Multi-Kernel Auto-Tuning on GPUs: Performance and Energy-Aware Optimization

***Abstract*—Prompted by their very high computational capabilities and memory bandwidth, Graphics Processing Units (GPUs) are already widely used to accelerate the execution of many scientific applications. However, programmers are still required to have a very detailed knowledge of the GPU internal architecture when tuning the kernels, in order to improve either performance or energy-efficiency. Moreover, different GPU devices have different characteristics, moving a kernel to a different GPU typically requires re-tuning the kernel execution, in order to efficiently exploit the underlying hardware. The procedure proposed in this work is based on real-time kernel profiling and GPU monitoring and it automatically tunes parameters from several concurrent kernels to maximize the performance or minimize the energy consumption. Experimental results on NVIDIA GPU devices with up to 4 concurrent kernels show that the proposed solution achieves near optimal configurations. Furthermore, significant energy savings can be achieved by using the proposed energy-efficiency auto-tunning procedure.**

Keywords*—*GPU, GPGPU, CUDA, OpenCL, auto-tuning, multi-kernel, energy-awareness.

I. Introduction

EXPLOITING the cappabilities of Graphics Processing Units (GPUs), for general-purpose computing (GPGPU) is emerging as a crucial step towards decreasing the execution time of scientific applications. In fact, when compared to the latest generations of multi-core CPUs, modern GPUs have proved to be capable of delivering significantly higher throughputs, thus offering the means to accelerate applications from many domains [1]. However, achieving the maximum computing throughput and energy efficiency has been shown to be a very difficult task [2]. Despite the set of existing GPU programming APIs (e.g., CUDA and OpenCL), the programmer is still required to have a very detailed knowledge of the GPU internal architecture, in order to conveniently tune the program execution and efficiently exploit the available computing capabilities.

To improve performance of a GPU kernel, current optimization techniques rely on adjusting the number of thread blocks and the number of threads within the block[1] as a relative measure of kernel-architecture interrelationship. However, many modern GPUs already allow adjusting their operating frequency, thus providing the opportunity for novel energy-aware GPU kernel optimizations. In fact, tight power and dissipation constraints may also limit the possibility of efficiently employing all available GPU resources, a problem which is expected to worsen in the presence of the *Dark Silicon* phenomenon [3]. Consequently, future kernel optimization strategies must also consider the energy efficiency of the developed codes, despite the greater difficulties to analyze and design for simultaneous performance and energy efficiency optimization.

In particular, by varying the thread block size or the device operating frequency, one can achieve significant energy savings, which are tightly dependent on the application kernel under execution. Although a more flexible parameterization might offer convenient compromises between computing performance (throughput) and energy efficiency, such trade-offs are not easy to establish and are often device-dependent. This variability is also observed when considering code-portability, where significant performance and energy efficiency degradations can be observed not only when executing the same application on other devices characterized by a different amount of resources (or even a different architecture), but even when running the same program in a different GPU characterized by a rather similar architecture and resources.

To circumvent such design optimization problems, a novel methodology is herein proposed. In particular, the presented approach envisages a maximization of the attained performance or of the resulting energy efficiency, for a set of simultaneously running GPU kernels. Such objective is attained by automatically selecting a convenient distribution of the fundamental kernel parameters and an appropriate device operating frequency. Accordingly, the major contributions of this paper are the following:

- new auto-tuning procedures targeting performance or energy-aware optimization, across different GPU architectures/generations and for several simultaneously executing kernels;
- novel approach to find the most convenient thread block configuration and GPU operating frequency, that relies on the reduced design optimization space;
- extensive experimental evaluation of the proposed procedures with several applications from standard GPU benchmark suites.

Experimental results with up to 4 concurrent kernels show that the proposed procedure discovers near-optimal configurations with a significant reduced number of optimization steps. Furthermore, it shows that up to 13% energy savings can be achieved by applying the proposed energy-aware optimization.

II. Background

A. Related Work

With the emergence of GPGPU, several research works tried to address the performance portability problem of GPU kernels. In particular, RaijinCL [4] is an auto-tuning library for matrix multiplication, capable of generating optimized codes for many GPU devices. However, this approach only considers

[1]For brevity, NVIDIA CUDA notation is used in this manuscript. In OpenCL, threads are referred as work-items and thread blocks as work-groups.

a specific application (matrix multiplication), which contrasts with the proposed procedures that are completely agnostic in terms of the considered application kernels.

Kazuhiko *et al* [5] studied the difference between the performance offered by OpenCL kernels versus CUDA kernels and provided some guidelines on performance portability. Sean *et al* [6] also concluded that special measures must be considered to obtain performance portability and stated that auto-tuning approaches might be a viable way to achieve such goal.

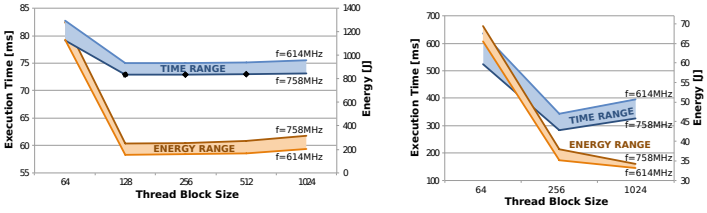
A few studies concerning the scheduling of parallel kernels on GPUs have also been presented in the last few years [7], [8]. However, they were mostly based on simulation environments using the GPGPU-SIM [9] simulator. This is mainly because the NVIDIA Kepler and Fermi architectures do not provide any mechanism to manually turn on/off the computation units, or even the possibility to individually assign application kernels to specific computation units. Kai *et al* [2] proposed a method to attain energy savings, but they aim GPU-CPU heterogeneous architectures and they are only focused on scaling the core and memory frequencies at the GPU.

Accordingly, the analysis of the current state of the art makes clear that performance and energy-aware portability on GPU devices is still an opened and unsolved problem in the GPGPU computing research area. To the best of our knowledge, there has not yet been any previous work targeting energy-aware auto-tuning, and offering performance portability across GPU devices without any further programmer intervention.

B. Motivation and Problem Statement

In general, the GPU kernel execution is organized in several thread blocks that are mapped to the existing set of SMs. Within each SM, the computations from the currently assigned thread blocks are performed in groups of data-parallel threads, which are referred as thread warps. Hence, the total number of existing thread blocks directly influences the utilization of the available SMs, while the characteristics and the number of threads (warps) allocated to each SM affects the utilization of its hardware resources. As a result, the overall GPU utilization can highly vary depending on how the total amount of the kernel work is organized in terms of the number of thread blocks and the number of assigned threads per block.

Furthermore, constraints such as the number of registers and the amount of shared memory do not only limit the maximum number of thread blocks per SM, but also affect the overall performance. However, these constraints are not necessarily the same for all GPUs. In fact, they have even been significantly changing in the latest GPU generations, forcing programmers to re-tune the applications every time they are ported to a different device. Hence, the selection of the most convenient configuration of the application kernel over the GPU has proven to be a non-trivial design optimization parameter [6]. This is emphasized on the kernel configuration examples that are illustrated in Figure 1, where significant performance differences can be observed when varying the thread block size (see "time range").



(a) Example kernel 1. (b) Example kernel 2.

Fig. 1: Kernel parameterization optimization examples in terms of performance and energy.

A non-trivial relation that can be observed in Figure 1 concerns the configurations achieving maximal performance and energy efficiency (see "time range" vs. "energy range"). In fact, even though performance and energy optimizations may result in the same kernel configuration for certain applications (e.g., 128 in Figure 1a), for other cases the optimal energy-efficient configuration considers a completely different distributions (e.g., 256 vs. 1024 in Figure 1b). This unpredictable behaviour, in terms of kernel configuration, opens the door for the new set of auto-tuning procedures that are proposed herein. By following the proposed methodology, the programmer is able to simply supply the kernel and GPU information and get, as output, the most convenient configuration (e.g., thread block size or GPU operating frequency) that allows him to further optimize the kernel for either the attained performance or the energy consumption. Furthermore, the presented procedure is completely kernel and GPU agnostic: it can work with any set of kernels and on any GPU device, independently of their characteristics and architectures, as shown in Section IV.

III. Automatic Tuning of GPU Kernel Execution

The procedures proposed in this paper allow the automatic determination of the most adequate configuration for a set of concurrent GPU kernels, such that different optimization goals can be achieved. In order to tackle fundamental aspects of modern GPU architectures and applications, two different auto-tuning procedures are derived that allow optimizing the GPU kernel execution for performance and energy-efficiency. By reducing the overall optimization space to a subset of possible GPU kernel configurations (in terms of the number of threads and thread blocks), the proposed procedures do not only allow for a fast discovery of the most adequate configurations, but they also guarantee a high utilization of the GPU resources.

A. Optimization Space: Considerations and Reduction

To constraint the optimization space to a subset of possible GPU kernel configurations, the proposed auto-tuning procedures take into account that the performance is maximized when all SMs are kept busy [1]. This is also valid for the energy-aware optimization procedure, motivated by the fact that even when idle, the power consumption of a single SM is typically relatively high [10]. Consequently, by efficiently using all the available SMs, one reduces the overall kernel execution time at a relatively low increase in power consumption.

In order to identify a full set of viable kernel configurations, it is firstly required to determine all possible distributions of the considered threads within each thread block. In order to verify

TABLE I: Architecture and kernel-specific parameters.

Type	Notation	Description
Architecture-specific	$\mathrm{devMaxTB_{SM}}$	max. num. of blocks per SM
	$\mathrm{devMaxW_{SM}}$	max. num. of warps per SM
	$\mathrm{devMaxSharedMem_{SM}}$	shared memory per SM
	$\mathrm{devSharedMemAllocUnit}$	shared memory alloc. unit size
	$\mathrm{devRegsLimit_{SM}}$	max. num. of registers per SM
	$\mathrm{devRegAllocUnit}$	registers alloc. unit size
	$\mathrm{devWarpSize}$	warp size
	$\mathrm{devMaxThreads_{SM}}$	max. num. of threads per SM
Kernel-specific	$\mathrm{kerW_{TB}}$	num of warps per thread block
	$\mathrm{kerSharedMem_{TB}}$	shared memory per block
	$\mathrm{kerRegs_{T}}$	num. of registers per thread
Architecture- and kernel-specific	$\mathrm{maxTB_{SM}}$	max. num. of thread blocks
	$\mathrm{occupancy}$	hardware utilization measure

such feasibility, the maximum number of thread blocks that can be realistically assigned per SM ($\mathrm{maxTB_{SM}}$) is calculated by considering three main limiting factors, namely: *i*) the maximum number of thread blocks per SM as sustained by the device; *ii*) the register availability; and *iii*) the amount of shared memory. For each of these limiting factors, a distinct value for the maximum number of thread blocks per SM is calculated, i.e., $\mathrm{TB_{lim1}}$, $\mathrm{TB_{lim2}}$ and $\mathrm{TB_{lim3}}$. Then, the minimum among these values is assigned as $\mathrm{maxTB_{SM}}$, such that:

$$\mathrm{maxTB_{SM}} = \min\{\mathrm{TB_{lim1}}, \mathrm{TB_{lim2}}, \mathrm{TB_{lim3}}\}. \quad (1)$$

Table I summarizes a set of architecture-specific and kernel-specific parameters that are used during the calculation of $\mathrm{TB_{lim1}}$, $\mathrm{TB_{lim2}}$ and $\mathrm{TB_{lim3}}$ values.

For each considered kernel, $\mathrm{TB_{lim1}}$ represents the maximum number of thread blocks per SM that does not violate the maximum device capabilities, i.e, the maximum number of thread blocks per SM ($\mathrm{devMaxTB_{SM}}$), and the maximum number of warps per SM ($\mathrm{devMaxW_{SM}}$). It is calculated as:

$$\mathrm{TB_{lim1}} = \min\left\{\mathrm{devMaxTB_{SM}}, \left\lfloor \frac{\mathrm{devMaxW_{SM}}}{\mathrm{kerW_{TB}}} \right\rfloor \right\}, \quad (2)$$

where $\mathrm{kerW_{TB}}$ represents the number of warps per thread block for the currently considered distribution, which is obtained by dividing the total number of threads in a block by the size of the warp ($\mathrm{devWarpSize}$).

$\mathrm{TB_{lim2}}$ is the maximum number of thread blocks that does not violate the shared memory constraints, computed as:

$$\mathrm{TB_{lim2}} = \left\lfloor \frac{\mathrm{devMaxSharedMem_{SM}}}{\mathrm{kerSharedMem_{TB}}} \right\rfloor. \quad (3)$$

The $\mathrm{kerSharedMem_{TB}}$ value can be calculated by dividing the amount of shared memory consumed by the kernel (given at compile time) and the shared memory allocation unit size ($\mathrm{devSharedMemAllocUnit}$).

The $\mathrm{TB_{lim3}}$ value represents the upper bound on the number of thread blocks, that is constrained by the number of registers in the SM. This value is calculated as:

$$\mathrm{TB_{lim3}} = \left\lfloor \frac{\mathrm{devRegsLimit_{SM}}}{\mathrm{kerRegs_{W}}} \right\rfloor / \mathrm{kerW_{TB}}, \quad (4)$$

where $\mathrm{kerRegs_{W}}$ is the number of registers used by the kernel per warp, calculated with the number of register per thread ($\mathrm{kerRegs_{T}}$), the register allocation unit size ($\mathrm{devRegAllocUnit}$) and the warp size ($\mathrm{devWarpSize}$).

Furthermore, to reduce the optimization space for kernel configurations, the device *occupancy* is used as the relative measure of GPU utilization [11]. In detail, once the $\mathrm{maxTB_{SM}}$ value is determined, the theoretical GPU occupancy for the current kernel configuration/distribution is computed as:

$$\mathrm{occupancy} = \frac{\mathrm{maxTB_{SM}} * \mathrm{kerW_{TB}} * \mathrm{devWarpSize}}{\mathrm{devMaxThreads_{SM}}}. \quad (5)$$

This measure represents (in percentage) the GPU ability to process the active warps [1]. As soon as the theoretical occupancy on a per distribution basis is calculated, only the distributions with higher occupancy are selected (e.g., $\mathrm{occupancy}{>}80\%$), as they provide the possibility of attaining the best performance, i.e., the minimum execution time. A similar approach to calculate the occupancy of each distribution that can also be adopted is the usage of the CUDA occupancy calculator [11].

B. Performance Auto-Tuning Procedure

Current trends in GPU kernel optimization mainly consider the performance of the developed codes, i.e., the optimization goal is to attain the minimum execution time. In order to ease this process, the proposed procedure aims at automatic tuning of kernel configuration parameters, and its main functionality is outlined in Algorithm 1. Based on the previously referred set of architecture-specific and kernel-specific parameters (see Table I), the proposed approach determines the configuration (d) that minimizes the kernel execution time.

A common approach when searching for the most adequate configuration is to exploit a full range of optimization parameters. However, this process may be a very time consuming and practically unfeasible process (even at the level of a single GPU device). In order to overcome this issue, the proposed procedure starts with a reduction of the overall optimization space, by selecting a subset of configurations with high device occupancy, according to the previously referred considerations (see steps 1–5 in Algorithm 1). Then, the subset of considered distributions is stored in a vector D and sorted in a non-decreasing order, based on their occupancy. If several distributions result in the same occupancy, they are subsequently ordered according to the number of thread blocks.

Whenever several different distributions need to be examined (steps 6–11 from Algorithm 1), the proposed procedure first selects the distribution with the best potential to provide the highest performance ($D[0]$), launches the kernel with that configuration and records the execution time ($T[0]$). Then, the next configuration from the D vector is evaluated. After each examined configuration ($D[i_d]$), its execution time ($T[i_d]$) is compared with the execution time obtained for the previously considered configuration ($T[i_d{-}1]$). As long as the currently obtained time results in a reduction of the execution time over the previously examined configuration, the proposed procedure continues with the auto-tuning by selecting the next configuration from the D vector. Once the currently examined

Algorithm 1 Performance Auto-Tuning Procedure.

Input: set of architecture-specific and kernel-specific parameters
Output: configuration (d) corresponding to the minimum execution time

1: **for all** distributions of the number of threads in a block **do**
2: Calculate $\mathrm{maxTB_{SM}}$ and theoretical occupancy;
3: **end for**
4: Store distributions with higher occupancy in vector D;
5: Sort D in non-decreasing order of occupancy (and thread block size);
6: **if** $length(D) > 1$ **then**
7: **for** i_d=0 to $length(D)$ **do**
8: Run kernel with distribution $D[i_d]$ and measure run time $T[i_d]$;
9: **if** $(i_d{>}0){\wedge}(T[i_d]{>}T[i_d{-}1])$ **then return** distribution $d{=}D[i_d{-}1]$;
10: **end for**
11: **end if**
12: **return** distribution $d{=}D[length(D){-}1]$;

configuration does not allow any further reduction of the execution time, the auto-tuning procedure is finalized and the previously examined configuration is marked as the one with the best discovered performance (see step 9 in Algorithm 1). As expected, if the reduced optimization space contains a single distribution, that configuration is immediately marked as the best one (see step 12 in Algorithm 1).

Whenever kernels with a significant amount of work to be distributed are considered, the proposed procedure will always converge with a significantly faster rate than the exhaustive search, due to reduced (and finite) number of possible configurations. In addition, by iteratively exploiting the configurations from such a reduced optimization space, the proposed procedure allows the GPU developers to perform fast early stage evaluations of the developed codes (e.g., evaluation of how the different kernel configurations affect the attainable execution time and how they fit to the underlying GPU hardware). In fact, the conducted performance evaluations can provide valuable insights when: *i*) detecting the possible execution bottlenecks; *ii*) obtaining further optimization guidelines; and *iii*) analyzing the portability of the developed codes across different GPU architectures. This is especially relevant to enable the porting of kernels optimized for a given architecture to different generations of GPU architectures (e.g., NVIDIA Tesla, Fermi and Kepler, or future architectures) and/or across GPU devices with different capabilities from the same architecture (e.g., NVIDIA GTX680 vs. Tesla K40c).

C. Energy-Aware Auto-Tuning Procedure

Energy-aware optimizations require dealing with a significantly larger optimization space than the one that is considered when optimizing for performance. In particular, besides the usual range of GPU kernel configurations, different frequency ranges at which the modern GPU devices can operate must also be taken into account. As a result, the optimization space for energy-efficiency represents a combination of the performance exploration space with the selection of the most suitable operating frequency.

In order to ease this process, a novel *procedure for energy-aware auto-tuning* is proposed herein, as outlined in Algorithm 2. This procedure allows the application programmer

Algorithm 2 Energy-Aware Auto-Tuning Procedure.

Input: set of architecture-specific and kernel-specific parameters
Input: F vector with GPU core frequency levels
Output: configuration (d) corresponding to the minimum energy consumption
Output: frequency (f) corresponding to the minimum energy consumption

1: (see steps 1–5 from Alg. 1)
2: Sort F in non-decreasing order of frequency levels;
3: Set i_f=0, d=0, $f_{min}{=}F[length(F)]$;
4: Set $f{=}{-}1$;
5: **while** $(i_f{<}length(F)){\wedge}(f{=}{=}{-}1)$ **do**
6: Set core frequency at $F[i_f]$ and run kernel with $D[d]$;
7: Measure execution time $T[d, i_f]$ and energy consumption $E[d, i_f]$;
8: **if** $(i_f{>}0){\wedge}(E[d, i_f]{>}E[d, i_f{-}1])$ **then** $f{=}F[i_f{-}1]$; **else** $i_f{=}i_f{+}1$;
9: **end while**
10: **if** $(f{=}{=}{-}1)$ **then** $f{=}f_{min}$;
11: Set core frequency at f;
12: **if** $(length(D){>}1)$ **then**
13: **for** i_d=0 to $length(D)$ **do**
14: Run kernel with $D[i_d]$; measure time $T[i_d, f]$ and energy $E[i_d, f]$;
15: **if** $(i_d{>}0){\wedge}(E[i_d, f]{>}E[i_d{-}1, f])$ **then**
16: **if** $(d{\neq}i_d{-}1){\wedge}(f{\neq}f_{min})$ **then** $d{=}i_d{-}1$; **goto** step 4;
17: **else return** distribution $d{=}D[i_d{-}1]$ and frequency f;
18: **end if**
19: **end for**
20: **end if**
21: **return** distribution $d{=}D[length(D){-}1]$ and frequency f;

to obtain useful insights on the most suitable GPU kernel configuration in terms of energy-efficiency, i.e., it determines the configuration d and frequency f that allow high utilization of GPU resources with the minimum energy consumption.

Similarly to the performance auto-tuning, the proposed procedure starts by performing the previously referred reduction of the optimization space and by creating the ordered D vector with a set of configuration candidates (step 1 in Algorithm 2). Then, the first configuration candidate from D is selected (step 3) and the variation in energy consumption is assessed over different GPU frequency levels. In detail, the proposed procedure aims at reducing the GPU operating frequency $F[i_f]$ (from an ordered frequency range in vector F), as long as the reduction in the energy consumption is attained for the currently examined candidate $D[d]$ (see steps 5–9 in Algorithm 2). Once executing at a lower frequency does not result in further reduction of energy consumption, i.e., the increased execution time does not compensate the reduction in power consumption, the previously determined frequency level is stored ($f{=}F[i_f{-}1]$).

Afterwards, the algorithm sets the GPU operating frequency at level f and starts evaluating the configuration candidates from D (see steps 11–21 in Algorithm 2). The evaluation of different configurations $D[i_d]$ is performed as long as they allow further reduction in energy consumption (see step 15 in Algorithm 2). If the determined configuration d with the minimum energy consumption differs from the one previously tested for frequency ranges, the proposed procedure proceeds with the evaluation across the remaining frequency ranges (steps 4–9 in Algorithm 2). As soon as the found distribution d matches the one previously tested for frequency ranges, the auto-tuning is finalized and the best distribution d and frequency level f are returned (see 17 and 21 in Algorithm 2).

Algorithm 3 Multi-Kernel Performance Auto-Tuning.

Input: architecture-specific and kernel-specific parameters for each kernel
Output: configurations d_i for each kernel i

1: **for** i=0 to number of kernels **do**
2: **for all** distributions of the number of threads in a block **do**
3: Calculate $\mathrm{maxTB_{SM}}$ and theoretical occupancy;
4: **end for**
5: Store distributions with high occupancy in vector D_i;
6: Sort D_i in non-decreasing order of occupancy (and thread block size);
7: **end for**
8: Run all kernels with distribution δ_i=$D_i[0]$ and measure run time $T_i[0]$;
9: Create vector K with kernel indexes sorted in decreasing execution time;
10: **for** i=0 to number of kernels **do**
11: for $K[i]$ kernel **run** steps 6–15 from **Alg. 1** to obtain d_i configuration;
12: **if** $(i{>}0){\wedge}(d_i{\neq}\delta_i)$ **then** δ_i=d_i; i=0;
13: **end for**

D. Automatic Tuning of Multiple GPU Kernels

Based on the contributions of the previously described single-kernel auto-tuning approaches, a set of procedures for simultaneous *automatic tuning of multiple GPU kernels* is proposed for different optimization goals. In brief, the proposed multi-kernel procedures automatically determine the most adequate configuration for each kernel in respect to the other simultaneously running kernels, i.e., a set of per-kernel configurations (and frequency level) that maximize performance or minimize energy consumption at the level of the whole execution. As a result, the proposed approaches provide the means for run-time multi-kernel execution optimization, even in the presence of shared resource contention, when several kernels are simultaneously co-scheduled.

The main functionality behind the proposed approach for the performance domain is presented in Algorithm 3. Similarly to the single-kernel approaches, this procedure firstly reduces the optimization space and creates a set of ordered D_i vectors with configuration candidates for each considered kernel i (see steps 1–7 in Algorithm 3). Afterwards, the proposed procedure selects the distributions with the greatest potential to provide good performance for each of the considered kernels ($D_i[0]$), marking such distribution as the preliminary distribution (δ_i). The respective kernels are then simultaneously launched with δ_i distributions and the execution time is recorded ($T_i[0]$). Based on this preliminary evaluation, vector K is created with the kernel indexes sorted in decreasing order of the obtained execution time (see steps 8–9 in Algorithm 3).

In steps 10–13, the algorithm iteratively examines the individual configurations for each kernel from K vector, such that the minimum overall execution time is attained. In detail, in order to determine the best configuration (d_i) for each currently examined kernel, the proposed procedure relies on the previously described algorithm for single-kernel performance auto-tuning (see steps 6–15 in Algorithm 1). Whenever the obtained distribution (d_i) differs from the δ_i distribution, the preliminary δ_i distribution takes the value of the currently determined one (δ_i=d_i) and the iterative procedure is restarted (see steps 10–13 in Algorithm 3). The proposed procedure stops when all the preliminary distributions match the currently determined ones.

Algorithm 4 Multi-Kernel Energy-Aware Auto-Tuning.

Input: architecture-specific and kernel-specific parameters for each kernel
Input: vector F with GPU core frequency levels
Output: configuration d_i for each kernel i
Output: frequency (f) corresponding to the minimum energy consumption

1: (see steps 1–7 from Alg. 3)
2: Sort F in non-decreasing order of frequency levels;
3: Set i_f=0, d_i=0, f_{min}=$F[length(F)]$;
4: Set f=-1;
5: **while** ($i_f{<}length(F)$) and ($f{==}{-}1$) **do**
6: Set core frequency at $F[i_f]$ and run all kernels with $D_i[d_i]$;
7: Measure total execution time $T[i_f]$ and energy consumption $E[i_f]$;
8: **if** $(i_f{>}0){\wedge}(E[i_f]{>}E[i_f{-}1])$ **then** f=$F[i_f{-}1]$; **else** i_f=$i_f{+}1$;
9: **end while**
10: **if** ($f{==}{-}1$) **then** f=f_{min};
11: Set core frequency at f and δ_i=d_i;
12: Create K, with kernel indexes sorted in decreasing energy consumption;
13: **for** i=0 to number of kernels **do**
14: for $K[i]$ kernel **run** steps 12–21 from **Alg. 2** to obtain d_i configuration;
15: **if** $(i{>}0){\wedge}(d_i{\neq}\delta_i)$ **then goto** step 4; δ_i=d_i; i=0;
16: **end for**
17: **return** distribution d_i=$D_i[length(D_i){-}1]$ and frequency f;

As it can be observed, this procedure is based on the fact that the minimization of the overall execution time for several simultaneously running kernels can only be achieved if the minimum execution time is attained at the level of each individual kernel (when simultaneously co-scheduled with the other GPU kernels). Furthermore, this procedure also adheres to the current parallel execution paradigm adopted in GPU architectures. For example, for several simultaneously launched applications, the GPU block scheduler always tries to satisfy the demands of the first application in terms of the number of SMs to be allocated (for thread blocks). In fact, this allocation is usually performed in such a way that the thread blocks from a single (first) application are spread as much as possible on the available SMs. As a result, the concurrent execution of the remaining applications highly depends on the availability of the GPU resources upon their reservation for previously lunched kernels.

Algorithm 4 presents the main functionality of the proposed procedure for simultaneous optimization of several GPU kernels, such that the energy consumption is minimized at the level of the whole execution. This procedure determines a set of per-kernel configurations d_i and a single frequency level f for all currently running kernels. This is due to the fact that, in current GPU architectures, the core frequency can only be altered at the level of the whole GPU chip. As referred before, the proposed procedure relies on reducing the optimization space to create a set of ordered D_i vectors with configuration candidates for each considered kernel i (see step 1 in Algorithm 4). Then, the first configuration candidates from each D_i set are selected and simultaneously run for different GPU frequency levels (steps 2–10). Similarly to the single-kernel procedure (see Algorithm 2), this process of reducing the operating frequency (from the ordered F set) is performed as long as the overall energy consumption (for all kernels) is reducing. Once lowering down the frequency does not provide any reduction of the energy consumption, the

TABLE II: Considered benchmark applications.

Application	Suite	Problem Size
lud	Rodinia	4096×4096
hotspot	Rodinia	8192
particle filter	Rodinia	100000
streamcluster	Rodinia	65536
MatrixMul	SDK	4096×4096

TABLE III: Characteristics of the used GPU devices.

	K40c	K20c	GTX 680	GTX 580
Base architecture	Kepler	Kepler	Kepler	Fermi
Compute capability	3.5	3.5	3.0	2.0
# SMs	15	13	8	16
# CUDA cores	2880	2496	1536	512
SM freq. ranges (MHz)	875, 810 745, 666	758, 705, 666 640, 614	1058	1594
Memory freq. (MHz)	3004	2600	3004	2025
# Registers	64K	64K	64K	32K
Shared mem. / Block	48K	48K	48K	48K
Max threads / SM	2048	2048	2048	1536
Max active blocks / SM	16	16	16	8

previously determined frequency level is stored (f=$F[i_f-1]$).

Afterwards, the GPU core frequency is set to f, the previously examined distributions are marked as preliminary (δ_i=d_i) and vector K is created with the kernel indexes sorted in the decreasing order of energy consumption (see steps 11–12 in Algorithm 4). Then, the individual configurations for each kernel from K vector are iteratively examined (see steps 13–15). For each currently examined kernel, the best configuration (d_i) is found by relying on a single-kernel procedure (see steps 12–21 in Algorithm 2). When the determined distribution (d_i) differs from δ_i, the δ_i distribution is assigned with the value of the determined one (δ_i=d_i). Then, the procedure is restarted from the frequency selection step, i.e., step 4 in Algorithm 2. The proposed procedure stops when all preliminary distributions match the currently determined ones.

IV. Experimental Results

To evaluate the proposed single and multi-kernel optimization procedures, a set of CUDA kernels was considered, as presented in Table II. The kernels were extracted from both the NVIDIA SDK [12] and Rodinia Benchmarks suite [13]. Automatic optimization of the execution of these benchmarks was performed on a set of GPU devices of different characteristics, namely: GTX580 (Fermi architecture) and GTX680, Tesla K20c and Tesla K40c (Kepler architecture). The characteristics of these devices are summarized in Table III. The kernel execution time was measured using the Performance Application Programming Interface (PAPI) to interface with the host CPU Time Stamp Counter (TSC), and by executing multiple times each considered configuration. Power consumption was measured by using the built-in sensors of Tesla K20c and K40c GPUs. As proposed in [14], the measured power values P_m were corrected by using the following expression:

$$\hat{P}(t_i) = P_m(t_i) + C\frac{P_m(t_{i+1}) - P_m(t_{i-1})}{t_{i+1} - t_{i-1}} \quad (6)$$

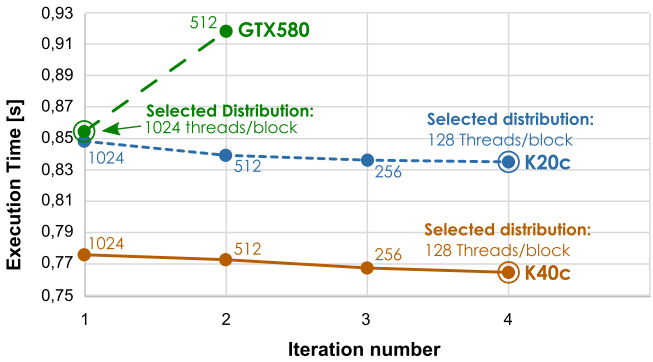


Fig. 2: Performance optimization procedure applied to the *streamcluster* kernel.

where $\hat{P}(t_i)$ is the corrected value, t_i is the time of power sample i and C is a constant representing the GPU capacitance. By using the approach proposed in [14] the capacitance C of each GPU was experimentally determined as: $C_{K20c} = 0.833$ and $C_{K40c} = 0.79$. The energy consumption was determined by computing the integral of the corrected power consumption over the kernel execution time.

As it was previously stated, the first step of the proposed auto-tuning procedures gathers the GPU characteristics and the kernel profiling information (see Table I). Since the presented evaluation relies on CUDA kernels, the *cudaGetDeviceProperties()* function is used for the former, whereas the later is obtained by using the NVIDIA compiler (*nvcc*) with flag -*Xptxas*=-*v*. Notwithstanding, a similar procedure could be used for OpenCL kernels.

A. Performance and Energy-Aware Optimization

Figure 2 presents the obtained results of the performance optimization procedure applied to the *streamcluster* kernel. By considering that the performance is maximized at the highest operating frequency and that the best configuration lies on power of 2 solutions, a total of 10 possible combinations can be evaluated. However, by cross comparing the GPU characteristics with the kernel profile information, the whole search space is constrained to 4 possible combinations. Nonetheless, not all possible combinations need to be tested with the proposed approach; as an example, for the GTX580 GPU, only 2 combinations are tested to determine the optimal configuration (which is achieved in all GPUs for the *streamcluster* benchmark).

By using the proposed energy-aware optimization procedure, it is also possible to optimize the kernels distribution and GPU operating frequency such as to minimize the energy consumption. While the search space of the performance optimization procedure is generally constrained to the highest operating frequencies, this is not the case for energy-aware optimization. As a consequence, the complete search space (number of possible solutions) considers all the combinations of possible blocksize configurations and frequency levels. As an example, a total number of 50 (40) possible configurations exist, for the *streamcluster* benchmark, when optimizing for the Tesla K20c GPU (K40c GPU). To reduce the search space,

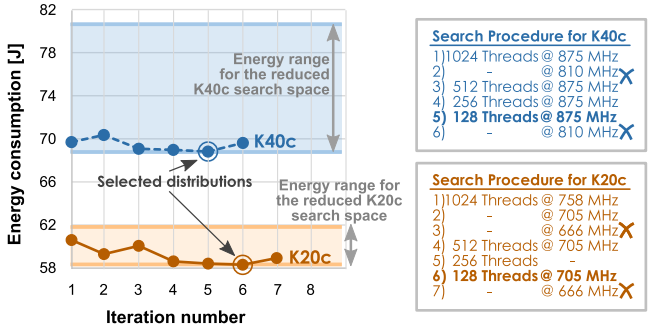
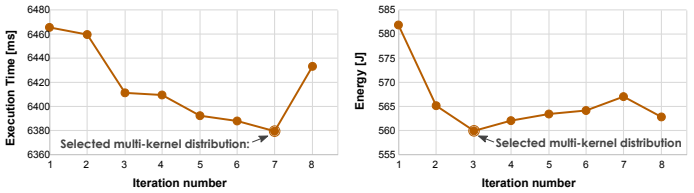


Fig. 3: Energy-aware optimization procedure for the *Streamcluster* kernel. The full search space for energy minimization includes 50 and 40 possible configurations for the K20c and K40c GPU, respectively.

the proposed optimization procedure starts by constraining the possible solutions to the cases with the highest theoretical GPU occupancy, reducing the search space to 20 (16) solutions. Then, the optimization procedure iteratively inspects the reduced search space according to Algorithm 2, such as to obtain a configuration with reduced energy consumption. As shown in Figure 3, only a set of up to 7 (6) possible configurations are required to achieve the minimum energy consumption case, for the *streamcluster* kernel.

Figure 4 presents the results of the performance and energy aware optimization procedures for a multi-kernel environment, composed by 4 kernels (*Lud*, *Streamcluster*, *Hotspot* and *ParticleFilter*). In the performance optimization case, 2250 possible combinations of kernel distributions exist when optimizing for the K40c GPU. After reducing the search space 48 possible combinations remain. However, only 8 need to be tested with the proposed procedure, as it is shown in Figure 4a. When executing the energy-aware procedure on the K40c GPU, the whole search space is composed of 9000 combinations, which are reduced to 192, by making the same considerations mentioned before. As it is shown in Figure 4b only 8 combinations of kernel distributions and GPU frequencies were tested the proposed approach.

The results for the performance and energy-aware optimization procedures for the full set of considered benchmarks are summarized in Table IV. To better analyze the proposed optimization procedures, this table includes: the auto-tuned configuration (columns *Blocksize* and *Frequency*); the number of iterations required to achieve the presented solution out of the total number of possible combinations (column *#iterations*); and the result of the performance/energy trade-off between the energy-aware and performance optimization procedures (columns *performance degradation* and *energy savings*). By analyzing the results in Table IV, it can be observed that the proposed optimization procedure is able to find a near-optimal solution using a reduced set of iterations, with a maximum of 8 iterations for the 4-kernel configuration. This significantly contrasts with an exhaustive search procedure that requires examining a large set of possible solutions, especially for the energy-aware and multi-kernel cases. As mentioned before, for a 4-kernel configuration, the total set of possible solutions is



(a) Performance. (b) Energy.

Fig. 4: Performance and energy-aware optimization procedure for 4 concurrent kernels, namely *Lud*, *Streamcluster*, *Hotspot* and *ParticleFilter*, executed on K40c GPU.

2250 and 9000 for the performance and energy-aware cases, respectively. Furthermore, not only is the number of iterations significantly reduced, but the obtained configurations are also very close to the optimal ones. In particular, the proposed algorithms were even able to find the oracle solutions in most considered cases.

From Table IV, it can also be observed that not only does the optimal kernel distribution changes from GPU to GPU, but it can also change when multiple kernels are concurrently executed. As an example, for the *Particle Filter* single-kernel case, the best distribution ranges between 128 and 512 threads/block when using the Tesla K20c and the GTX580 GPU, respectively. Furthermore, by comparing the kernel distributions and operating frequency of the concurrent 4-kernel case and the corresponding cases when each kernel is individually executed, different solutions are observed. This reinforces the advantages of integrating the proposed auto-tuning procedures with the applications source-code, in order to allow optimizing the kernel execution to the target GPU.

Finally, it is also important to highlight that significant energy-savings can be achieved by applying an energy-aware optimization procedure. In particular, an average of 4% (3%) energy saving is observed for Tesla K20c (Tesla K40c), with a maximum value of 13% (12%) for the *Lud* kernel.

V. Conclusion

This paper proposes a set of novel single- and multi-kernel auto-tuning procedures for performance and energy-aware optimizations, which find near-optimal solutions concerning the kernel thread block configurations and the GPU operating frequency. The proposed procedures do not only aid application developers to optimize their code, but also they provide the means for efficient code portability across different generations of GPU architectures. To achieve such a goal, search space reduction is first applied to construct a constrained set of possible configurations. Then, the proposed procedures are applied to select the most appropriate kernel configuration (and GPU operating frequency), depending on the optimization goal, i.e., to maximize performance and/or to minimize energy consumption. As a result, the proposed procedures are able to find the near-optimal solutions by using a small number of iterations (up to 8 for the presented cases).

Experimental results highlight the importance of the proposed procedures, by showing that the optimal kernel distributions can significantly vary when porting the kernels to GPUs with different architectures and/or characteristics. Furthermore,

TABLE IV: Summary of the obtained results with the performance and energy-efficiency auto-tuning procedures.

BENCHMARK	GPU	PERFORMANCE AUTO-TUNNING		ENERGY-EFFICIENCY AUTO-TUNNING				
		Blocksize	#iterations	Blocksize	Frequency	#iterations	Performance degradation	Energy savings
SINGLE KERNEL OPTIMIZATION								
StreamCluster	K20c	128	4 (10)	128	705 MHz	7 (50)	2%	2%
	K40c	128	4 (10)	128	875 MHz	7 (40)	0%	0%
	GTX680†	1024	2 (10)	-	-	-	-	-
	GTX580†	1024	2 (10)	-	-	-	-	-
Particle Filter	K20c	128	3 (9)	512	758 MHz	3 (45)	$< 1\%$	1%
	K40c	128	3 (9)	128	810 MHz	7 (36)	8%	1%
	GTX680†	512	3 (9)	-	-	-	-	-
	GTX580†	256	3 (9)	-	-	-	-	-
Lud	K20c	16×16	2 (5)	32×32	705 MHz	4 (25)	19%	13%
	K40c	16×16	2 (5)	32×32	810 MHz	4 (20)	18%	12%
HotSpot	K20c	16×16	1 (5)	16×16	758 MHz	2 (25)	0%	0%
	K40c	16×16	1 (5)	16×16	875 MHz	2 (20)	0%	0%
MULTIPLE KERNEL OPTIMIZATION								
Lud & Particle Filter	K20c	16×16 256	5 (45)	32×32 512[(a)]	758 MHz	6[(a)] (225)	13%	5%
Lud & MatrixMul	K20c	16×16 32×32	3 (25)	16×16 32×32	705 MHz	5 (125)	6%	5%
StreamCluster & HotSpot	K40c	128 16×16	4 (50)	512[(b)] 16×16[(b)]	810 MHz	6[(b)] (200)	$< 1\%$	$< 1\%$
Streamcluster & ParticleFilter & Lud & Hotspot	K40c	128 128 16×16 32×32	8 (2250)	1024 512 32×32 32×32	745 MHz	8 (9000)	7%	4%

† Energy optimization not possible since there is no built-in power sensor.

[(a)] The optimal configuration (32×32, 128 @ 640 MHz) was not found, which would use 0.3% less energy.

[(b)] The optimal configuration (1024, 32×32 @ 810 MHz) was not found, which would use 0.2% less energy.

depending on the kernel (or a set of kernels), significant energy savings can be achieved by applying the proposed energy-aware optimization strategies.

REFERENCES

[1] "CUDA C Best Practices Guide," available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[2] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *41st International Conference on Parallel Processing (ICPP)*. IEEE, 2012, pp. 48–57.

[3] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 365–376.

[4] R. Garg and L. Hendren, "A portable and high-performance general matrix-multiply (GEMM) library for GPUs and single-chip CPU/GPU systems," in *22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2014.

[5] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in *The fifth international workshop on automatic performance tuning*, 2010, p. 7.

[6] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An experimental study on performance portability of OpenCL kernels," in *Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[7] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 157–166.

[8] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *18th International Symposium on High Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[9] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2009, pp. 163–174.

[10] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *40th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2013, pp. 487–498.

[11] NVIDIA, "CUDA occupancy calculator," 2011, available: http://developer.download.nvidia.com/compute/cuda/3_1/sdk/docs/CUDA_Occupancy_calculator.xls.

[12] ——, "CUDA C/C++ SDK code samples," 2011, available: http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[14] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 2014, p. 28.