Watts according to [15]. Among other metrics, we collect the GPU utilization, memory utilization, the power/performance state, SM clock and temperature (if

available). All these information are logged into a file following a csv format for further processing. The sample period for collecting those metrics depends on the hardware [15]. Thus, we conducted a brief study to characterize the sampling rate and we found out that 50 milliseconds (20 Hz) is adequate to accurately monitor the power on both studied GPUs.

For the post-treatment of the log file, we extract only power draws when the performance state is equal to P0 (the highest state). Similar to our measuring approach, authors in [22] use the pstate value P0 to detect GPU activity. Actually, the simple launch of NVSMI would get the driver loaded and thus the GPU moving automatically to pstate P0. For that, they invoke every time NVSMI seconds before launching a program. While, we propose to enable the persistence mode on our GPU devices for accurate measurements. By activating this mode, the GPU is kept initialized even when no users are connected to it and the power/performance state is moved down to a certain level (usually P8). Indeed, we ensure by enabling the persistence mode that the power draw before launching NSVMI is stable. Finally, we obtain accurate measurements of the energy consumption (E) of a kernel running on a GPU by integrating its power (P) over its corresponding execution time (T).

B. System configurations

In this work, we rely on two different infrastructures for the availability of different generations of GPU on each. We run our experiments on the the Grid'5000[1] infrastructure, in particular on the orion cluster. The orion cluster is composed of 4 homogeneous nodes, each consisting of 2 Intel Xeon E5-2630 with 6 physical cores per CPU, 32 GiB of RAM and an NVIDIA Tesla M2075 GPU card. Our GPU is based on the Fermi architecture. It is connected to the rest of the system using a PCI-Express 2.0 x16 interface. We use Linux x86_64 Driver Version and CUDA runtime version 8.0 on a custom Debian GNU/Linux image.

Also, we run our experiments on the ENS Lyon infrastructure, mainly on the grunch cluster. The grunch cluster is composed of 2 Intel Xeon E5-2695 with 14 physical cores per CPU, and 2 NVIDIA Kepler K20Xm GPU cards (we count only on one card in this work). Our GPU is based on the Kepler architecture. It is connected to the rest of the system using a PCI-Express 3.0 x16 interface. We use Linux x86_64 Driver Version 418.74 and CUDA runtime version 10.0 on a custom Debian GNU/Linux image. A brief description of resource characteristics of Tesla M2075 and Kepler K20Xm is presented in Table I. We enabled the persistence mode and the Error Correcting Code (ECC) for both GPUs during our experiments.

TABLE I
TESLA M2075 AND KEPLER K20XM RESOURCES DESCRIPTION

Device	M2075	K20Xm
Architecture	Fermi	Kepler
Capability	2.0	3.5
#SMs	14	14
#CUDA cores/SM	32	192
Memory bus width	384-bit	384-bit
Registers/Block	32768	65536
GM size (GB)	5.301	5.701
Max #threads/block	1024	1024
Max #of threads/SM	1536	2048
Limit Power Draw (W)	225	235

C. CUDA kernels

For our study, we use two simple CUDA kernels and three GPGPU applications in order to cover a variety of application domains and different kernel features such as data access patterns and diverse computing units usage such as Single Precision (SP) cores, Double Precision (DP) units or Special Function Units (SFU). In the following, we explain in detail CUDA kernels' characteristics.

1) Simple CUDA Kernels: As a start, we rely on two *representative* kernels taken from the CUDA SDK samples as each represents a kernel type where: vectorAdd is a memory-bound kernel and matrixMul with shared memory is a compute-bound one. We modify their source code by adding a loop inside each kernel because the block execution time is very small (order of microseconds). For that, we use large loop sizes as shown in Table III. Concerning the vectorAdd launch configuration, we vary the vector sizes and test with three different block_sizes (1024,1,1), (512,1,1) and (256,1,1). While for matrixMul, we use block_sizes equal to (32,32,1) and (16,16,1) and we only vary the width of matrix B to ensure the launch of blocks with same amount of work.

2) Applications: We rely on CUDA kernels originated from both the Rodinia Benchmark suite [7] and the CUDA SDK samples as shown in Table II. Thus, we use three real-world applications: BlackScholes, Hotspot and Back Propagation. Indeed, BlackScholes is an option pricing application that implements the BlackScholes model for European options. Hotspot is a popular thermal simulation tool in physics simulation used for processor temperature estimation. Moreover, Back Propagation is a machine-learning algorithm used during the training process of a layered neural network. This application contains two phases: a forward phase and a backward phase, which corresponds each to a CUDA kernel. In order to have enough reliable power measurements, we modify our CUDA applications to execute the same kernel multiple times (refer to Table III). We present in the following kernel launch configurations for the different studied kernels: for BlackScholes, we use the block_size (128,1,1) and we only vary the number of options. For Hotspot, we vary the grid size and use the block_size (16,16,1). For Back Propagation, we vary the number of nodes in the input layer, and we use the block_size (16,16,1).

[1]https://www.grid5000.fr

TABLE II
OVERVIEW OF KERNELS' CHARACTERISTICS

Kernel	Type	Main Kernel Usage
Backprop_K1	compute-bound	Shared Memory, SP operations
Backprop_K2	memory-bound	Global Memory, DP operations
BlackScholes	compute-bound	Global Memory, SP, SFU operations
Hotspot	compute-bound	Shared Memory, DP, SP, SFU operations
VectorAdd	memory-bound	Global Memory, SP operations
MatrixMul	compute-bound	Shared Memory, SP operations

IV. MODELING THE PERFORMANCE AND ENERGY CONSUMPTION OF CUDA KERNELS: THE ANALYTICAL APPROACH

In the section, we present the feature selection process, the scheduling algorithm obtained through our empirical study, as well as the performance and energy models formulated into simple mathematical equations.

A. Feature Selection

We conduct several experiments in order to study the impact of different execution configuration parameters (number of blocks, number of threads per block) and the number of active SMs on performance, power and energy consumption of CUDA kernels. Our findings reveal that the number of blocks and number of threads per block are highly correlated to the performance and energy consumption for different types of applications. Taking into account this correlation and the fact that the number of block is a user-defined parameter and not an architecture parameter, we select the number of blocks (NB) as a key component in our modeling approach. Indeed, we assume that all blocks of a certain CUDA kernel have the same amount of work.

B. Blocks Scheduling

According to NVIDIA, each block can be scheduled on any of the available SMs *"in any order, concurrently or sequentially depending on available resources such as registers, shared memory or limited number of warps per SM"* [18]. Based on our observations on both NVIDIA devices, the CUDA runtime system maps blocks to SMs following a round-robin fashion. Even though details about blocks and/or warps scheduling are not publicly documented by NVIDIA, many works such as [31] claim as well that it follows a round-robin algorithm. Indeed, we capture clearly this aspect on micro-benchmarks (using *vectorAdd* for instance) where the work done by a single block is long enough.

In Figure 2 (a), the execution time of vectorAdd with block_size equal to 1024 follows a stepwise curve on both GPUs. Every time we have a number of blocks higher than the number of SMs on M2075, we wait until it finishes its execution on the 14 SMs first. This is mainly due to the fact that only one block of size 1024 can reside per SM. For K20Xm (see Figure 2 (b)), we observe that the runtime execution increases each time after the execution of 28 blocks. Actually, on this GPU the number of max resident blocks

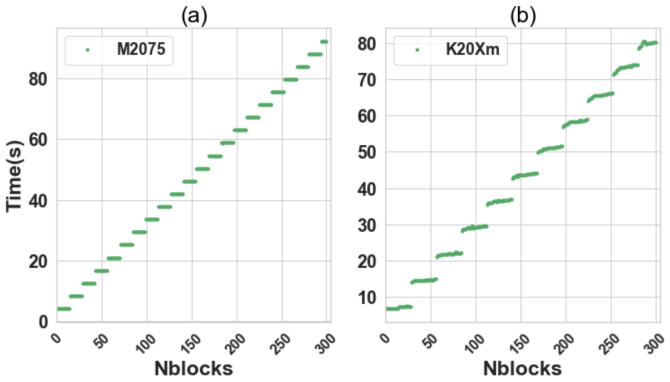

Fig. 2. Profiling the runtime of vectorAdd with TB=1024 on M2075 (a) and K20Xm (b)

of size 1024 per SM is equal to 2. The maximum amount of resident blocks per SM is calculated with the CUDA occupancy calculator [24]: an XLS file that allows us to compute the ratio of active warps/blocks to the maximum number of warps/blocks supported on a SM for a given CUDA kernel. We highlight here that the scheduling algorithm is crucial for our simulation approach.

C. Performance and Energy models

The execution time of running a CUDA kernel can be calculated using the equation 1 where the slope of the regression line is a and b is the intercept:

$$T(NB) = a \times NB + b \tag{1}$$

The energy consumption E of a program can be written as the sum of its dynamic energy E_d and static energy E_s (equation 2). Where dynamic energy represents the variable power consumed by the active use of components during the execution of a program. While static energy represents the fixed power consumed due to the loss of energy from the capacitors during the same interval of time. On this basis, we propose to predict the energy consumption of a CUDA kernels with (NB) blocks following those equations (equation 3). Where E_b denotes the dynamic energy of one block, P_s is defined as the static power and T(NB) is the execution time obtained with the equation 1.

$$E(NB) = E_d(NB) + E_s(NB) \tag{2}$$

$$E(NB) = E_b \times NB + P_s \times T(NB) \tag{3}$$

To predict the energy consumption of blocks (NB) following the equation 3: First, we make the assumption that the static power P_s is equal to the idle power. We obtain this value by monitoring the GPU when it is ON and no application is running. Second, we predict the dynamic energy of one block (E_b). For that, we subtract the static power from measured power values. Then we measure the dynamic energy by multiplying the sum of dynamic power

values by the sampling rate. Finally, we apply a Simple linear Regression (SLR) on the dynamic energy profile. We consider the slope as the dynamic energy of one block.

We can extend our models further by taking into consideration the round-robin scheduling of blocks and the number of SMs (nbSMs). We can re-write them as shown in equations 4 and 6, where T_r and E_r represent respectively the execution time and the energy consumption of executing a round. We predict the runtime of a round T_r by relying on the slope of equation 1. Actually, we make the abstraction that during a round the GPU executes a number of blocks which is equal to the number of SMs in order to simplify our models because we see no need to add more parameters. In reality, this is not always the case, especially with small block_sizes where more than one block can reside by SM.

$$T(NB) = T_r \times ceil(\frac{NB}{nbSMs}) \tag{4}$$

$$T_r = a \times nbSMs \tag{5}$$

$$E(NB) = E_r \times ceil(\frac{NB}{nbSMs}) \tag{6}$$

$$E_r = E_b \times nbSMs + P_s \times T_r \tag{7}$$

We rely on predicted values of E_b and P_s to calculate the energy consumption of one round (see equation 7). By using the predicted energy consumption of executing a round and its execution time, we deduct the average power consumption of a round P_r.

V. Predicting the performance and energy consumption of CUDA kernels: the SimGrid approach

Simulation is a very pervasive approach to study the behavior of GPGPU applications in terms of performance and energy consumption. In this section, we present the SimGrid framework then we describe the simulation process from building to calibration for accurate energy predictions.

A. The SimGrid Framework

SimGrid is a versatile open-source framework for developing simulators and studying the behavior of distributed computing systems such as grids, clouds, or peer-to-peer systems [23]. It provides models and APIs that the user can rely on to create his/her own simulator. As SimGrid's simulation models, we can find the multicore machine model, the TCP model and the churn model for P2P networks, etc. Indeed, it yields the *S4U interface* mainly dedicated to describe and simulate abstract algorithms as well as the *SMPI interface* dedicated to the simulation of existing MPI applications. Furthermore, users can extend SimGrid without modifying it via the plugin mechanism. Some plugins are distributed by default with SimGrid such as the *Energy* plugin.

B. How to build and calibrate the simulator?

For our work, we rely on the *S4U interface* to describe our algorithm and on the plugin *Energy* to simulate the energy consumption of our CUDA kernels. First, we need to activate the *Energy* plugin by calling *sg_host_energy_plugin_init()*. As for a classical SimGrid simulation, we need to describe our application. We implemented our algorithm based on the round-robin scheduling. For that, we should provide three input parameters:

1) The number of SMs
2) The number of Blocks
3) The number of flops to be executed at each round.

More specifically, the value of flops in a round is defined as the number of flops executed when the number of blocks is equal to the number of SMs. For example, for our GPU devices M2075 and K20Xm, it is the number of flops when running a round of 14 blocks. Second, we need to describe our platform file as shown in Figure 3. We define our GPU as a monocore host with a certain capacity of computation, referred as *Speed*. We have made this abstraction to avoid the default execution model of mutlicore machines in SimGrid, where cores support multithreading. Because calibration is the key for good predictions, we highlight how we obtain the speed value. Speed is only the flops/s of executing a round. For that, we need to provide the number of flops used for running a round and the time it takes T_r. The number of flops for each kernel could be calculated or monitored (via the NVIDIA profiler), depending on the complexity of the algorithm.

According to our observations, the linear regression model requires only three real measurements: two small blocks and a medium one or inversely for the modeling process. Last but not least, we represent an example with two scenarios: (1) GPU is ON and (2) GPU is OFF, of our energy model parameters for kernel vectorAdd with Tb=1024 on M2075 (refer to Figure 3):

1) P_s=29.4W: when the GPU is running but it has no work to do.
2) P_r=153.65W: when the number of blocks is equal to the number of SMs.
3) P_{off}=10W: when the host is off (cost of the network card on Grid'5000 waiting to wake up in mode *wake on LAN*)

Indeed, the idle power consumption of M2075 and K20Xm is respectively equal to 29.4W and 19.18W.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
  <zone  id="AS0" routing="Full">
     <host id="GPU" speed="3413f" core="1" >
       <prop id="wattage_per_state" value="29.4:153.65" />
       <prop id="wattage_off" value="10" />
     </host>
</zone>
</platform>
```

Fig. 3. Platform file calibrated for vectorAdd with TB=1024 on M2075

TABLE III
OVERVIEW OF DATASETS RANGES USED FOR VALIDATION

Kernel	Validation Datasets	Configurations
Backprop	*layersize* in range(25600,10^6)	*loop_size*=10^4
BlackScholes	*nboptions* in range(10^5,10^7)	*loop_size*=10^4
Hotspot	*gridsize* in range(200,1024)	*pyramid_height*=2, *sim_time*=20000, *temp_file*=temp_1024, *power_file*= power_1024
VectorAdd	*vectorsize* in range(1024,10^7)	*loop_size*=10^7
MatrixMul	*MatrixBx* in range(132,30000)	*loop_size*=10^6

VI. EVALUATION

In this section, we present our evaluation methodology: data sets, kernels configuration and evaluation metrics. Then we discuss the simulation results compared to real measurements.

A. Evaluation methodology

For validating our models through simulation, we choose our data sets randomly with the NumPY Library dedicated for scientific computing with Python. Indeed, we were very careful that values from datasets used for validation are different from ones used for modeling. As shown in Table III, we present datasets ranges and other configurations. For evaluation metrics, we rely on the Worst Relative Error (WRE), the Best Relative Error (BRE) and the Average Relative Error (ARE), obtained with the equation 8, where y_i and $\hat{y_i}$ are respectively the measured energy and the simulated energy.

$$ARE = 100 * \Sigma_{i=1}^{n} \frac{(\hat{y_i} - y_i)}{y_i} \quad (8)$$

B. Results

The evaluation on the Tesla M2075 GPU exhibits good predictions in terms of average relative error (ARE) for all evaluated kernels as shown in Table IV. For different types of kernels, we have accurate predictions and more particularly on compute-bound kernels such as matrixMul and BlackScholes. This can be explained by the simple fact that the GPU succeeds in hiding latencies by executing computations and especially when we have many blocks at hand. Thus, the correlation between runtime and the number of blocks is perfectly linear in those cases. For memory-bound kernels such as vectorAdd, we can notice that the model struggles a bit especially with small block_sizes. Indeed, when running blocks with block_size equal to 1024 on M2075, we are forcing the GPU to run one block per SM. This configuration likely ensures a fixed latency unlike with smaller blocks_sizes, where we can have more than one block per SM. Hence, the memory bandwidth can be saturated for a certain number of blocks, making the curve not smoothly linear. Plots in Figure 4 illustrate the accuracy of our predictions where we can observe the normalized measured vs. simulated runtime and energy for three different CUDA kernels on Tesla M2075.

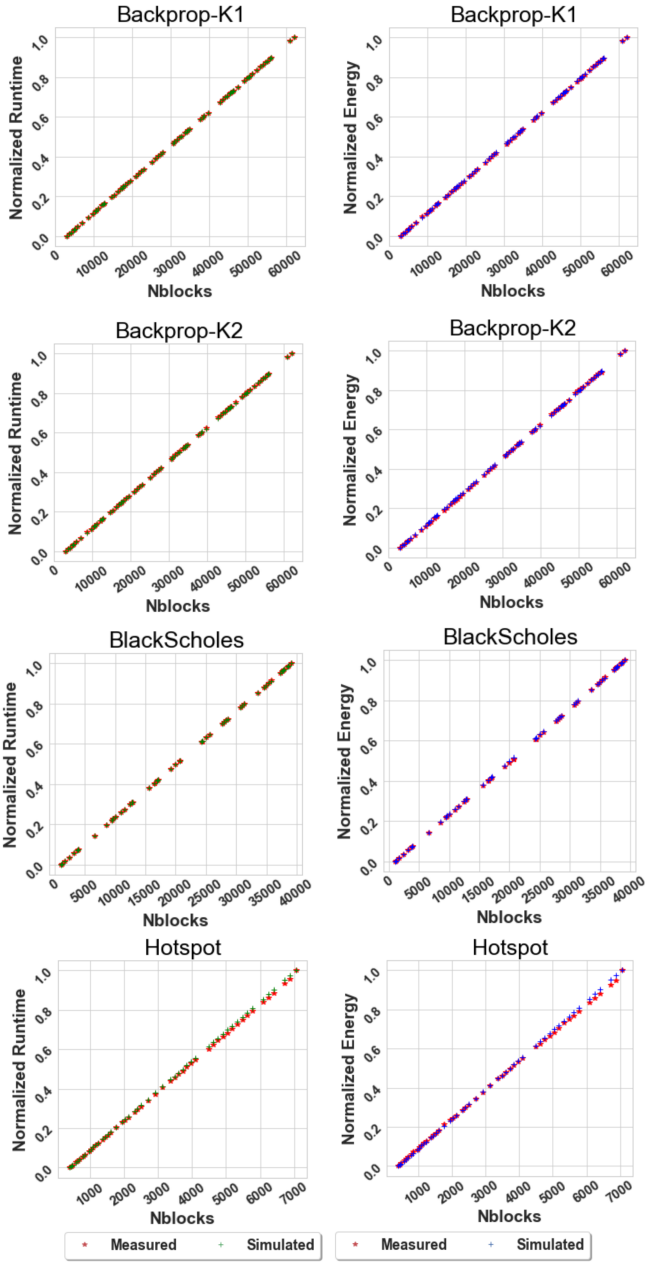

Fig. 4. Simulated vs. measured normalized Runtime and Energy of Backprop kernel 1 and 2, BlackScholes and Hotspot on M2075

On the other hand, the evaluation on the Kepler K20Xm GPU shows also promising results, even though we have higher worst relative errors (WRE) comparing to results on Tesla M2075. After analyzing our data, we find out that those values are mainly present for the first blocks in the evaluation dataset. The question that may arise here is: why we have this more pronounced on our K20Xm than on M2075 ? Actually, the Kepler device for some kernels such as Hotspot is capable to optimize for runtime and energy with a higher number of blocks. We estimate that would not be very possible on M2075, as the Kepler generation offers significant increase in double precision performance comparing to Fermi generation

TABLE IV
EVALUATION OF CUDA KERNELS SIMULATIONS ON M2075

Kernel	Time (%)			Energy (%)		
	WRE	BRE	ARE	WRE	BRE	ARE
Backprop_K1	1.33	0.00938	0.457	2.16	0.0313	0.836
Backprop_K2	3.02	0.27	2.42	4.29	0.00085	1.23
BlackScholes	6.64	0.215	2.2	5.80	0.18	1.74
Hotspot	4.79	0.0785	1.3	5.39	0.0220	1.52
vectorAdd_1024	3.60	0.877	3.41	7.62	0.083	1.47
vectorAdd_512	11.84	0.3	4.83	16.82	0.193	5.19
vectorAdd_256	20.76	0.0693	6.69	17.47	0.0164	6.86
matrixMul_1024	4.12	0.0145	0.94	10.83	0.0251	1.65
matrixMul_256	9.62	0.240	2.40	7.79	0.007	2.06

TABLE V
EVALUATION OF CUDA KERNELS SIMULATIONS ON K20XM

Kernel	Time (%)			Energy (%)		
	WRE	BRE	ARE	WRE	BRE	ARE
Backprop_K1	8.88	3.35	4.10	5.85	3.77	4.16
Backprop_K2	2.21	0.249	0.518	3.79	1.39	3.10
BlackScholes	18.04	0.292	2.51	9.56	0.14	1.16
Hotspot	16.46	0.015	3.52	7.81	0.202	2.83
vectorAdd_1024	24.52	0.009	2.26	24.40	0.05	3.06
vectorAdd_512	15.20	1.0	3.67	17.98	0.169	2.97
vectorAdd_256	13.73	0.02	2.36	14.02	0.04	3.77
matrixMul_1024	19.72	0.917	6.54	17.73	0.864	8.33
matrixMul_256	13.69	0.858	5.42	15.52	0.086	5.95

and provides higher number of cores and special function units (SFUs). Those units are indeed used in the Hotspot kernel as shown in Table II. We recommend to use at least one large value for the number of blocks to model on K20Xm, in order to capture the optimization done by the device.

C. Limitations of our Energy Model

As shown in Section V, results are very promising for different types of kernels. However, the model simplicity is not without any cost. First, though our model is mainly inspired from the CUDA execution model, we can not explore all applications or kernels written in CUDA. Here, we point to applications that rely on subroutines such as the ones available within the CUDA Basic Linear Algebra Subroutine library (cuBLAS). With this programming approach, the developer does not know beforehand the number of blocks or sometimes the number of kernels that would be launched. Instead, librairies and/or compiler take care of everything. Second, our energy model has shown limitations at predicting the energy consumption of kernels with a small number of blocks. We believe that this limitation is not very critical as the heart core of GPGPU programming is to dispatch a huge amount of work to the GPU. Last but not least, we need to provide calibration values for each CUDA kernel. Finally, unlike low-level models, our high-level model does not point out directly bottlenecks.

VII. RELATED WORK

Since the emergence of GPU-based systems, studies of GPU performance as well as power modeling and simulation have received a lot of attention among the research community. In this section, we focus only on related works tackling GPU power modeling and simulation.

A. GPU power models

Indeed, we can find in the literature performance-counter models [10], [11] that use linear-based statistics to estimate GPU power consumption. For instance, authors in [10] adopt the Support Vector Regression (SVR) model, as it outperformed the traditional square based linear regression (SLR). However, their model lacks global memory access counts, as the profiling of this metric was not available with the tested GPU. We can find as well performance-counter models that use non-linear statistical methods such as sophisticated random forest [12], Artificial Neural Network (ANN)[13] and regression trees [14]. Comparisons between linear-based and non-linear based methods conducted in [13] shows that ANN is more accurate than Multiple Linear regression (MLR). Nonetheless, according to [9], neural network accuracy is highly correlated to *"many configurations such as the number of layers, activation functions, and optimization options, which can be costly to test and configure"*. More recently, the roofline model orginally proposed by [25], was extended to better suit the GPU architecture and its related characteristics such as in [26] and [28]. Despite the fact that roofline models offer simple visual representations, they mainly rely on a representative number of parameters or on performance counters such as in [27].

B. GPU simulators

Sheaffer et al. [29] was the first work to propose Qsilver, a functional graphics performance, power and temperature simulator. Although, their work can not be immediately applicable to modern general purpose GPUs, it has served as basis for most of the literature on energy and power simulation for GPUs. Their approach is based on predicting hardware events (from a model of the GPU architecture and the application code) which are used as input to the power model. More recently, two open-source GPU power models, namely GPUWattch [6] and GPUSimPow [5] are proposed, which adopt the same methodology. Both models are built on and available with GPGPU-Sim [30], an open source framework that simulates every detail of the GPU. Moreover, they rely on the McPAT tool [33] to model the micro-architecture elements of their studied GPUs. However, they differ in the way GPU architecture and power are modeled, for example only in the work [5] that the Warp Control Unit (WCU) was modeled. In addition, Lim et al. [32] proposed a GPU power model using McPAT to model each sub-component of an NVIDIA GTX580 GPU. Despite the high accuracy of those cycle-level simulators, most of them demand an in-depth knowledge of the architecture to tune them for other GPU architectures. Thus, it could be a real burden of a GPU programmer who simply wants to predict the energy consumption of his kernel.

VIII. CONCLUSIONS

In this work, we simulate the energy consumption of CUDA kernels using the open-source framework SimGrid.

Our approach is based on the core of the CUDA execution model, where the kernel work is divided into blocks and dispatched to Streaming Multiprocessors (SMs). Our model predicts the energy consumption of a CUDA kernel using only one single variable: the number of blocks. In other words, it far exceeds the limitations of counter-based and architecture-detailed models. Also, we expect our model to be portable across generations as we rely on the number of blocks, which is a highly generic parameter.

The accuracy of the energy model is highly dependent on the accuracy of the runtime model. For that, we take care of accurately building and calibrating our simulator. We validate our model using six different CUDA kernels exhibiting various characteristics on two commercial Tesla NVIDIA GPUs. We compare our real measurements to simulation results based on the average relative error. Indeed, we report the highest average relative errors on M2075 equal to 6.69% and 6.86% for runtime and energy respectively. For the evaluation on K20Xm, we observe the highest average relative errors equal to 6.54% and 8.33% for runtime and energy respectively. Finally, we look forward with those promising results of our models to extend them for studying energy efficiency by exploring optimization techniques such as concurrent kernels and Dynamic Voltage Frequency Scaling (DVFS).

ACKNOWLEDGMENT

The authors would like to thank the Grid'5000 team for providing us the platform for experimenting. The Grid'5000 experimental testbed is supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and other Universities and organizations. This work is supported by the Hac Specis Inria Project Lab.

REFERENCES

[1] P. Kogge et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," DARPA IPTO, Tech. Rep., Sep. 2008.

[2] Top500 List, *https://www.top500.org/lists/top500/*. Last accessed December 18, 2019

[3] Green500 List, *https://www.top500.org/green500/*. Last accessed August 6, 2020

[4] S. Hong and H. Kim (2012). An integrated GPU power and performance model. ACM SIGARCH Computer Architecture News, 38(3), 280. *10.1145/1816038.1815998*

[5] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa and B. Juurlink, How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator. ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software, 97–106. *10.1109/ISPASS.2013.6557150*

[6] J. Leng et al., GPU wattch†: Enabling energy optimizations in GPGPUs. Proceedings - International Symposium on Computer Architecture, (2013), 487–498. *10.1145/2485922.2485964*

[7] S. Che et al., Rodinia: A benchmark suite for heterogeneous computing, 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, 2009, pp. 44-54.*10.1109/IISWC.2009.5306797*

[8] NVIDIA,*https://docs.nvidia.com/cuda/cuda-samples/index.html*. Last accessed January 4, 2020

[9] R. A. Bridges, N. Imam and T.M. Mintz, Understanding GPU Power : A Survey of Profiling, Modeling and Simulation Methods, ACM Comput. Surv. 49, 3, Article 41 (December 2016), 27 pages. *10.1145/2962131*

[10] X. Ma, L. Zhong and Z. Deng, Statistical Power Consumption Analysis and Modeling for GPU-based Computing. Proceedings of the SOSP Workshop on Power Aware Computing and Systems (HotPower '09), 1–5. *10.1.1.157.7211*

[11] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo and S. Matsuoka, Statistical power modeling of GPU kernels using performance counters. 2010 International Conference on Green Computing, Green Comp 2010, (May 2014), 115–122. *10.1109/GREENCOMP.2010.5598315*

[12] J. Chen, B. Li ,Y. Zhang ,L. Peng and J.K. Peir,Statistical GPU power analysis using tree-based methods. 2011 International Green Computing Conference and Workshops, IGCC 2011. *10.1109/IGCC.2011.6008582*

[13] S. Song, C. Su, B. Rountree and K.W. Cameron, A simplified and accurate model of power-performance efficiency on emergent GPU architectures. Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013, 673–686. *10.1109/IPDPS.2013.73*

[14] W. Jia, E. Garza, K.A. Shaw and M. Martonosi, GPU Performance and Power Tuning Using Regression Trees. ACM Transactions on Architecture and Code Optimization (2015), 12(2), 1–26. *10.1145/2736287*

[15] NVIDIA,*https://developer.nvidia.com/nvidia-system-management-interface*. Last accessed December 18, 2019

[16] NVIDIA,*https://developer.nvidia.com/nvidia-management-library-nvml*. Last accessed February 18, 2020

[17] K. Kasichayanula et al., Power aware computing on GPUs. Symposium on Application Accelerators in High-Performance Computing. 64-73, (2012), *10.1109/SAAHPC.2012.26*

[18] NVIDIA, *https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*. Last accessed January 10, 2020

[19] J. Coplin and M. Burtscher, Power Characteristics of Irregular GPGPU Programs, Workshop on General Purpose Processing Using GPUs,Salt Lake City, UT, USA, 2014. *10.13140/2.1.2530.0485*

[20] H. Wang and Q. Chen, Power Estimating Model and Analysis of General Programming on GPU (2012). JSW, 7, 1164-1170.

[21] I. Zecena et al., Energy consumption analysis of parallel sorting algorithms running on multicore systems, in International Green Computing Conference (IGCC), San Jose, California, USA, pp. 1-6, 2012

[22] M. J. Ikram, O. A. Abulnaja, M. E. Saleh and M. A. Al-Hashimi, Measuring power and energy consumption of programs running on kepler GPUs, 2017 Intl Conf on Advanced Control Circuits Systems (ACCS) Systems and 2017 Intl Conf on New Paradigms in Electronics and Information Technology (PEIT), Alexandria, 2017, pp. 18-25. *10.1109/ACCS-PEIT.2017.8302995*

[23] SimGrid, *https://simgrid.org/doc/latest/*. Last accessed May 5, 2020

[24] NVIDIA, *https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html*. Last accessed May 6, 2020

[25] S. Williams, A. Waterman, and D. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 4 (April 2009), 65–76.

[26] J. W. Choi, D. Bedard, R. Fowler and R. Vuduc, "A Roofline Model of Energy," 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Boston, MA, 2013, pp. 661-672, doi: 10.1109/IPDPS.2013.77.

[27] A. Lopes, F. Pratas, L. Sousa and A. Ilic, "Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling," 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, CA, 2017, pp. 259-268, doi: 10.1109/ISPASS.2017.7975297.

[28] M. Ghane, J. M. Larkin, L. Shi, S. Chandrasekaran and M. S. Cheung (2018). Power and Energy-efficiency Roofline Model for GPUs. ArXiv, abs/1809.09206.

[29] J. W. Sheaffer, K. Skadron, and D. P. Luebke, Fine-grained graphics architectural simulation with Qsilver. In ACM SIGGRAPH 2005 Posters (SIGGRAPH '05). Association for Computing Machinery, New York, NY, USA, 118–es. *10.1145/1186954.1187089*

[30] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, Analyzing CUDA workloads using a detailed GPU simulator. ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software, (May), 163–174. *10.1109/ISPASS.2009.4919648*

[31] J. Zhing and B. He, Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling, IEEE Trans. Parallel Distrib., 25 (6) (2014) 1522-1532.

[32] J. Lim et al., Power Modeling for GPU Architectures Using McPAT. ACM Trans. Des. Autom. Electron. Syst. 19, 3, Article 26 (June 2014), 24 pages. *10.1145/2611758*

[33] S. Li et al., "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), New York, NY, 2009, pp. 469-480.