Power-aware Performance Tuning of GPU Applications Through Microbenchmarking

ABSTRACT

Tuning GPU applications is a very challenging task as any source-code optimization can sensibly impact performance, power, and energy consumption of the GPU device. Such an impact also depends on the GPU on which the application is run. This paper presents a suite of microbenchmarks that provides the actual characteristics of specific GPU device components (e.g., arithmetic instruction units, memories, etc.) in terms of throughput, power, and energy consumption. It shows how the suite can be combined to standard profiler information to efficiently drive the application tuning by considering the three design constraints (power, performance, energy consumption) and the characteristics of the target GPU device.

Keywords

GPU; Performance; Power; Microbenchmarking.

1. INTRODUCTION

Graphic Processing Units (GPUs) have become increasingly used as general-purpose accelerators thanks to their computational power and programmability. Besides providing high performance, they also achieve excellent energy efficiency [12]. This makes them well suited to a variety of architectures, ranging from supercomputers to low-power and mobile devices [1].

On the other hand, the large number of operating hardware resources (e.g., cores and register files) employed in GPUs to support the massive parallelism can lead to a significant power consumption. The elevated levels of power consumption have a sensible impact on such many-core device reliability, ageing, performance scaling and deployment into a wide range of application domains. Different techniques have been proposed to manage the high levels of power dissipation and to continue scaling performance and energy. They include approaches based on dynamic voltage/frequency scaling (DVFS) [7], CPU-GPU work division [10], architecture-level/runtime adaptations [19], dynamic resource allocation [6], and application-specific (i.e., programming-level) optimizations [22]. Particularly in this last category, it has been observed that source-code-level transformations and application specific optimizations can significantly affect the GPU resource utilization, performance, and energy efficiency [17].

In this context, even though profiling tools (e.g., *CUDA nvprof*) exist to help programmers in the application analysis and optimization targeting performance, they do not provide a complete view of the GPU features (especially on power consumption and energy efficiency) neither they provide a correlation among these design constraints. What is missing is a solution to measure, on a given GPU architecture, the potential effects of code optimizations on every design constraint before implementing them.

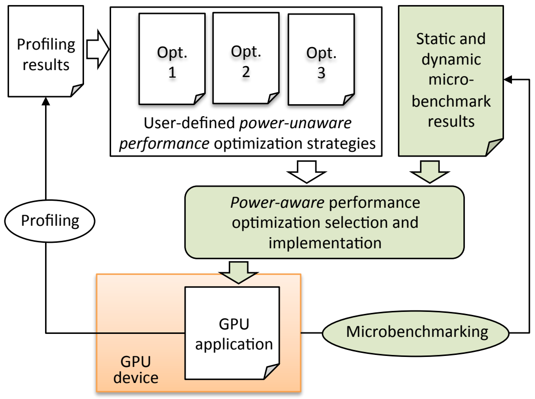

Profiling results
Opt. 1
Opt. 2
Opt. 3
User-defined *power-unaware performance* optimization strategies
Static and dynamic micro-benchmark results
Profiling
Power-aware performance optimization selection and implementation
GPU application
GPU device
Microbenchmarking

FIG. 1: *Overview of the proposed approach.*

To overcome this limitation, this paper presents a suite of microbenchmarks, which aims at *characterizing* a GPU device in terms of performance, power, and energy consumption. The microbenchmark suite has been designed to be compiled and run on any CUDA GPU device, with the aim of quantitatively characterizing, *statically* and *dynamically*, all the functional components of the device. The functional components include arithmetic instruction units, memories (shared, cache, DRAM, constant), scheduling and synchronization units.

Fig. 1 shows how the proposed microbenchmarking can be combined with the standard profiling for a power-aware performance tuning of GPU applications. Given a GPU application, the standard profiling information allows defining a set of potential optimizations targeting performance. The microbenchmarks are run once for all in the target GPU device. By considering the functional components involved by an optimization strategy, the microbenchmark results on such components allow classifying the potential and the useless optimization strategies for the target design constraint before implementing them. The model allows the flow to be iterated for incremental tuning of the application.

The suite has been applied to characterize two different GPU devices (i.e., NVIDIA Kepler GTX660 and Maxwell GXT980), which are representative of the respective architectures, and to efficiently guide the tuning of two representative and widely used parallel applications.

The paper is organized as follows. Section 2 presents the related work. Section 3 presents the suite and how it is used to characterize a given device. Section 4 reports the experimental results, while Section 5 draws the conclusions.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

2. RELATED WORK

Different papers have been presented to evaluate functional and architectural characteristics of GPUs through microbenchmarking. Wong et al. [20] developed a microbenchmark suite to measure the CUDA-visible architectural characteristics of the NVIDIA GTX280 GPU. Such a measure includes some undisclosed characteristics of the processing elements and memory hierarchy.

Nishikawa et al. [13] evaluated the throughput and power efficiency of three 128-bit block ciphers (AES, Camellia, and SC2000) on an NVIDIA Kepler GTX680 and on AMD Radeon GCN HD7970. They compared such features with those provided by an NVIDIA GTX580 and AMD Radeon HD6770, whose architectures are one generation older. Yan et al. [21] proposed an OpenCL microbenchmark suite for GPUs and CPUs. They evaluated the performance characteristics of several components, such as, bus (bandwidth), memories, branch and thread scheduling on both a multicore X86 CPU and on different NVIDIA GPUs.

Fang et al. [4] proposed a microbenchmarking methodology based on short elapsed-time events (SETEs) to obtain memory microarchitectural details in multi- and many-core architectures. The methodology allows analysing, besides traditional cache/memory latency and off-chip bandwidth, the details of SW and HW prefetching units. Mei et al. [11] proposed a fine-grained benchmarking approach and they applied it on two popular GPUs (NVIDIA Fermi and Kepler) to expose previously unknown characteristics of the memory hierarchies.

Thoman et al. [18] proposed a suite of OpenCL microbenchmarks, which allows measuring functional characteristics of both GPUs and CPUs. Lemeire et al. [8] presented the most complete and comprehensive set of microbenhmarks among those proposed in literature, for both computational units and memory analysis.

Nevertheless, all these approaches have three main limitations. First, they are limited to *static* characteristics of GPUs. Indeed, as explained in Section 3, also the dynamic characteristics of a GPU are essential to understand how application bottlenecks involving selected functional components or underutilization of them can affect the code quality. Second, they do not cover all the functional components of the GPU devices. Third, they are sensitive to the compilation phase, which often makes the generated low-level code very inaccurate in measuring the GPU characteristics.

In this work, we focus on CUDA GPU microbenchmarking. OpenCL microbenchmarks (e.g., [18]) allow flexibility for computation on heterogeneous systems, but they cannot guarantee completeness and accuracy since such a programming model does not provide routines to directly interface with the low level features of the supported devices.

3. THE MICROBENCHMARK SUITE

A microbenchmark is a GPU kernel that exercises a specific functional component of the device and whose instructions can be evaluated at a clock-cycle accuracy. A microbenchmark main procedure consists of a long sequence of one or more selected instructions (e.g., arithmetic instructions, memory accesses) that executes without any interference deriving from other instructions. Each microbenchmark selectively stresses a functional component without or minimally affecting the others to provide reliable and accurate feedback. To do that, we implemented the microbenchmarks by combining common CUDA C/C++ language with inline intermediate assembly to avoid compiler side-effects.

The PTX language has been exploited to force a specific operation on a data type, to avoid compiler optimizations (which cannot be avoided by simply setting compiler flags like `-O0` in both C/C++ and PTX compilation), to prevent caching/local-storage mechanisms, and to restrict the memory access space.

Tables 1 and 2 summarize the GPU components and the corresponding low-level instructions statically and dynamically exercised by the proposed suite. The tables compare the completeness and the accuracy of the suite with the best and more complete suites at the state of the art (i.e., [8, 18]). The accuracy is essential for a correct characterization of the timing features of a GPU component, and even more to

Componet	Benchmark	Instructions	Thoman et al. [18]	Lemeire et al. [8]
ALU	**32-bit Integer Simple**	`add, sub`	✗	✓(ND)
ALU	**32-bit Integer Complex**	`mul`	✗	✓(ND)
ALU	**32-bit Integer Bit operations**	`clz, mbs, brev, bfi, bfe`	✗	✗
ALU	**32-bit Integer Shift**	`shl, shr`	✗	✗
ALU	**32-bit Integer Pop. count**	`popc`	✗	✗
ALU	**32-bit Integer Remainder**	`rem`	✗	✗
ALU	**64-bit Integer Simple**	`add, sub`	✗	✗
FPU	**32-bit FP Simple**	`add, sub, mul`	✓(74.0%)	✓(ND)
FPU	**32-bit FP Complex**	`div, div.ftz, div.approx, div.approx.ftz`	✓(85.8%)	✓(ND)
SFU	**32-bit FP Transcendental op.**	`sin, cos, exp, rsqrt, rcp, log`	✓(45.4%)	✓(ND)
DFU	**64-bit FP Simple**	`add, sub`	✓(74.1%)	✓(ND)
DRAM	**DRAM**	`load, store`	✗	✓(ND)
L2	**L2**	`load, store`	✗	✓(ND)
L1/Shared mem.	**Shared memory**	`load, store`	✗	✓(ND)
Constant mem.	**Constant memory**	`load, store`	✗	✓(ND)

Table 1: Microbenchmarks for static characteristics

Componet	Benchmark	Thoman et al. [18]	Lemeire et al. [8]
ALU	**Loop unrolling**	✗	✗
ALU	**ILP**	✗	✗
DRAM	**Coalescence**	✗	✗
DRAM	**Access size**	✗	✗
Shared memory	**Bank conflicts**	✗	✗
Streaming Multiprocessor	**Device occupancy (SM)**	✗	✗
SM scheduler	**Device synchronization**	✗	✗
SM scheduler	**Thread divergence**	✗	✗

Table 2: Microbenchmarks for dynamic characteristics

understand how a generic application can affect power and energy consumption of such a component. The accuracy is measured as the number of useful ("pure") instructions for a given component microbenchmarking over the total number of the microbenchmark instructions. Each microbenchmark of the proposed suite reaches an accuracy value equal to 99.99%. Such an accuracy is not reached by the counterparts (as reported in brackets). We derived the accuracy of the suite proposed by [8] experimentally (see Section 4), since the suite is not released with the source code.

3.1 GPU static characteristics

The suite allows analysing the peak characteristics of the arithmetic and memory components of the GPU by applying extensive workloads on them. The *arithmetic* microbenchmarks target the complete set of arithmetic instructions natively supported by the GPU, by distinguishing between integer and floating-point over 32 and 64-bit word sizes. The *memory* microbenchmarks give information on the throughput (bandwidth) of DRAM, L1/shared, constant, and L2 cache memories. The *DRAM microbenchmark* executes several accesses at different memory locations with a stride of 128 bytes between grid threads to avoid L1 cache interferences. The *L2 microbenchmark* repeats a compile-time sequence of store instructions on the same memory address. We used cache modifiers [14] to avoid L1 cache hits in the store operations. *Shared* and *constant memory microbenchmarks* consist of a sequence of store/load instructions.

3.2 GPU dynamic characteristics

The suite includes dynamic microbenchmarks, which analyse the dynamic characteristics of the device by exercising the functional components with different intensity.

The memory microbenchmarks analyse how the memory access pattern of threads affects the memory throughput. This includes memory coalescence, memory access size, and bank conflicts involved by the implemented access pattern. As an example, the microbenchmark in Fig. 2 measures the impact of global memory *coalescence* on the memory throughput. To do that, the microbenchmark implements different patterns of memory accesses,

```
   __device__ clock_t devClocks[RESIDENT_WARPS];
   __device__ int devTMP;
   __device__  volatile int devMemory[SIZE];
   template<int TH_GROUP_SIZE>
   __global__ Coalescence()
1: int thread_group_id = GLOBAL_THREAD_ID / TH_GROUP_SIZE;
2: int L1_bank_offset = thread_group_id · CACHE_LINE_SIZE;
3: volatile int* pointer = devMemory + L1_bank_offset +
   (GLOBAL_THREAD_ID%TH_GROUP_SIZE);
4: int R1 = threadIdx.x;                       // assign dynamic value
5: clock_t start_tm = clock64();
6: InstrSeq<N>(pointer, R1);                 // call the function N times
7: clock_t end_tm = clock64();
8: if (LANE_ID ==0) then devClocks[WARP_ID] = end_tm - start_tm;
9: if (THREAD_ID == 1024) then devTMP = R1; // never executed
```

```
   template<int N>                         // template metaprogramming
   __device__ __forceinline__ InstrSeq(volatile int* pointer, int& R1)
1: const int STRIDE = RESIDENT_WARPS · CACHE_LINE_SIZE;
2: #pragma unroll                                  // loop unrolling
3: for (int i = 0; i < 4096; i++) do
4:     asm volatile("ld.volatile.s32 %0, [%1]" : "=r"(R1) :
5:                 "l"(pointer + i * STRIDE) : "memory");
6: end
7: InstrSeq<N-1>(pointer, R1);                      // recursive call
```

FIG. 2: *Example of the microbenchmark code to measure the impact of global memory coalescence.*

where each pattern guarantees a different coalescence degree. Considering a base address (`devMemory`), each thread calculates the own *L1 bank offset* through the global identifier (`GLOBAL_THREAD_ID`), the size of the L1 cache bank (`CACHE_LINE_SIZE`), and through the `TH_GROUP_SIZE` variable (lines 1, 2 in the upperside of Fig. 2). This allows forcing different thread accesses to be grouped into the same L1 cache banks and, as a consequence, to group such thread accesses into coalesced global memory transactions. Then, each thread calculates the final pointer through base address, L1 bank offset, and thread offset in the bank. Fig. 3 shows an example, which underlines how grouping threads into coalesced transactions is parametrized through the `TH_GROUP_SIZE` variable. The microbenchmark dynamically sets such a variable to control the coalescence degree.

The code implements *volatile* quantifiers (`devMemory` definition and line 3 in the upper side of Fig. 2). This allows avoiding *local-storage* optimizations by the compiler, which may change the coalescence degree forced by the proposed strategy. The code adopts *recursive* and *template-based* meta-programming to generate an arbitrarily long sequence of arithmetic instructions ($N \times 4,096$ `store` instructions in the example). This allows improving the accuracy of the functional characteristics measurement. In the bottom side of Fig. 2, the `STRIDE` variable represents the minimum value of memory address offset that allows preventing false positive L1 cache hits. Such an offset guarantees that any thread cannot calculate the same pointer in two different loop iterations. Both the `STRIDE` and the global memory offsets (`i`·`STRIDE`) are computed at compile time to guarantee that the measured memory throughput is not distorted by such value computation.

The *arithmetic* microbenchmarks analyse the dynamic characteristics of the GPU computational units over two optimization aspects: unrolling and instruction-level parallelism (ILP). The *unrolling* microbenchmark iteratively executes a chunck of code in a loop, where the loop iterations/loop unrollings are set dynamically and increasingly from the minimum to the maximum. The microbenchmark returns the effect of eliminating conditional statements in terms of performance and power. The *ILP* microbenchmark executes a sequence of unrelated instructions, where the sequence length is set dynamically and incrementally.

The suite includes the *shared memory* microbenchmark, which generates a different amount of bank conflicts, from zero to the maximum value. The *access size* microbenchmark copies one large array into another multiple times by varying, at each iteration, the size of the data block.

The *thread scheduling and synchronization* microbenchmarks aim at studying the dynamic behavior of the device

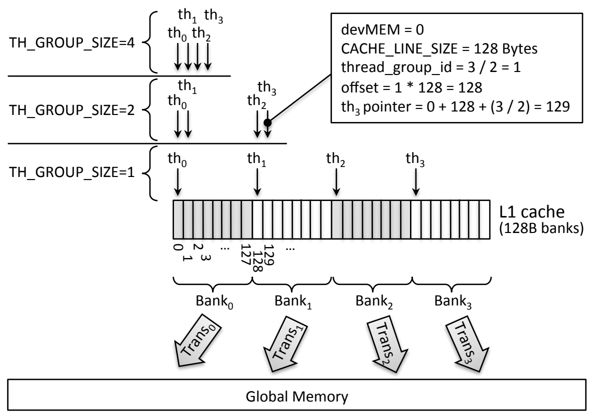


FIG. 3: *Controlled memory coalescence*

by varying the streaming multiprocessor occupancy and the degree of thread divergence. They also aim at characterizing the device by considering the synchronization overhead caused by thread block barriers over the whole kernel. The *device occupancy (SM)* microbenchmark evaluates the contribution of different number of active SMs on the computation, while the *device synchronization* one analyses the impact of synchronization barriers in the code.

3.3 GPU Device Characterization

We run the microbenchmarks on an NVIDIA GeForce GTX660 and on a GTX980, which are representative of the Kepler and Maxwell architectures, respectively.

The devices have been evaluated with CUDA Toolkit 7.5, AMD Phenom II X6 1055T (3GHz) host processor, and Ubuntu 14.04 operating system. The proposed suite is independent from the specific CUDA-enabled GPU device and from the adopted CUDA Toolkit version.

Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements.

Power and energy consumption information has been collected through the *Powermon2* power monitoring device [2]. The analysis has been performed with the default GPU frequency setting and by disabling any PCI/GPU adaptive frequency or thermal throttling mechanisms (i.e., *GPUBoost*).

Table 3 reports the results obtained by running the static microbenchmarks on the GTX980 device. For the sake of space and without loss of generality, we do not report the *static* characteristics of the GTX660 device, since they are not necessary for understanding the focus of the paper. We only report, for the GTX660, the most useful *dynamic* benchmarking results.

The static microbenchmarks are organized over columns and consist of 10^9 instructions per SM. For each microbenchmark, the table reports the execution time, the theoretical peak throughput of the corresponding functional unit provided in the device specifications [15] (*Spec. throughput*) and that measured through the proposed microbenchmark (*Real throughput*). The static microbenchmarks measure the maximum arithmetic instruction throughput of simple integer operations (`add`, `sub` , etc.), complex integer operations (`mul`, `mad`, etc.), integer population count, shift, remainder, bitwise (bit insert, bit reverse, etc.), simple single precision floating-point operations (`add`, `mul`, etc.), complex single precision floating-point operations (transcendental functions such as `sin`, `rcp`, etc.) and double precision floating-point operations. The device specifications do not include the theoretical peak throughput of the integer 64-bit and integer 32-bit remain operation (`rem`) and integer 32-bit complex operation since such operations have not an embedded hardware implementation. They are performed through a combination of different hardware units.

The results of the static microbenchmarking allow understanding the microbenchmark accuracy by comparing the measured throughput with the throughput reported in the

	Integer 32-bit						Integer 64-bit	FP 32-bit		FP 64-bit
	Simple	Complex	Pop. Count	Shift	Bit Op.	Rem	Simple	Simple	Special	Simple
Execution Time (ms)	8.6	31.5	29.6	15.5	15.5	965.2	18.8	8.6	32.5	223.9
Spec Throughput OPs per Cycle per SM	128	n.a.	32	64	64	n.a.	n.a.	128	32	4
Real Throughput OPs per Cycle per SM	116.3 (66.3†)	31.8	32.0	63.4	63.5	1.0	51.6	116.3 (101.3*) (104.4†)	29.8 (33.9*)	4.0 (7.6*) (err†)
Avg. Power (W)	75.2	88.2	69.4	70.8	79.1	100.2	80.9	72.2	84.4	76.8
Max Power (W)	86	93	72	77	85	114	88	86	99	88
Energy (J)	0.7	2.8	2.1	1.1	1.2	96.7	1.5	0.6	2.74	17.20
Energy efficiency MIPS per Watt	26,564	6,190	8,370	15,672	13,995	178	11,269	28,120	6,262	999
nano Joule per instruction	0.04	0.16	0.12	0.06	0.07	5.63	0.09	0.04	0.16	1.0

Table 3: *GTX980 - Characterization of arithmetic instructions (static characteristics).* **results of Thoman et al.[18],* †*results of Lemeire et al.[8]*

	Dram	L2	Shared	Constant
Exec Time (ms)	9,536.2	2,522.5	847.6	226.4
Real Throughput (Trans$\times$Cycle)$\times$SM	0.09 (0.08†)	0.33 (0.14†)	1.02 (0.85†)	4.06 (3.08†)
Avg. power (W)	113.3	105.8	94.1	76.0
Max power (W)	117	112	105	87
Energy (J)	67.54	16.7	5.0	1.1
Energy efficiency (10^6Transactions/Watt)	15.8	64.4	215.3	998.0
nano Joule per mem. transaction	62.9	15.5	4.6	1.0

Table 4: *Characteristics of mem. accesses on the GTX980.* †*results of Lemeire et al.[8]*

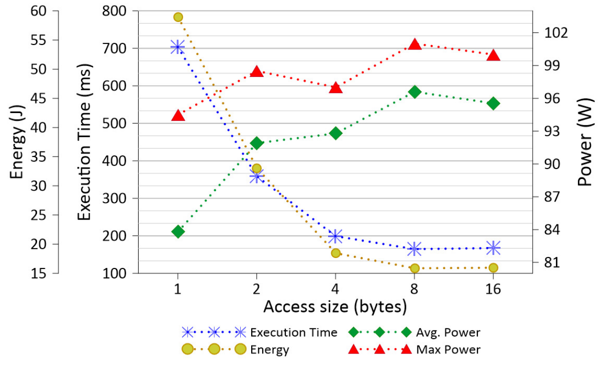


FIG. 4: *GTX980 DRAM access size*

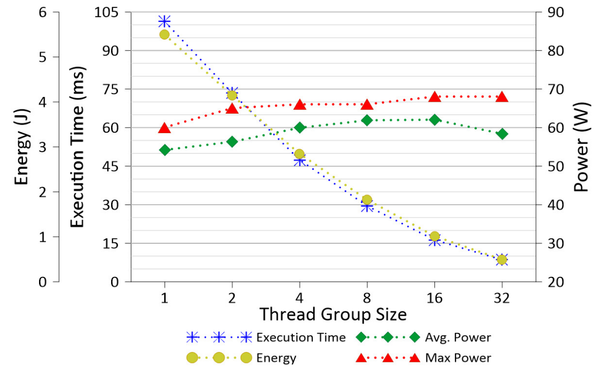


FIG. 5: *GTX980 thread divergence*

device specification.

They also underline the accuracy difference between each microbenchmark of the proposed suite and the corresponding microbenchmark, when provided, of the best suites at the state of the art (i.e.,[18, 8]). It is worth noting that the accuracy of the state of the art microbenchmarks for simple instructions is very low. This is due to the compiler activity on the source code (unavoidable even disabling any optimization flag), which inserts "spurious" instructions in the executable code. Such optimizations lead the throughput measured on the FP instructions to be even higher than the real throughput

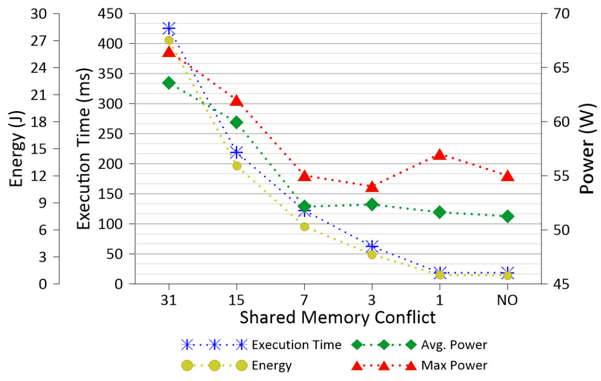
Table 3 also reports information about power and energy consumption, which is not reported in the device specifications. The energy efficiency (or performance per watt)[5] is defined as the number of operations/instructions computed per second per Watt. We refer to million instructions per second (MIPS) for arithmetic benchmarks and million of memory transactions for memory benchmarks. Finally, the table shows the power consumption (nJ) per single instruction/memory transaction.

Table 4 reports the results of the static microbenchmarks on the GPU memories. They allow understanding how the throughput differs among memories. As an example, an application running on the GTX980 accesses the shared memory 11 times faster than the DRAM (the corresponding microbenchmarking on the GTX660 reports that the same application running on the GTX660 accesses the shared memory 5 times faster than the DRAM). The DRAM accesses strongly affect the average and peak power. The constant memory, which presents the best energy efficiency, is 3.3 times more energy efficient than the L2 cache in the GTX980 (1.5 times in the GTX660). The microbenchmarks of the state of the art suites (i.e.,[18, 8]) do not allow measuring power and energy consumption since they are too fast also for the best sampling frequency available in literature provided by the *Powermon2* device.

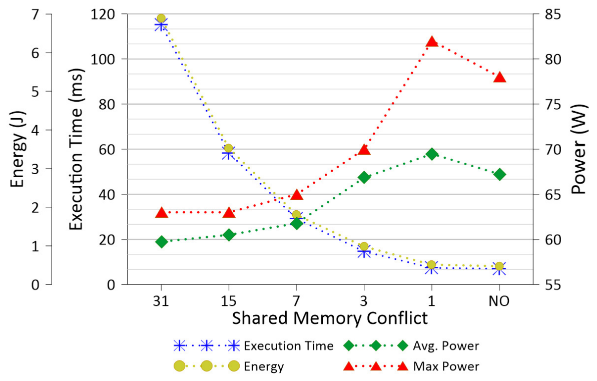
Figures 4-6 report some of the dynamic microbenchmarking results (the most relevant for the case study presented in Section 4). Fig. 4 reports the impact of the thread access size in DRAM, starting from 1-byte to 16-byte blocks per thread. Increasing the access size sensibly improves both performance and energy consumption at the cost of slightly more average and peak power in the GTX980. The results are proportionately similar in the GTX660.

Fig. 5 reports the analysis of thread divergence. Performance and energy consumption linearly improve by moving from the maximum divergence (1-sized thread groups in the left) to the minimum divergence (32-sized thread groups in the right), at the cost of a slight increase of peak power.

Figures 6a and 6b quantify the effect of bank conflicts in shared memory for both the GTX660 and GTX980. They underline that the bank conflicts similarly impact on performance and energy on the two devices, while they affect average and peak power in the opposite way. In the GTX660, average and max power sensibly decrease (up to 20%) by decreasing the bank conflicts from the maximum (31) to 7. After that (from 7 to 0) there is no meaningful effect on them. In the GTX980, there are no meaningful variations on the power by decreasing the bank conflicts from the max-



(a) *GTX660*



(b) *GTX980*

FIG. 6: *Shared memory bank conflicts*

imum to 7, while further reducing the conflicts (from 7 to 0) involves the most sensible power increase. This is due to the advanced instruction scheduler of the Maxwell architecture that, differently from that in the GTX660, keeps up the throughput of few or no bank conflicts.

The *memory coalescence* microbenchmark analyses the impact of the coalescence in DRAM memory accesses on the device performance, power, and energy consumption. The results, which are not reported for the sake of space, quantitatively show that performance, power, and energy are linearly proportional to the coalescence degree.

4. EXPERIMENTAL RESULTS

We used the proposed microbenchmarks combined with the standard profiler information to drive the tuning of two widespread parallel applications, *vector reduction* and *matrix transpose*. Each application tuning has been performed for both the GTX660 and GTX980 devices.

4.1 Vector Reduction

Vector reduction is one of the most common and important application cores in parallel computing. It consists of performing a binary and associative operation over all elements of a data vector to obtain a single final value.

We started from the implementation presented in [3], which applies, as associative operator, the addition to a vector of integers. Our goal was to generate two distinct variants of the original code, the first (branch 1) targeting a lower peak power, the second (branch 2) targeting performance speedup.

For the first branch, the best code optimization we identified to lower the peak power without losing performance was reimplementing the following code pattern to control the thread execution paths:

```
if (threadIdx.x % (2 * stride) == 0)
    Mem[threadIdx.x]+=Mem[threadIdx.x+stride];
```

The idea was to replace the `rem` PTX operations used in the original code with `add` and `mul` PTX operations (see Table 3 for the peak power comparison among such arithmetic instructions). On the other hand, the identified modification potentially reduces the *thread divergence*, since it forces only the neighbouring threads to compute the reduction body (second line of the code above). However, by analysing the microbenchmark results on the thread divergence (Fig. 5 for the GTX980, and similar for the GTX660), we expected no meaningful increase of peak power as a *side-effect* of any thread divergence reduction.

According to the results of such an analysis, we expected slightly higher performance and much lower peak power in this code version w.r.t. the original code in both the devices. We also expected a stronger peak power reduction and a lower speedup in the GTX660 while a weaker power reduction and a higher speedup in the GTX980, as then confirmed by the results (-24% peak power and 1.4x speedup in GTX660, -20% peak power and 2.1x speedup in GTX980). Table 5 summarizes the obtained results and reports, for each code version, the used profiler metrics, the most relevant features that characterize the code, the analysed microbenchmarks, the execution time, the average and peak power, and the energy consumption.

In the other optimization branch (v2.x) we identified a different memory access strategy [9], which allows applying a variety of memory access sizes in the application. We thus analysed the microbenchmark results on the memory access sizes (Fig. 4) and observed that increasing the access size can lead to a sensible performance improvement with no meaningful side-effects on peak power. We applied the technique proposed in [15] to cast the inputs to a data type of larger size, thus forcing vectorized memory accesses. The technique improved both the performance and the peak power w.r.t. the original code and, as expected, the increasing of memory access size led to a further speedup with no significant power increase in both the devices.

4.2 Matrix Transpose

We analyzed the matrix transpose implementation provided in [16] and, by considering the profiling information, we identified two optimization branches targeting memory coalescence for global memory accesses and bank conflict reduction for shared memory accesses, respectively. Coalescence and memory access pattern are two of the most important factors to be considered in the tuning phase of any GPU application, especially if the application, like the matrix transpose, is memory-bound.

In particular, the memory coalescence optimization allows sensibly improving the performance speedup, while the bank conflict reductions allows tuning the tradeoff between performance and peak power (see Fig. 6).

By combining the profiling information of the original code, which underlined a low global memory accesses efficiency (`global_mem_eff`=0.125) and the dynamic characteristics of the memory coalescence provided by the corresponding microbenchmark, we expected, for the first branch, an increase of both performance speedup and peak power proportional to the memory coalescence improvement on both the devices. We thus optimized the memory coalescence by re-organizing the thread block configuration in v1.0. Table 6 shows the results, which confirm the expected positive speedup at the cost of a higher peak power (5.6x speedup and +10% peak power in GTX660, 2.1x speedup and +12% peak power in GTX980).

In the other branch (v2.x), we identified a different memory access pattern, which allows taking advantage of the shared memory to locally transpose a tile of the whole matrix and to optimize the memory load/store operations of the matrix elements. The implementation of such an optimization (v2.1) provided a further tuning opportunity, since the profiling of such a code version indicated bad access patterns in shared memory. This was underlined by a low value of `shared_mem_eff`[1], which involves a waste of the memory bandwidth. By analysing the results of the mi-

[1] A value of `shared_mem_eff` equal to 6% corresponds to 31 bank conflicts in the shared memory microbenchmark (see Figs. 6a,6b):

$$\frac{1\ access}{total\_accesses{=}32} \cdot \frac{smem\_bank\_size{=}8byte}{data\_size{=}4byte} = 6\%.$$

Opt. branch	Ver.	Profiler metrics and features	Related microbenchmark	GTX660 Exec. time (ms)	GTX660 Avg. power (W)	GTX660 Peak power (W)	GTX660 Energy (J)	GTX980 Exec. time (ms)	GTX980 Avg. power (W)	GTX980 Peak power (W)	GTX980 Energy (J)
-	orig.	`inst_per_warp`= 347 integer_rem	-	260.8	83.2	101	21.7	108.7	165.3	169	17.9
1	v1.0	`inst_per_warp`= 130	Divergence Arith. throughput	184.7	73.7	77	13.6	52.6	167.5	135	6.7
2	v2.0	`gmem_throughput=75GB/s` (GTX660) `gmem_throughput=145GB/s` (GTX980)	Access size	37.4	77.7	82	2.9	17.2	132.2	151	2.1
	v2.1	`gmem_throughput=111GB/s` (GTX660) `gmem_throughput=165GB/s` (GTX980)	Access size	23.2	77.1	85	1.7	15.2	133.5	153	1.6

TABLE 5: *Vector reduction characteristics on GTX660 and GTX980 devices.*

Opt. branch	ID	Profiler metrics and features	Related microbenchmark	GTX660 Exec. time (ms)	GTX660 Avg. power (W)	GTX660 Peak power (W)	GTX660 Energy (J)	GTX980 Exec. time (ms)	GTX980 Avg. power (W)	GTX980 Peak power (W)	GTX980 Energy (J)
-	orig.	`global_mem_eff`= 0.125 `ipc`= 0.3 (GTX660) `ipc`= 0.4 (GTX980)	-	52.2	75.0	79	3.9	55.8	86.5	92	4.8
1	v1.0	`global_mem_eff`= 0.5 `ipc`= 0.6 (GTX660) `ipc`= 0.6 (GTX980)	DRAM Coalescence	9.3	69.3	87	0.6	26.8	94.2	103	2.5
2	v2.0	`shared_mem_eff`= 6%	Bank conflicts	10.4	69.5	81	0.7	16.3	90.8	97	1.4
	v2.1	`shared_mem_eff`= 100%	Bank conflicts	6.3	61.4	74	0.4	10.4	90.2	105	0.9

TABLE 6: *Matrix transpose characteristics on the GTX660 and GTX980 devices.*

crobenchmarks on the bank conflicts (Figs. 6a and 6b), we expected, as a consequence of solving such a bottleneck, an improvement on both performance and peak power in the GTX660. We thus applied the *memory padding* technique [16] to reduce the bank conflicts, which led to 8.3x speedup and -5% peak power w.r.t the original version. As suggested by the microbenchmarks, we would not have applied such a time consuming optimization to reduce the peak power in the GTX980. The uselessness of such an optimization on the GTX980 has been then confirmed by the experimental results (+14% peak power w.r.t. the original version).

5. CONCLUSIONS

This paper presented a suite of microbenchmarks to statically and dynamically characterize GPU devices in terms of performance, power, and energy consumption. The paper showed how the microbenchmark results can been combined with the standard profiler information to efficiently tune any parallel application for a given GPU device and for a given design constraint (performance speedup or peak power reduction). Experimental results have been conducted on two widespread parallel applications for two representative GPU devices. They showed how the proposed microbenchmarking can improve the efficiency of the tuning task by identifying the potential from the useless optimization strategies before implementing them.

6. REFERENCES

[1] NVIDIA Tegra X1. http://www.nvidia.com/object/tegra.html.

[2] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proc. of IEEE SoutheastCon*, pages 479–484, 2010.

[3] J. Cheng, M. Grossman, and T. McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[4] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization*, 11(4):art. n.5, 2015.

[5] W.-c. Feng and K. Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[6] S. Hong and H. Kim. An integrated gpu power and performance model. In *Proc. of ACM/IEEE ISCA*, pages 280–289, 2010.

[7] T. Komoda, S. Hayashi, S. Miwa, and H. Nakamura. Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping. In *Proc. of IEEE ICCD*, pages 349–356, 2013.

[8] J. Lemeire, J. G. Cornelis, and L. Segers. Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model. In *Euromicro PDP*, pages 456–463, 2016.

[9] J. Luitjens. Faster Parallel Reductions on Kepler, 2014. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[10] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Proc. of IEEE ICPP*, pages 48–57, 2012.

[11] X. Mei, K. Zhao, C. Liu, and X. Chu. Benchmarking the memory hierarchy of modern GPUs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8707 LNCS:144–156, 2014.

[12] S. Mittal and J. S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, Aug. 2014.

[13] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa. Throughput and power efficiency evaluation of block ciphers on kepler and GCN GPUs using micro-benchmark analysis. *IEICE Transactions on Information and Systems*, E97-D(6):1506–1515, 2014.

[14] NVIDIA. PTX: Parallel Thread Execution ISA, 2015. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[15] Nvidia CUDA. Programming guide, 2015. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[16] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. *Nvidia CUDA SDK Application Note*, 28, 2009.

[17] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proc. of ACM SIGPLAN PPoPP*, pages 11–22, 2012.

[18] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer. Automatic opencl device characterization: guiding optimized kernel design. In *European Conf. on Parallel Processing*, pages 438–452. Springer, 2011.

[19] Y. Wang, S. Roy, and N. Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proc. of ACM/IEEE DATE*, pages 300–303, 2012.

[20] H. Wong, M.-M. Papadopoulou, M. Sadooghi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of IEEE ISPASS*, pages 235–246, 2010.

[21] X. Yan, X. Shi, L. Wang, and H. Yang. An OpenCL micro-benchmark suite for GPUs and CPUs. *Journal of Supercomputing*, 69(2):693–713, 2014.

[22] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Fixing performance bugs: An empirical study of open-source GPGPU programs. In *Proc. of IEEE ICPP*, pages 329–339, 2012.