A Fine-grained Performance Model for GPU Architectures

Nicola Bombieri Federico Busato Franco Fummi
Dept. of Computer Science
University of Verona - Italy
Email: name.surname@univr.it

***Abstract*—The increasing programmability, performance, and cost/effectiveness of GPUs have led to a widespread use of such many-core architectures to accelerate general purpose applications. Nevertheless, tuning applications to efficiently exploit the GPU potentiality is a very challenging task, especially for inexperienced programmers. This is due to the difficulty of developing a SW application for the specific GPU architectural configuration, which includes managing the memory hierarchy and optimizing the execution of thousands of concurrent threads while maintaining the semantic correctness of the application. Even though several profiling tools exist, which provide programmers with a large number of metrics and measurements, it is often difficult to interpret such information for effectively tuning the application. This paper presents a performance model that allows accurately estimating the potential performance of the application under tuning on a given GPU device and, at the same time, it provides programmers with interpretable profiling hints. The paper shows the results obtained by applying the proposed model for profiling commonly used primitives and real codes.**

I. Introduction

Even though graphics processing units (GPUs) are increasingly adopted to run general purpose applications in several domains, programming such many-core architectures is a challenging task [1], [2]. Even more challenging is efficiently tuning applications to fully take advantage of the GPU architectural configuration. Bottlenecks of a GPU application such as high thread divergence or poor memory coalescing have a different impact on the overall performance depending on which GPU device the application is run [3].

Different profiling tools (e.g., CUDA nvprof, AMD APP) have been proposed to help programmers in the application development and analysis. Nevertheless, interpreting the large number of metrics and measurements they provide to improve the application performance is often difficult or even prohibitive for inexperienced programmers.

This paper proposes a comprehensive and accurate performance model for GPU architectures, which aims at supporting programmers in the development and tuning of GPU applications. The model relies on two concepts, *microbenchmarks* and *optimization criteria*. The microbenchmarks consist of specialized chunks of GPU code that have been developed to (i) exercise specific functional components of the device (e.g., arithmetic instruction units, shared memory, cache, DRAM, etc.) and (ii) measure the actual characteristics of such components (i.e., throughput, latency, delay). The result of the measure allows the model to weigh the performance prediction by considering the GPU architecture configuration.

The optimization criteria aim at quantitatively expressing the quality of a given application to exploit the potential of a specific GPU device (e.g., strong coalescence in memory accesses) as well as identifying causes of performance bottlenecks (e.g., high thread divergence, workload unbalancing). At the same time, they aim at guiding the application developer during the tuning activity through understandable hints, by selectively pointing out the causes of such bottlenecks.

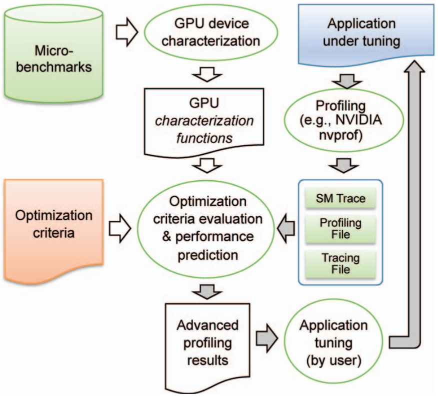

Micro-benchmarks
GPU device characterization
Application under tuning
GPU *characterization functions*
Profiling (e.g., NVIDIA nvprof)
Optimization criteria
Optimization criteria evaluation & performance prediction
SM Trace
Profiling File
Tracing File
Advanced profiling results
Application tuning (by user)

FIG. 1: *Overview of proposed model components and application.*

Figure 1 shows an overview of the proposed model application for tuning a GPU application. First, the microbenchmarks are run on the GPU device to extrapolate the *characterization functions*, i.e., dynamic characteristics of the functional components of the device. Then, the application under tuning is profiled through a standard profiling tool and the resulting information is combined with the characterization functions to measure how much the given application satisfies the optimization criteria. The resulting information provides the actual quality level and the potential improvement of each optimization criterion, the impact of each improvement on the overall application performance, and the overall potential performance of the application under tuning. The model allows the flow (underlined by grey arrows in Figure 1) to be iterated for incremental tuning of the application.

The paper presents the results obtained by applying the proposed model for tuning different GPU applications, by underlining how the advanced profiling results have been effectively used to focus the tuning effort in specific code optimizations.

The paper is organized as follows. Section II presents the related work. Section III presents the microbenchmark concept and how they have been developed to support the proposed model. Section IV presents the optimization criteria definition and evaluation. Section V explains how all the information are combined to predict the application performance, while Section VI reports the experimental results. Section VII is devoted to the concluding remarks.

II. Related work

Different performance models for GPU architectures have been proposed in literature. They can be classified into *specific models*, which apply on a particular application or pattern

978-3-9815370-6-2/DATE16/©2016 EDAA

only, and *general-purpose models*, which are applicable to any program/kernel for a comprehensive profiling [4].

In the class of specific models, [5] proposes an approach for performance analysis of *code regions* in CUDA kernels, while, in [6], the authors focus on profiling divergences in GPU applications. A different analytical approach is proposed in [7], which aims at predicting the kernel execution times of sparse matrix-vector multiplications.

General-purpose models allow profiling applications from more optimization criteria point of view and, thus, they can give different hints on how to optimize the code. As an example, [8] proposes a model for NVIDIA GPUs based on two different metrics: Memory warp parallelism (MWP) and computation warp parallelism (CWP). Although the model predicts the execution costs fairly well, the understanding of performance bottlenecks from the model is not so straightforward. This model has been extended in two different ways [9], [3]. [9] introduces two kernel behaviors, MAX and SUM, and shows how they allow generating predictions close to the real measurements. Nevertheless, they do not provide clues as how to choose the right one for a given kernel. In contrast, [3] extends the model with additional metrics, such as cache effect and SFU instructions.

All these analytical performance models, although accurate in several cases, rely on simulators (e.g., Ocelot, GPGPU, Barra) to collect necessary information for profiling, which implies a high overhead in the profiling phase. An attempt has been made in [10] for collecting more efficiently information on the GPU characteristics and using simple static analysis methods to reduce the overhead of runtime profiling.

Besides the often prohibitive overhead introduced in the profiling phase, especially for complex applications, a big problem of the simulator-based models is portability. They can be applied to profile applications on GPU models that are supported by the simulator, which, often, is not updated to the last releases of GPU models.

Differently from the analytical models, [11] and [12] are based on machine-learning techniques, which allow identifying hardware features and using feature selection, clustering and regression techniques to estimate the execution times. Nevertheless, both these models are inaccurate, thus providing approximate estimations with high variability.

We propose a general-purpose, fine-grained, performance model that, similarly to the machine-learning models, relies on microbenchmarks to characterize the device and on several application criteria to measure the implementation quality, gives interpretable hints and accurate performance prediction.

III. MICROBENCHMARKS

Each microbenchmark is developed with the aim of satisfying the following properties:

P1: Stressing capability. The microbenchmark applies heavy and extensive workloads to the selected functional component. This allows reaching the fully work-loaded steady state of the component and measuring the real (vs. theoretical) peak performance, while minimizing any side effect that may incur during the measurement, as described in Section III-A. This includes measuring the real latency of memory accesses for each memory level (i.e., registers, shared, L1 cache, L2 cache, and global memory).

P2: Intensity variability. The microbenchmark must exercise the functional component with different intensity. This allows predicting the effect of improving an optimization criterion on the application performance (which is generally not linear), as explained in Section III-B.

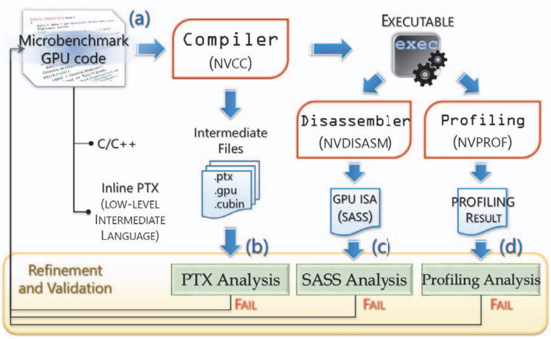


FIG. 2: *Microbenchmark development. (a) Code writing, (b) Compilation and PTX analysis, (c) Disassembling and GPU ISA Analisys, (d) Profiling analysis.*

P3: Selectivity. The microbenchmark exercises, as much as possible, only a specific functional component of the GPU device. This allows us to selectively associate the microbenchmark to a specific optimization criterion, as described in Section IV.

P4: Portability. The microbenchmark is developed independently from any model or configuration of GPU architecture.

A. Microbenchmark Development

Figure 2 shows the design process of a microbenchmark. The microbenchmark code is written with the aim of satisfying the four properties presented above. Since the compiler may optimize such a code (e.g., dead code elimination, code-block reordering etc.) and, through the consequent side effects it may elude the target properties, the code is checked and refined at different steps along the compilation process.

First, the code is written by combining CUDA C/C++ and inline PTX [13] languages (Figure 2(a)). The PTX (intermediate) assembly statements allow the code to be compilable for any device model (property $P4$) and, at the same time, to prevent *high-level* compiler optimizations. As an example, Figure 3 shows the microbenchmark code developed to measure the peak throughput of an integer arithmetic operation (i.e., `add`) through a long sequence of the arithmetic instructions with no interrupt or intermediate operation. The code implements dynamic value assignments to registers (see rows 1 and 2 in the upper side of the figure) to avoid the *constant propagation* optimization by the compiler[1]. The code also adopts *recursive* and *template-based* metaprogramming. This allows generating an arbitrarily long sequence of arithmetic instructions ($8,191 \times N$ `add` instructions in the example[2]).

As a second level checking, the code is compiled to generate both the intermediate file (Figure 2(b)) and the executable binary file. The intermediate PTX file is analysed to verify whether the target properties still hold after the higher-level compilation step. If not, the microbenchmark code undergoes a refinement iteration. Once verified, the code undergoes a more accurate check, whereby the executable binary file is disassembled in the native ISA code, called SASS (Shader ASSembly), as shown in Figure 2(c). This allows checking the properties after the lower-level compilation step.

Finally, the microbenchmark is validated through the profiler (Figure 2(d)) to ensure that it exercises only the target functional component.

[1] Static value assignments to registers are generally solved and substituted by the compiler optimizations through inlining operations.

[2] In the example, 8,191 is the maximum number of unrolling iterations the pragma `unroll` supports. After that, the compiler would insert control statements for the loop. Such a limitation is overcome through recursive calls (row 5 of the template in Figure 2).

```
__global__ ADD_THROUGHPUT()

int R1 = clock();   // assign dynamic values to R1,R2 to↵
int R2 = clock();   // avoid constant propagation
int startTimer = clock();
Computation<N>(R1, R2);   // call the function N times
int endTimer = clock();
```

```
template<int N>()   // template metaprogramming
__device__ __forceinline__ COMPUTATION(int R1, int R2)

#pragma unroll 8191   // maximum allowed unrolling
for (int i = 0; i < 8191; i++) do
    asm volatile(”add.s32 : ”=r”(R1) : ”r”(R1), ”r”(R2));
end   // volatile: prevent ptx compiler optimization
Computation<N-1>(R1, R2);   // recursive call
```

FIG. 3: *Example of microbenchmark code. The code aims at measuring the maximum instruction throughput of the* `add` *operation.*

B. GPU Device Characterization through Microbenchmarks

Similarly to the example of Figure 3, the developed microbenchmarks are applied to measure the peak performance of all arithmetic operations and of the memory at different hierarchy levels. In particular, they measure the maximum arithmetic instruction throughput of integer operations (`add`, `mul`, comparison, bitwise, etc.), simple single precision floating-point operations (`add`, `mul`, etc.), complex single precision floating-point operations (`sin`, `rcp`, etc.) and double precision floating-point operations.

Microbenchmarks are also applied to exercise the functional components with different intensity. As an example, the shared memory throughput is analysed by running a microbenchmark that generates a different amount of bank conflicts, from zero to the maximum value, and measures the corresponding access times. The effect of the bank conflicts over the access time is then represented by a *characterization function* through a sampling, quantization, and interpolation process [14]. The characterization function strongly depends on the device architecture.

Overall, in the proposed model, microbenchmarks are applied to extrapolate:

- $\digamma_{Divergence}$ as the function that characterizes the effect of the thread divergence on performance. It is obtained through a microbenchmark that incrementally increases the percentage of control statements in the code.
- $\digamma_{DRAMThr}$ as the function that characterizes the effect of the (under)utilization of the global memory bandwidth on performance. It is obtained through a microbenchmark that incrementally exercises the memory bus through different amount of exchanged data.
- $\digamma_{ARITHThr}$ as the function that characterizes the effect of the (under)utilization of the arithmetic units on performance.
- $\digamma_{SHMEM}$ as explained in the example above.

The characterization functions are used as parameters in the evaluation of the optimization criteria targeting the divergence, the throughput/occupancy, and the shared memory efficiency, respectively, as explained in Section IV-G.

Finally, microbenchmarks have been defined to calculate the percentage of use of each functional components (i.e., shared memory, L1 and L2 caches, DRAM, arithmetic units) during an application run. They are used to calculate the application potential speedups, as explained in Section V.

IV. Optimization Criteria

The optimization criteria are defined to cover all the crucial properties a GPU application should satisfy to exploit the full potential of the GPU device. We consider the properties adopted in [3], [5], [6], [8], [9] concerning divergence, memory coalescing, and load balancing. In addition, we define optimization criteria to cover synchronization issues. Differently from the literature, the proposed criteria are more accurate (i.e., fine-grained) to evaluate such properties. All the criteria are defined in terms of events and static information, which are all provided by any standard GPU profiler. Each criterion value is expressed in the range [0, 1], where 0 represents the worst and 1 represents the best evaluation of such an optimization.

A. Host Synchronization

Many complex parallel applications organize the compute-intensive work into several functions offloaded to GPUs through host-side kernel calls. Depending on the code complexity and on the workflow scheduling, this mechanism may involve significant overhead that can compromise the overall application performance. The host synchronization criterion aims at evaluating the amount of time spent to coordinate the kernel calls. It is defined as follows:

$$\textsc{HostSync} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{KernelExeTime_i}}{\mathrm{KernelStart}_N + \mathrm{KernelExeTime}_N - \mathrm{KernelStart}_1}$$

where N is the number of kernels in which the application has been organized, $KernelExeTime_i$ is the real execution time of kernel i on the device, and $KernelStart_i$ is the clock time in which kernel i starts executing.

This criterion helps programmers to understand if the overall application speedup is bounded by an excessive host synchronization activity. Merging different kernels, using inter-block synchronization [15] or reducing small memory transfers improve the quality value of this criterion.

B. Device Synchronization

In GPU computing, the synchronizations of threads in blocks are one of the main causes of idle state and, thus, they strongly impact on the application performance. Beside introducing overhead in the kernel execution, they also limit the efficiency of the multiprocessors in the warp scheduling activity. This criterion gives a quality value of a kernel by measuring the total time spent by the kernel for synchronizing thread blocks:

$$\textsc{DeviceSync} = 1 - \mathrm{StallSync}$$

where $StallSync$ represents the percentage of the GPU time spent in synchronization stalls over the total number of stalls. $StallSync$ depends on the load balancing among threads as well as the number of synchronization points (i.e. thread barriers) in the kernel.

C. Thread Divergence

Branch conditions that lead threads of the same warp to execute different paths (i.e., thread divergence) are one of the main causes of inefficiency of a GPU kernel. This criterion evaluates the thread divergence of a kernel as follows:

$$\textsc{Divergence} = \frac{\#\mathrm{ExeInstructions}}{\#\mathrm{PotExeInstructions}} \times \digamma_{\mathrm{Divergence}}$$

where $\#ExeInstructions$ represents the total number of instructions executed by the threads of a warp and $\#PotExeInstructions$ represents the total number of instructions potentially executable by the threads of a warp. The final value is calculated as the average over all warps run by the kernel. The value is weighted by the characterization function $\digamma_{Divergence}$ presented in Section III-B.

D. Warp Load Balancing

This criterion expresses how well the workload is uniformly distributed over the cores of each single SM:

$$\textsc{LoadBalanc}_{\text{WARP}} = \frac{\left(\frac{\text{TotActiveWarps}}{\text{TotActiveCycles}}\right)}{\left(\frac{\text{BLOCKSIZE}}{32}\right)\cdot \#\text{Blocks\_per\_SM}}$$

where $TotActiveCycles$ represents the total number of clock cycles in which the single SMs are not in idle state. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The denominator represents the theoretical maximum occupancy of the SMs in terms of number of warps. It is calculated by considering the block size and the number of blocks mapped to each single SM.

E. Streaming Multiprocessor (SM) Load Balancing

Besides the load balancing on each single SM, the model evaluates the load balancing at SM level. The SM load balancing criterion is defined as follows:

$$\textsc{LoadBalanc}_{SM} = 1 - \frac{\max\limits_{\text{SM}}(\text{TotActiveCycles}) - \text{AvgCycles}}{\max\limits_{\text{SM}}(\text{TotActiveCycles})}$$

$$\text{where} \qquad \text{AvgCycles} = \frac{\sum\limits_{SM} \text{TotActiveCycles}}{|\text{SM}|}$$

F. L1/L2 Granularity

GPU applications require optimized data access patterns and properly aligned data structures to achieve high memory bandwidths. In particular, efficient applications hide the latency of memory accesses by combining multiple memory accesses into single *transactions* that match the granularity (i.e., the cache line size) of the memory space[3]. The proposed performance model includes two complementary criteria to describe the quality of memory access patterns:

$$\text{L1\_}\textsc{Granularity} = \frac{|\#\text{L1\_transactions}|\cdot 128}{\sum\limits_{\text{T}\in\{\text{Mem\_instr}\}} |\text{Mem\_instr}_\text{T}|\cdot \text{size}_\text{T}}$$

$$\text{L2\_}\textsc{Granularity} = \frac{|\#\text{L2\_transactions}|\cdot 32}{\sum\limits_{\text{T}\in\{\text{Mem\_instr}\}} |\text{Mem\_instr}_\text{T}|\cdot \text{size}_\text{T}}$$

The criteria take into account the number of actual transactions towards the L1(L2) memory, the cache line size (128 Bytes for L1, 32 Bytes for L2), the total number of memory instructions (`load` and `store`) to access the global memory, and the size of their accesses $size_T$ (1/2/4/8/16 Bytes).

G. Shared Memory Efficiency

This criterion measures the kernel efficacy to exploit the *data locality* concept through the on-chip shared memory. The shared memory allows high memory bandwidth for concurrent accesses, but it requires appropriate access patterns to achieve the full efficiency. On the other hand, an excessive and disorganized use of the shared memory leads to bank conflicts, which involve the memory instructions to be re-executed thus serializing the thread execution flow. This optimization criterion is defined as follows:

$$\textsc{ShMemEfficiency} = \frac{\#\text{SharedLoadTrans} + \#\text{SharedStoreTrans}}{\#\text{SharedLoadAcc} + \#\text{SharedStoreAcc}} \times \\ \times F_{SHMEM}$$

The formula is defined in terms of total number of transactions towards shared memory for both load and store operations over the total number of accesses in shared memory for load and store instructions (which includes the re-executed memory instructions due to bank conflicts). It is weighted by the shared memory characterization function (F_{SHMEM}) presented in Section III-B

H. Throughput/Occupancy

The *Throughput/Occupancy* criterion is defined as follows:

$$\begin{array}{l}\textsc{Throughput/}\\ \textsc{Occupancy}\end{array} = \begin{cases} 1 & \text{if } MemThr \approx 1 \\ 1-(1-\text{occ})\cdot \textsc{MemThr} & otherwise \end{cases}$$

where the occupancy value (*occ*) depends on the kernel configuration (i.e., block size, grid size, and amount of shared memory allocated for the kernel variables), and

$$\textsc{MemThr} = \frac{\text{AchivedThroughput}}{\text{TheoreticalPeakThroughput} \times F_{\text{DRAMThr}}}$$

The theoretical peak throughput is weighted through the $F_{DRAMThr}$ characterisation function. If the memory throughput value is close to 1, the throughput/occupancy criterion cannot be further improved. Otherwise, the throughput/occupancy criterion is calculated as the potential improvement of the memory throughput metric by using all device threads ($1 -$ *occ*). This criterion is particularly useful in such applications that do not achieve the theoretical occupancy of the device. As proved in [16], a high theoretical GPU occupancy is not necessary to reach the peak performance. In contrast, a high theoretical occupancy and a low value of the throughput/occupancy criterion suggests optimizing the application kernel through a re-configuration to increase the occupancy.

V. Performance Prediction

Improving any of the optimization criteria presented in Section IV impacts on the overall application speedup. A speedup increasing is proportional to the criterion improvement. The potential speedup of the *host synchronization, divergence, warp and SM load balancing* and *throughput/occupancy* criteria are defined as follows:

$$\textsc{HostSync}^{\text{SP}} = \frac{1}{\textsc{HostSync}}$$

$$\textsc{Divergence}^{\text{SP}} = \frac{1}{\textsc{Divergence}}$$

$$\textsc{LoadBalanc}_{WARP}{}^{\text{SP}} = \frac{1}{\textsc{LoadBalanc}_{WARP}}$$

$$\textsc{LoadBalanc}_{SM}{}^{\text{SP}} = \frac{1}{\textsc{LoadBalanc}_{SM}}$$

$$\begin{array}{c}\textsc{Throughput/}\\ \textsc{Occupancy}^{\text{SP}}\end{array} = \frac{1}{\textsc{Throughput/Occupancy}}$$

The potential speedup of the *device synchronization* criterion also depends on the fraction of time spent in stall state over the total kernel time:

$$\textsc{DeviceSync}^{SP} = \left(1 - \frac{\text{TotActiveWarps }/|\text{Warps}|}{\text{CLK\_cycles}}\right)\cdot \text{StallSync}$$

$|Warps|$ represents the maximum number of thread warps of the device, while CLK_cycles represents the total number of GPU clock cycles elapsed to execute the kernel. The value in the round brackets represents the overall percentage of inactivity of the GPU warps (i.e., warps in stall state).

The potential speedup definition of *L1,L2 granularity* and *shared memory efficiency* criteria also depends on the percentage of time the application uses the L1, L2, and shared memory, respectively, over the total execution time:

[3] This concept applied to the L1 cache is also known as *memory coalescing*.

$$\text{L1\_Granularity}^{\text{SP}} = \frac{1}{\text{L1\_Granularity}} \cdot L1^{\%}$$

$$\text{L2\_Granularity}^{\text{SP}} = \frac{1}{\text{L2\_Granularity}} \cdot L2^{\%}$$

$$\text{ShMemEfficiency}^{\text{SP}} = \frac{1}{\text{ShMemEfficiency}} \cdot \text{ShMem}^{\%}$$

$L1^{\%}$, $L2^{\%}$, and $ShMem^{\%}$ are evaluated as follows. The model classifies the application activity in terms of DRAM accesses, cache accesses, shared memory accesses, arithmetic instructions, and idle states. The profiler provides the accurate evaluation of the idle states, the exact amount of memory transactions for each memory typology, and the number of arithmetic instructions. Twelve microbenchmarks (eight for memory accesses considering both load and store operations and four for arithmetic instructions) allow estimating the memory latencies and the arithmetic instruction throughputs. $L1^{\%}$, $L2^{\%}$, and $ShMem^{\%}$ are calculated by comparing the sum of such latencies spent in a specific memory level with the total cycles elapsed during the kernel execution.

Finally, the overall potential speedup of the application is defined as follows:

$$\text{PotentialSpeedup} = \begin{cases} \dfrac{1}{\text{MemThr}} & \text{if memory-bounded} \\ \dfrac{1}{\text{ArithThr}} & \text{if compute-bounded} \end{cases}$$

where $MemThr$ has been defined in Section IV-H, while $ArithThr$ is defined as follows:

$$\text{ArithThr} = \frac{AchivedThroughput}{TheoreticalPeakThroughput \times F_{ARITHThr}}$$

The formula expresses the potential speedup of the application under tuning as the inverse of the memory throughput or the arithmetic throughput depending on whether the application is memory-bounded or compute-bounded. Such an information is provided by the profiler.

VI. Experimental Results

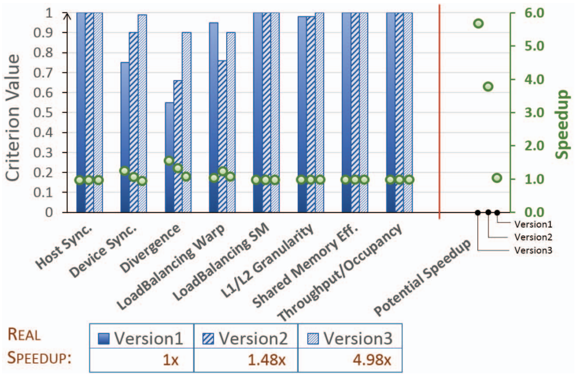
The proposed performance model has been applied for tuning and improving the performance of three different CUDA applications, *reduction*, *matrix transpose*, and *BFS* for an NVIDIA (Kepler) GEFORCE GTX 780 device. The experiments have been run on such a device with CUDA Toolkit 7.0, AMD Phenom II X6 1055T (3GHz) host processor, Debian 3.2.60 operating system, and NVIDIA `nvprof` profiler.

A. Case study 1: Parallel Reduction

Given a vector of data $\{x_1, x_2, \ldots, x_n\}$, the reduction applies an operator $\oplus$ to all elements and returns a single element $R = x_1 \oplus x_2 \oplus \ldots \oplus x_n$. We analysed the reduction implementation provided in [17], and we applied two tuning iterations with the proposed model.

Figure 4 shows the results. In the left-most side, the columns represent the optimization value [0-1] for each criterion at each tuning iteration. The dot in a column represents the potential contribution of an improvement of such a criterion in the predicted overall speedup. In the right-most side, the figure reports the overall potential speedup of the application (see Section V), which is calculated for the original code (Version1) as well as for the two optimized versions of the code. The *L1* and *L2 granularity* criteria has the same value and are thus reported in Figure 4 as a single item.

In the analysis of the original code (Version1), the model predicted a potential speedup of 5.8x. The criteria values underline that the application bottlenecks are mainly due to high thread divergence, inadequate synchronization of GPU threads, and unbalancing at warp level. We first optimized the code by focusing on synchronization. We organized the threads by using the warp-centric method proposed by [18], which allowed us to reduce the number of barriers from $\log(\text{BLOCKSIZE})$ to one.



Real Speedup:	▆ Version1	▨ Version2	▨ Version3
	1x	1.48x	4.98x

FIG. 4: *Experimental results of case study 1.*

The analysis of such a first optimization (Version2) confirmed the improvement on the thread synchronization (see *device synchronization* criterion), which influenced (positively) the divergence level of threads. Nevertheless, the results underlined a slight improvement of the *memory troughput* metrics, which motivates the marginal increasing of the Version2 speedup (1.48x). On the other hand, the model predicted a further potential improvement of the speedup up to 3.9x, by suggesting to optimize the divergence aspect.

We addressed the divergence issue in the second optimization (Version3) by increasing the number of elements computed by a single thread. We also applied instruction-level parallelism techniques to increase the arithmetic throughput. The analysis of Version3 shows that all the optimization criterion values are close to the maximum and the potential speedup is close to 1x. These values suggest that any further optimization on the considered criteria on the adopted GPU device would not improve the current speedup. We measured the Version 3 speedup equals to 4.98x, while the potential speedup predicted by the model was 5.8x.

B. Case study 2: BFS

In parallel computing, BFS is one of the most representative irregular application that involves thread divergence, workload imbalance, and poorly coalesced memory accesses. We analysed the BFS implementation provided in [19].

Figure 5 shows the results obtained by applying the proposed performance model for two optimization steps of the code. In the original imlementation (Version1), the results indicate many different causes of performance bottlenecks and a potential speedup up to 10x. We first focused in the low value of the *host synchronization* criterion, which was due to a high number of kernel calls. We optimized the code by enabling the *inter-block synchronization*[15], which allows the device and the host execution to be completely separated and, thus, the application to be organized into one single kernel.

The analysis of such a first optimization (Version2) confirmed the total elimination of the host synchronization overhead thanks to the single kernel implementation. This allowed reaching a fair speedup (2.44x). On the other hand, the results show that the optimization didn't impact on the other criterion values. The results underline that the code suffers from *L1/L2 granularity*, for which the criterion value is the lowest and

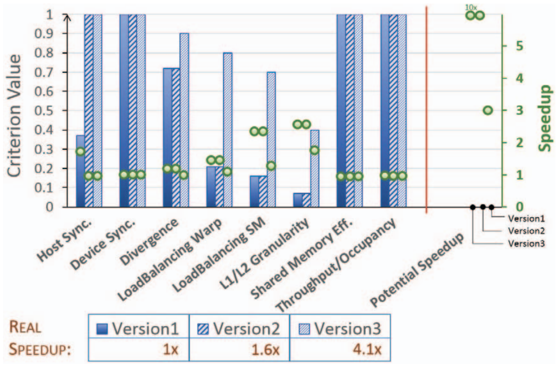


FIG. 5: *Experimental results of case study 2.*

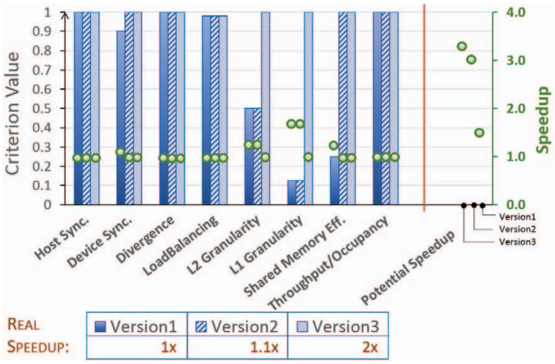


FIG. 6: *Experimental results of case study 3.*

the potential contribution ($\approx 2.5x$) in the overall speedup is the highest. Nevertheless, to the best of our knowledge, we could not have removed such a bottleneck, which is mainly due to the irregular data structures on which the implemented algorithm works. We focused on the low values of the warp/SM load balancing criteria, which suggest to better organize the GPU thread allocations. We optimized the code (Version3) by re-arranging the threads in groups with the aim of cooperative visiting single vertices instead of sets of vertices. Version3 provides a speedup of 4.1x w.r.t. the original code. The analysis of Version3 underlines that improving the *L1/L2 granularity* would be the main important optimization to double the speedup and to reach the predicted 10x value.

C. Case study 3: Matrix Transpose

We analysed the matrix transpose implementation presented in [20], which is characterized by data tiling in shared memory and thread organization in 2D hierarchical grids and blocks.

Figure 6 shows the results. The original code already provides values close to the maximum for the *host* and *device synchronizations*, *divergence*, and *Warp/SM load balancing* criteria. For all the other criteria, even though they have very low values (between 0.1 and 0.5), the model predicts marginal potential speedups. This is due to the fact that the application algorithm relies on very regular and independent tasks. This justifies the limited potential overall speedup (≈3.3x) predicted by the model.

In the first optimization (Version2), we focused on improving the shared memory bank conflicts (*shared memory efficiency* criterion) by applying the *memory padding technique* [20]. As expected, since such a technique has more impact on the NVIDIA Fermi than on the NVIDIA Kepler architecture [20] the gained speedup is marginal.

In the second optimization (Version3) we taken into account the memory access patterns to improve the *L1* and *L2 granularity* criteria. Their low values suggest that the memory accesses do not match the granularity of the respective caches, thus involving a waste of the memory bandwidth. We fully optimized both the criteria by simply re-organizing the thread block configuration and by resizing the memory *tiles* (as shown by the third columns of the three criteria in Figure 6). The Version3 implementation provides a speedup of 3x against the 3.3x predicted by the model.

VII. Remarks

This paper presented a fine-grained performance model for GPU architectures. It relies on microbenchmarks to characterize the GPU device and on several application criteria to measure the implementation quality, to give interpretable hints, and to accurately calculate potential performance. The paper presented the results obtained by applying the proposed model for tuning different GPU applications, by underlining how the advanced profiling results have been effectively used to focus the tuning effort in specific code optimizations.

References

[1] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, “Parallel computing experiences with cuda,” *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008.

[2] J. Nickolls and W. J. Dally, “The gpu computing era,” *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar. 2010.

[3] 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*, 2012, pp. 11–22.

[4] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, “A survey of performance modeling and simulation techniques for accelerator-based computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 272–281, 2015.

[5] R. Dietrich, F. Schmitt, R. Widera, and M. Bussmann, “Phase-based profiling in gpgpu kernels,” in *Proc. IEEE ICPPW*, 2012, pp. 414–423.

[6] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira Jr., “Profiling divergences in gpu applications,” *Concurrency Computation Practice and Experience*, vol. 25, no. 6, pp. 775–789, 2013.

[7] P. Guo and L. Wang, “Accurate cross-architecture performance modeling for sparse matrix-vector multiplication (spmv) on gpus,” *Concurrency Computation*, vol. 27, no. 13, pp. 3281–3294, 2015.

[8] S. Hong and H. Kim, “An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness,” *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun. 2009.

[9] K. Kothapalli, R. Mukherjee, M. Suhail Rehman, S. Patidar, P. Narayanan, and K. Srinathan, “A performance prediction model for the cuda gpgpu platform,” in *Proc. of IEEE HiPC*, 2009, pp. 463–472.

[10] M. Zheng, V. Ravi, W. Ma, F. Qin, and G. Agrawal, “Gmprof: A low-overhead, fine-grained profiling approach for gpu programs,” in *Proc. of IEEE HiPC*, 2012.

[11] A. Kerr, G. Diamos, and S. Yalamanchili, “Modeling gpu-cpu workloads and systems,” in *Proc. of GPGPU*, 2010, pp. 31–42.

[12] K. Sato, K. Komatsu, H. Takizawa, and H. Kobayashi, “A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems,” in *Proc. of IEEE ISPA*, 2011, pp. 135–142.

[13] “Nvidia, nvidia compute ptx: Parallel thread execution.”

[14] S. Bochkanov and V. Bystritsky, “Alglib-a cross-platform numerical analysis and data processing library,” *ALGLIB Project. Novgorod, Russia*, 2011.

[15] S. Xiao and W. chun Feng, “Inter-block gpu communication via fast barrier synchronization,” Dept. of Computer Science Virginia Tech, Tech. Rep., 2009.

[16] V. Volkov, “Better performance at lower occupancy,” in *Proceedings of the GPU Technology Conference, GTC*, vol. 10. San Jose, CA, 2010.

[17] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[18] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating cuda graph algorithms at maximum warp,” in *Proc. of ACM PPoPP*, 2011, pp. 267–276.

[19] F. Busato and N. Bombieri, “Bfs-4k: An efficient implementation of bfs for kepler gpu architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2015.

[20] G. Ruetsch and P. Micikevicius, “Optimizing matrix transpose in cuda,” *Nvidia CUDA SDK Application Note*, vol. 18, 2009.