Top-Down Performance Profiling on NVIDIA's GPUs

Alvaro Saiz
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
alvaro.saizf@alumnos.unican.es

Pablo Prieto
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
prietop@unican.es

Pablo Abad
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
abadp@unican.es

Jose Angel Gregorio
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
monaster@unican.es

Valentin Puente
Computer Engineering Group
Universidad de Cantabria
Santander, Spain
vpuente@unican.es

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00026

***Abstract*—The rise of data-intensive algorithms, such as Machine Learning ones, has meant a strong diversification of Graphics Processing Units (GPU) in fields with intensive Data-Level Parallelism. This trend, known as general-purpose computing on GPU (GP-GPU), makes the execution process on a GPU (seemingly simple in its architecture) far from trivial when targeting performance for many dissimilar applications. A proof of this is the existence of many profiling tools that help programmers to understand how to maximize hardware utilization. In contrast, this paper proposes a profiling tool focused on microarchitecture analysis under large sets of dissimilar applications. Therefore, the tool has a double objective. On the one hand, to check the suitability of a GPU for diverse sets of application kernels. On the other hand, to identify possible bottlenecks in a given GPU microarchitecture, facilitating the improvement of subsequent designs. For this purpose, using Top-Down methodology proposed by Intel for their CPUs as inspiration, we have defined a hierarchical organization for the execution pipeline of the GPU. The proposal makes use of the available hardware performance counters to identify how each component contributes to performance losses. We demonstrate the feasibility of the proposed methodology, analyzing how different modern NVIDIA architectures behave running relevant benchmarks, assessing in which microarchitecture component performance losses are the most significant.**

Keywords—GPU, NVIDIA, Top-Down, Hardware Event Counters, Performance Profiling.

I. Introduction

The quest for performance has been a constant in microprocessor history. In the beginning frequency increase was one of the primary sources of performance, supported by fast integration technology evolution. More recently, alternative sources of performance, such as instruction and thread level parallelism (ILP and TLP respectively) have come to the rescue to maintain the rhythm of earlier improvements. The recent popularity gained by applications able to process a huge amount of data (the Machine learning field is probably the clearest example) has turned data level parallelism (DLP) into an extremely profitable source of performance gain. Nowadays, most general-purpose processors include vector extensions able to operate with a growing amount of data with a single instruction (SIMD), making use of specific functional units and registers. However, GPUs have turned out to be much more suitable devices for this purpose, thanks to their massive parallelism and better energy efficiency [1]. Initially designed for graph processing in videogames and 3D applications, GPUs are currently employed in diverse fields, such as machine learning [2], climate research applications [3][4], cryptocurrency mining [5], genome sequencing [6][7], etc.. The popularization of GPUs has turned this kind of hardware into a sort of general-purpose device, extending the term GP-GPU to identify the heterogeneous utilization of current GPUs.

Performance optimization in a GPU is far from trivial and requires both programming experience and a good understanding of underlying microarchitecture. Despite being an apparently simple microarchitecture (compared to a speculative, superscalar, out-of-order CPU), the management of threads, instructions and data is complex, making it difficult to identify sources of performance degradation. To help programmers to optimize their code, vendors such as NVIDIA provide sophisticated toolsets to collect and interpret a large amount of information about execution progress. In this way, for example, programmers can easily identify which parts of their code are the most time-consuming ones.

These profiling tools are in most cases programmer oriented. They provide information related to planning and runtime features of a single application, but they fail to show how the microarchitecture performs and do not identify its main bottlenecks. From a computer architecture viewpoint, a much more relevant analysis would consist of the evaluation of many state-of-the-art applications, identifying common points of performance degradation. This kind of analysis would enable early detection of performance problems and better evolution of next generation microarchitectures (aiming to adapt the architecture to the requirements of as many apps as possible and to avoid adjusting applications to the microarchitecture).

Intel's Top-Down methodology [8], makes use of hardware event counters to detect performance bottlenecks in the microarchitecture of general-purpose processors. Our work is inspired by this methodology, extending its use to GPUs, with emphasis on NVIDIA GPUs. The proposed tool does not aim to replace current frameworks employed by both hardware architects and software developers, but to be a complement. NVIDIA profiling tools are the best option for fine performance tuning of a kernel to specific hardware architecture. Similarly, the use of hardware simulation tools (such as GPGPUsim [9]) is the appropriate methodology to evaluate microarchitecture proposals on significant code sections. However, none of these frameworks has the appropriate features to perform a

This work has been supported by the Spanish Government (Agencia Estatal de Investigacion) under grant PID2019-110051GB-I00

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00026

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

performance analysis in a large set of applications. Concerning simulation tools, computational effort limits their affordable scope. As an example, the authors in [10] demonstrate that performance simulation is approximately 9 orders of magnitude slower compared to real hardware (from less than a second to more than a week). Similarly, the current approach of profiling tools, focusing on a detailed code analysis of each application kernel, makes it difficult to extract generalized conclusions about the behavior of underlying architecture.

In contrast, the methodology proposed, with a reduced runtime overhead, enables the identification of which parts of the microarchitecture should be optimized to improve a large and significant enough group of applications. As this preliminary exploration is extensive, it could enable the concentration of simulation efforts on those key bottlenecks detected. Similarly, results provide software developers information about how their software behaves on a specific architecture, and what should be the target of any code improvement. In conclusion, as GPU computing spreads throughout new fields with more dissimilar applications, Top-Down methodology is a suitable complement for a growing design space exploration. The main contributions are listed here:

- We analyze the organization of NVIDIA GPUs, defining a hierarchical organization for the components inside each pipeline stage.
- We evaluate the available hardware event counters, in order to define which ones are necessary to propose the Top-Down methodology.
- We develop a tool that automates the evaluation process, making use of NVIDIA profiling tools and evaluating it when running several workloads and multiple GPU architectures.

II. Background

As computing technology evolves, understanding how code interacts with underlying hardware becomes more difficult. For this reason, it becomes necessary the utilization of middleware tools that help in reasoning about low-level behavior. Microarchitecture profiling is one of the key aspects in code optimization and it is extremely helpful to enable hardware architects to understand pipeline behavior. Consequently, new processor generations provide a growing amount of information concerning hardware, making architectural performance details "visible" to the programmer.

A. Performance Monitoring Units & Hardware Events

Current processors (both CPU and GPU) make use of devoted hardware in charge of monitoring the behavior of different components in the microarchitecture. These units, known as Performance Monitoring Units (PMUs) count and collect multiple events, which are then used to infer system performance issues. In the case of NVIDIA's GPUs, Performance Counters are the elements in charge of event counting. The access to these counters is through the CUDA Profiling Tools Interface library (CUPTI) [11] and events can be collected in two different ways:

- HWPM: this profiling mechanism collects data from any GPU hardware unit, but the number of units that can be monitored is limited to a subgroup of available hardware units.
- SMPC: through this mechanism, only data from the Streaming Multiprocessor (SM) are collected, but every SM in the GPU can be checked at the same time.

The low-level API (CUPTI) available for NVIDIA GPUs presents a few peculiarities that must be considered for our proposal. First, the number of available counters is limited, and depends on the method employed to collect events. If it is necessary to measure more events than the available counters, re-execution of the code is required. These execution replays, known as passes, can be defined at different granularities, per kernel or user defined. Second, until Turing architecture NVIDIA provides two types of measurements for hardware counters, events and metrics. Events are values extracted from the direct measurement of a single microarchitectural event, while metrics are extracted through the operation of multiple events, and provide results that can be computed as ratios, throughputs or other usual values employed to measure performance (such as Instructions per Cycle or IPC). The utilization of metrics can have a direct impact on the number of passes required, because the number of events required to calculate each metric cannot be predicted. This model combining events and metrics has been available in compute capabilities (CC) from 3.0 to 7.2. Since the advent of Turing architecture this model has been unified, eliminating the event term and integrating these measurements as metrics.

B. CLI Profiling Tools

Working with metrics and events with low-level libraries is a complex process. Fortunately, NVIDIA provides a profiling middleware that delivers an abstraction layer between the PMU and the application under evaluation. *nvprof* [12] is a command-line interface (CLI) focused on performance profiling. Its default operation mode returns general information about each kernel executed in the GPU, as well as the memory transactions between GPU and CPU. *nvprof* can also perform a more specific analysis, reading user-defined events and metrics. This was the default CLI profiling tool until capability 7.2. Starting from capability 7.5, *ncu* (Nsight Compute CLI) [13] has become the default command line profiler. This tool provides programmer-oriented information about each kernel executed, divided into three sections. First, general information about frequencies and utilization of GPU units is shown, as well as some hints about throughput and memory bandwidth losses. Second, *ncu* displays information about execution issues related to application programming (grid and block size, number of threads, registers per thread, memory utilization per block, etc.). Finally, an occupation-based analysis is performed (occupation per warp, maximum theoretical occupation per SM, etc.). Similarly to *nvprof*, *ncu* also provides direct access to every metric available in the PMU.

These CLI tools have the main purpose of helping the programmer in the fine-tuning process needed to improve the execution of its application kernels. However, they also provide a structured and capability independent framework on top of which to develop our methodology.

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

C. Top-Down Methodology for CPUs

Current CPUs present complex microarchitectures: superscalar, out-of-order execution, speculation, etc. Understanding the impact of each component on the execution of a fraction of code is complex and methodologies such as Top-Down [8] can help with this issue.

This methodology describes a mechanism to identify the performance bottlenecks, describing processor pipeline through a hierarchy of components and identifying using PMU events which fraction of pipeline stalls are caused by each component. The hierarchy, shown in Figure 1, defines the first-level fractions of the pipeline as follows:

- *Frontend Bound*: the frontend of the processor corresponds mainly to the stages in charge of instruction fetching and decoding. The stalls detected in this section are related to cache instruction misses, (Frontend Latency Bound) or decoding inefficiencies (Frontend Bandwidth Bound).
- *Bad Speculation*: this component identifies performance stalls caused for two main reasons: cycles wasted by the CPU executing a speculative instruction that is not retired later and the cycles devoted to restoring the processor state before prediction miss.
- *Retiring*: this category represents correctly retired instructions. It does not represent stalls, but real performance, so a higher value implies better performance.
- *Backend Bound*: The backend of the pipeline corresponds to the stages in charge of the execution of the instructions provided by the frontend. The stalls in this category are related to the inability to complete an operation, caused by, for example, the lack of accessed data in L1 cache (Memory Bound) or the lack of available functional units (Core Bound)

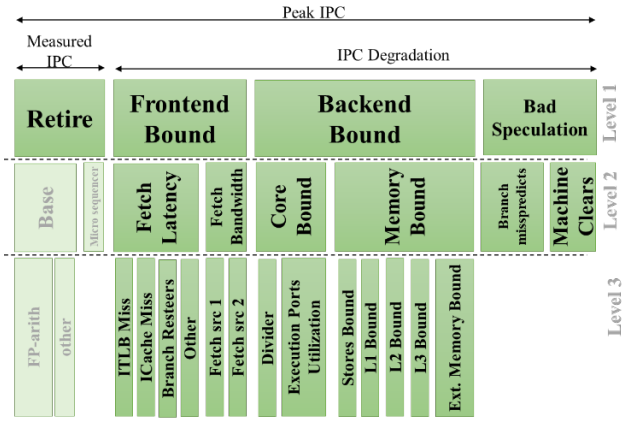

Figure 1. Top-Down hierarchical structure

As the hierarchy level increases, each part of the pipeline is described in more detail, splitting the microarchitecture into more specific components and determining their contribution to performance loss. The main target of this work is to try to replicate this analysis model for NVIDIA GPUs. The microarchitecture of GPU processors also presents a pipelined structure and multiple hardware events can be measured. Keeping this in mind we will define a hierarchical representation adapted to GPU pipeline and analyze whether the events and metrics available are enough to implement the Top-Down methodology.

III. NVIDIA'S GPU MICROARCHITECTURE

The peak performance of a GPU relies on its capacity to extract as much data-level parallelism as possible, which reflects on its hardware organization. A GPU processor is made up of multiple Graphic Processing Clusters (GPCs) sharing the L2 cache level of the memory hierarchy. Each GPC is divided into multiple Texture Processing Clusters (TPCs) and each TPC consists of two Stream Multiprocessors (SMs). The SM is the basic hardware unit in the GPU, in charge of selection, planning and execution of kernel threads. Each SM has its own private L1 instruction and data caches and an in-order execution pipeline (in some cases, it is made up of two or more sub-partitions with equivalent pipelines) capable of executing 32 threads simultaneously.

From a programmer perspective, hardware resources can be seen as a large grid to map all the available threads. This grid consists of a bi-dimensional structure of blocks, each of them grouping 1024 or 2048 threads. Blocks are not executed in a fixed order, they are concurrent, asynchronous and time multiplexed. The threads of each block are grouped into warps for execution, each warp consisting of 32 threads in NVIDIA architectures. The threads inside a warp traverse the pipeline stages at the same time and a stall in a single thread affects the rest of the threads in the warp. Once the block is assigned to a SM, all its threads must complete execution through its in-order pipeline [14].

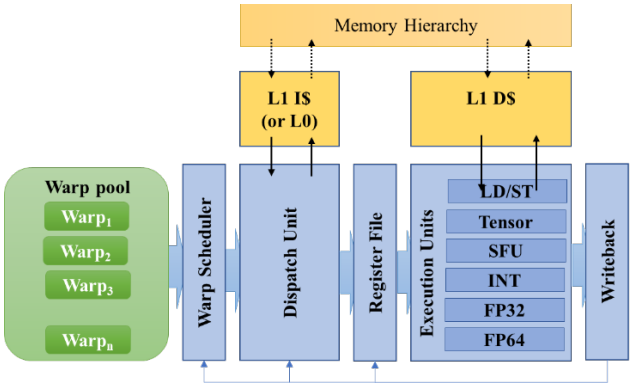

Figure 2. Pipeline sketch of the Streaming Multiprocessor.

A sketch of a generic pipeline can be seen in Figure 2. The first step inside a SM pipeline consists of the selection of which warps are executed on each sub-partition, making use of a first warp scheduler. Chosen warps move to the Dispatch Unit, in charge of deciding which warps will move forward in each cycle. Inside the Dispatch Unit the current PC of each warp indicates which instructions must be fetched and decoded. These instructions are stored in an Instruction Buffer, where a second scheduling algorithm (a sort of scoreboard) decides which instructions should be issued. Next, source and destination registers are allocated at the Register File. Each warp instruction generates 32 operations, requiring a large number of available

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

registers to work properly. Every thread inside the warp needs its operands read before advancing to the next stage. Finally, warp instructions move forward to the functional units, in charge of performing the operations. Each sub-partition provides Functional Units for multiple types of operations, the followings being the most usual:

- FP64/FP32: double/single precision, FP operations
- INT: integer operations.
- LD/ST: units in charge of memory operations.
- SFU: specific operations such as sin, cos, sqrt, log, etc.
- Texture Unit: texture operations, graphic-oriented.

The pipeline finishes by writing back the results and updating warp PCs to point to the next instruction to be executed. Taking this pipeline structure as a starting point, the next section describes how to implement a hierarchical organization, making use of available events and metrics to estimate the contribution of each hierarchy component to performance degradation.

IV. Top-Down on Nvidia GPUs

All the code generated to implement the methodology is available as an anonymous git repository [1], as well as the scripting required to replicate the results provided in Section V.

The objective of the proposed methodology is to identify the performance losses that occur in the pipeline of the GPU, with respect to a theoretical ideal execution. In this ideal execution, the instruction of each of the threads that compose the warps in every SM (and their corresponding sub-partitions) must be executed in one clock cycle. Consequently, this value will be proportional to the number of warp operations executed in the SMs.

We define two main sources of performance loss in a GPU: stalls and divergence. Stalls occur when a particular portion of the GPU pipeline cannot execute any operation (it is blocked) because it does not have the necessary resources to perform it. What we call divergence occurs when there is underutilization of the threads of a warp, for example, due to a conditional jump, or when instructions need to be executed more than once. For this purpose, the actual performance (IPC_{RETIRE}) can be expressed as:

$$IPC_{RETIRE} = IPC_{MAX} - (IPC_{DIVERGENCE} + IPC_{STALL}) \quad (1)$$

Where IPC_{MAX} is the maximum theoretical IPC, IPC_{STALL} is the IPC lost due to stalls in the pipeline, $IPC_{DIVERGENCE}$ is the IPC lost due to underutilization of the threads of a warp.

The proposed Top-Down hierarchy for GPUs in this work is shown in Figure 3. The sources of performance loss denoted in (1) should be captured in the first level of analysis (highest level of the hierarchy). With more details provided in deeper levels of analysis.

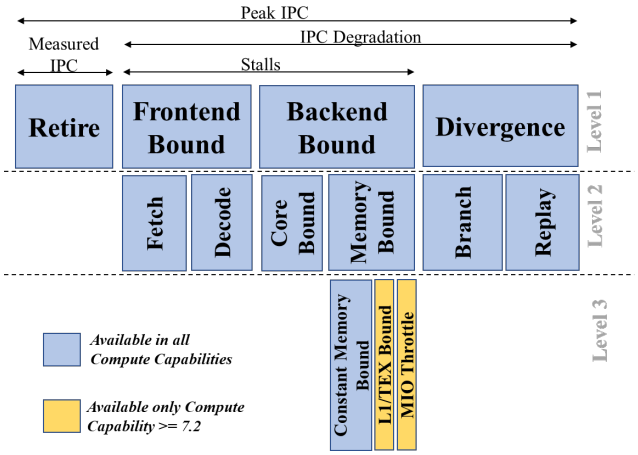

Figure 3. Proposed Top-Down hierarchy for NVIDIA GPUs

A. Retire

The Retire IPC determines the actual throughput obtained by the application, i.e., the measured IPC, defined as the number of instructions that are completely executed per cycle in a SM. It should be noted that the metric "IPC" provided by the tool, indicates the number of instructions executed in an SM per cycle, but assuming every thread of the warp has executed this instruction. Therefore, for each instruction, the percentage of warp threads that make use of the functional units must be considered. The IPC_{RETIRE} can then be calculated as:

$$IPC_{RETIRE} = IPC_{REPORTED} \times Warp\_Efficiency \quad (2)$$

To obtain those values, we make use of hardware counters, through the events and/or metrics provided by the profiling tools. These depend on the GPU microarchitecture used in the analysis as explained previously and are indicated by the Compute Capability (CC), which denotes the general characteristics and specifications of the GPU.

TABLE I. Retire Metrics (CC < 7.2)

Metric	Variable	Description
ipc	$IPC_{REPORTED}$	Average number of executed instructions per cycle, per SM.
warp_execution_efficiency	Warp_Efficiency	Ratio of average active threads per warp to the maximum.

TABLE II. Retire Metrics (CC ≥ 7.2)

Metric	Variable	Description
smsp__inst_executed.avg.per_cycle_active	$IPC_{REPORTED}$	Average number of instructions per cycle, per SM.
smsp__thread_inst_executed_per_inst_executed.ratio	Warp_Efficiency	Ratio of average active threads per warp to the maximum.

B. Divergence

We define two causes of performance loss that are contemplated within the Divergence portion of the hierarchy:

[1] https://github.com/nvidiatopdown/TopDownNvidia

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

the warp underutilization (e.g., execution of conditional jumps) and the repetition of instructions. When a conditional jump is reached during code execution in GPUs, two different situations can occur. In an IF without ELSE, there will be threads with different flows inside the warp, i.e., some threads execute instructions that other threads within the same warp do not have to execute (those that enter the IF vs. those that do not). This situation implies that some of the GPU's functional units may be wasted, because in a warp all threads must execute the same instruction. Additionally, in an execution of an IF with ELSE, some threads execute one path while others execute the other. This case is more taxing than the previous one since both parts have to be executed.

Although this problem has been present in GPUs since their inception (and some architectures still have it), the trend in recent NVIDIA architectures is to solve this problem by taking advantage of threads that are not doing work to stole instructions from others. In our Top-Down analysis, the IPC loss due to warp underutilization can be calculated as:

$$IPC_{BRANCH} = IPC_{REPORTED} \times (1 - Warp\_Efficency) \quad (3)$$

These values are obtained with the metrics in TABLE I and TABLE II.

In addition, we must consider the instruction re-execution e.g., the Functional Units of each of the SM sub-partitions are limited and, in some cases, there are not enough of them to satisfy the complete execution of the warp. In this case, it is necessary to dispatch the same instruction multiple times. The effect of having to dispatch the same warp instruction several times causes a loss of performance as more cycles are required to complete the operation. The Replay contribution can be expressed as:

$$IPC_{REPLAY} = IPC_{ISSUED} - IPC_{REPORTED} \quad (4)$$

TABLE III. REPLAY METRICS (CC < 7.2)

Metric	Variable	Description
ipc	$IPC_{REPORTED}$	Average number of instructions per cycle, per SM.
issued_ipc	IPC_{ISSUED}	Average number of instructions issued per cycle, per SM, including replayed instructions.

TABLE IV. REPLAY METRICS (CC ≥ 7.2)

Metric	Variable	Description
smsp__inst_executed.avg.per_cycle_active	$IPC_{REPORTED}$	Average number of instructions per cycle, per SM.
smsp__inst_issued.avg.per_cycle_active	IPC_{ISSUED}	Average number of instructions issued per cycle, per SM, including replayed.

Issued and reported IPC can be directly obtained through the metrics in TABLE III and TABLE IV. Finally, the contribution of the divergence in (1) can be obtained as:

$$IPC_{DIVERGENCE} = IPC_{BRANCH} + IPC_{REPLAY} \quad (5)$$

Where IPC_{BRANCH} and IPC_{REPLAY} can be obtained with equations (3) and (4) respectively.

TABLE V. FRONTEND METRICS (CC < 7.2)

Metric	Variable	Description
stall_inst_fetch	$STALL_{FETCH}$	Percentage of stalls due to the next assembly instruction not yet been fetched.
stall_sync	$STALL_{FETCH}$	Percentage of stalls occurring because the warp is blocked at a syncthreads() call.
stall_other	$STALL_{DECODE}$	Percentage of stalls occurring due to miscellaneous reasons, including conflicts in the registers bank.

C. FrontEnd

This portion of the hierarchy includes stalls caused by the FrontEnd of the GPU pipeline, which can be divided into two stages. The first stage is the one that determines the warp chosen to be executed in each of the sub-partitions of each of the SMs. Stalls in this section, called Fetch Bound, include losses due to failures in the instruction cache, synchronization between threads, etc. The second stage is where reading and assignment of registers for each thread is done. Stalls in this section, called Decode Bound, include conflicts in the register bank or latencies in the dispatch unit.

FrontEnd Stalls can be expressed then as:

$$STALL_{FRONTEND} = STALL_{FETCH} + STALL_{DECODE} \quad (6)$$

Each variable in (6) is obtained as a sum of all the metrics associated with it in TABLE V and TABLE VI.

TABLE VI. FRONTEND METRICS (CC ≥ 7.2)

Metric	Variable	Description
smsp__warp_issue_stalled_no_instruction_per_warp_active.pct	$STALL_{FETCH}$	Percentage of stalls waiting to be selected to fetch an instruction or waiting on an instruction cache miss.
smsp__warp_issue_stalled_barrier_per_warp_active.pct	$STALL_{FETCH}$	Percentage of stalls because the warp is waiting for sibling warps at a CTA barrier.
smsp__warp_issue_stalled_membar_per_warp_active.pct	$STALL_{FETCH}$	Percentage of stalls because the warp is waiting on a memory barrier.
smsp__warp_issue_stalled_branch_resolving_per_warp_active.pct	$STALL_{FETCH}$	Percentage of stalls waiting for a branch target to be computed, and the warp PC to be updated
smsp__warp_issue_stalled_sleeping_per_warp_active.pct	$STALL_{FETCH}$	Percentage of stalls due to all threads in the warp being in the blocked, yielded, or sleep state.
smsp__warp_issue_stalled_misc_per_warp_active.pct	$STALL_{DECODE}$	Percentage of stalls occurring due to miscellaneous reasons, including conflicts in the registers bank.
smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct	$STALL_{DECODE}$	Percentage of stalls because the warp is waiting on a dispatch stall.

Due to limitations on the available metrics, IPC_{STALL} must be obtained indirectly. The individual stalls needed in our hierarchy are given by the tool (TABLE V and TABLE VI) in the form of a percentage of global stalls. For this reason, the global IPC_{STALL} must be calculated from the rest of the variables in (1).

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

$$IPC_{STALL} = IPC_{MAX} - IPC_{DIVERGENCE} - IPC_{RETIRE} \quad (7)$$

IPC_{RETIRE} and $IPC_{DIVERGENCE}$ are obtained as seen in sections IV.B and IV.C, while IPC_{MAX} is a system parameter that depends on the architecture. As all of the metrics used refer to the average performance of the SMs, we define the theoretical maximum performance of a GPU, i.e. the performance of an ideal theoretical execution, as the most efficient use of each Stream Multiprocessor (SM). If possible, each SM dispatch unit will send one instruction from the warp to the functional units. This way, the maximum IPC (IPC_{MAX}) will occur when all the instructions sent by the dispatch units are executed completely, so IPC_{MAX} is equal to the number of Dispatch Units per SM.

Finally, $IPC_{FRONTEND}$ is calculated as a portion of the global IPC_{STALL} (7), using $STALL_{FRONTEND}$ from (6).

$$IPC_{FRONTEND} = \frac{STALL_{FRONTEND}}{100} \times IPC_{STALL} \quad (8)$$

And each portion contributes proportionally:

$$IPC_{FETCH} = \frac{STALL_{FETCH}}{100} \times IPC_{STALL} \quad (9)$$

$$IPC_{DECODE} = \frac{STALL_{DECODE}}{100} \times IPC_{STALL} \quad (10)$$

D. BackEnd

Finally, under this category are stalls in the Backend of the GPU pipeline. This includes the execution of the instructions of each thread, as well as the possible dependencies that may occur.

As mentioned in Section III, instruction execution is performed using the functional units of each of the sub-partitions of the different SMs. However, an instruction might not be executed if a functional unit is busy resolving a previous instruction. In that case, there will be warp threads that must remain blocked waiting for other threads to finish and free the functional unit. This waiting time can be high, considering that there are operations that take longer than others (for example, a division with respect to an addition), and this is reflected in the Core Bound portion of our Top-Down hierarchy. Problems regarding dependencies and problems in the memory hierarchy are reflected in the Memory Bound portion of the hierarchy.

TABLE VII. BACKEND METRICS (CC < 7.2)

Metric	Variable	Description
stall_ exec_dependency	$STALL_{CORE}$	Percentage of stalls because an input is not yet available
stall_pipe_busy	$STALL_{CORE}$	Percentage of stalls occurring because a compute operation cannot be performed because the compute pipeline is busy
stall_ memory_dependency	$STALL_{MEMORY}$	Percentage of stalls because a memory operation cannot be performed due to resources not being available, or because too many requests are outstanding
stall_constant_ memory_dependency	$STALL_{MEMORY}$	Percentage of stalls occurring because of immediate constant cache miss
stall_memory_throttle	$STALL_{MEMORY}$	Percentage of stalls occurring because of memory throttle

TABLE VIII. BACKEND METRICS (CC ≥ 7.2)

Metric	Variable	Description
smsp__warp_issue_ stalled_math_pipe_throttle_ per_warp_active.pct	$STALL_{CORE}$	Percentage of stalls because the warp is waiting for the exec. pipe to be available.
smsp__warp_issue_ stalled_long_scoreboard _per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls waiting for a scoreboard dependency on a L1TEX operation.
smsp__warp_issue _stalled_imc_miss_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls because the warp is waiting for an immediate constant cache (IMC) miss.
smsp__warp_issue_ stalled_mio_throttle_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls waiting for the MIO (memory input/output) instruction queue not to be full.
smsp__warp_issue_ stalled_drain_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls after EXIT waiting for all memory instructions to complete.
smsp__warp_issue_ stalled_lg_throttle_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls waiting for the L1 inst. queue for local and global (LG) operations not to be full.
smsp__warp_issue_ stalled_short_scoreboard_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls waiting for a scoreboard dependency on a MIO operation (not to L1TEX).
smsp__warp_issue_ stalled_wait_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls because the warp is waiting on a fixed latency execution dependency.
smsp__warp_issue_ stalled_tex_throttle_ per_warp_active.pct	$STALL_{MEMORY}$	Percentage of stalls waiting for the L1 instruction queue for texture operations not to be full.

Analogously to the Frontend, the performance loss in this case is determined by the stalls or locks, according to the following expression:

$$STALL_{BACKEND} = STALL_{CORE} + STALL_{MEMORY} \quad (11)$$

Similarly to what happened in the FrontEnd, we need to obtain the IPC_{STALL} (7) and represent each portion as a fraction of it.

$$IPC_{BACKEND} = \frac{STALL_{BACKEND}}{100} \times IPC_{STALL} \quad (12)$$

$$IPC_{CORE} = \frac{STALL_{CORE}}{100} \times IPC_{STALL} \quad (13)$$

$$IPC_{MEMORY} = \frac{STALL_{MEMORY}}{100} \times IPC_{STALL} \quad (14)$$

V. GPU TOP-DOWN USE CASE

In this section, we will test the proposed methodology, evaluating different NVIDIA GPUs running CUDA features and well-known Benchmarks. We evaluate the versatility of the tool, checking where the bottlenecks are, analyzing two recent GPUs of different compute capabilities. Finally, we analyze the cost of performing an analysis of this style and its feasibility.

We will work on two different architectures: a NVIDIA GTX 1070 installed on a laptop and a NVIDIA Quadro RTX 4000 installed on a server system. The GTX 1070 GPU is Pascal architecture, so our tool will use *nvprof* underneath. On the other

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

hand, the Quadro RTX 4000 GPU has Turing architecture, which implies the use of *nsight* for the analysis. TABLE IX summarizes the main characteristics of both GPUs.

TABLE IX. GPU CHARACTERISTICS

Feature	NVIDIA GTX 1070	NVIDIA Quadro RTX 4000
Compute Capability	6.1 (Pascal)	7.5 (Turing)
Memory	8GB DDR5	8GB DDR6
CUDA cores	1920	2304
SMs	15	36
SM Subpartitions	4	2
Power	150W	160W

A. CUDA Feature Analysis

Our first experiment demonstrates how to evaluate the microarchitectural implications of any feature present in modern versions of the CUDA Toolkit. NVIDIA provides a set of code samples for developers which demonstrate the multiple features in CUDA Toolkit. In this section we will focus on the binaryPartitionCG sample, which illustrates the benefits of binary partition cooperative groups and reduce within the thread block. Thread synchronization is a key aspect for performance. The simple construct provided by early CUDA versions was limited to a barrier to synchronize all threads of a thread block. Currently, programmers demand a more flexible way to define synchronization groups, smaller than thread blocks. CUDA cooperative groups provides an API to define synchronization groups at different granularity levels, from smaller-than-thread ones to global synchronization across multiple GPUs. Cooperative groups enable greater performance, design flexibility, and software reuse.

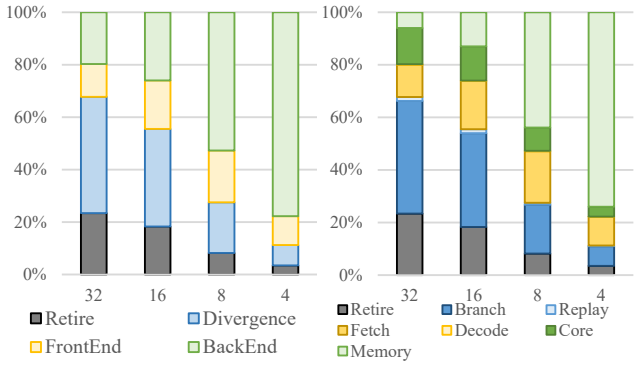

Figure 4. binaryPartitionCG Top-Down, level1 (left) and level2 (right)

The code in binaryPartitionCG sample is intended to illustrate the basic usage of binary partition cooperative groups within the thread block tile when divergent path exists. In this example, each thread loads a value from a random array. Next it checks if it is odd or even, creating cooperative groups based on this predicate. Finally, the number of odd/even values in the group is counted based on size of the binary groups and a global counter of odd values is updated as well as the global even & odd elements sum (using reduce).

In this section, we explore the performance impact of cooperative groups' size, partitioning a thread block into tiles of a decreasing number of threads, from warp size (32 threads in this case) to four threads per tile. For each tile size we apply the Top-Down analysis (level 1 and level 2), showing in Figure 4 the obtained results corresponding to the Turing architecture. As can be seen, tile size has a significant effect on overall performance, which clearly gets worse as the tile size decreases. A smaller tile size seems to have a beneficial effect on conditional jumps, reducing the contribution of Divergence to IPC degradation. In contrast, the Memory hierarchy becomes the main bottleneck as the tile size decreases. This degradation is so relevant that branch improvement is not able to compensate the growing Backend bound. As can be seen, this kind of methodology allows extracting relevant conclusions about the impact of a single CUDA feature on each part of the GPU pipeline.

B. Rodinia Benchmark

Rodinia is a benchmark developed by the University of Virginia [15] in 2009. It implements several applications from different scientific fields to compare performance on both CPUs and GPUs. The applications have been chosen to exploit different behaviors, making use of different parallelisms, access patterns and data sharing. Since its publication, the Benchmark has been growing in terms of number of applications (doubling them), performance improvements and correction of some errors or bugs. For the analysis, we will use only the latest version used in GPUs, 3.1.

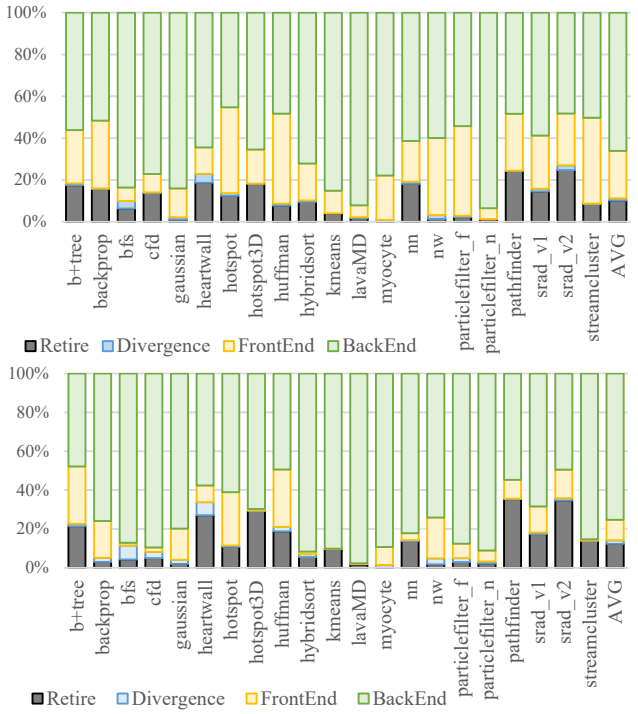

Figure 5. Rodinia Top-Down on NVIDIA Pascal (up) and Turing (down)

We apply, on both available architectures, the Top-Down methodology previously defined, trying to establish which part of the microarchitecture represents the weak spot for performance. For this purpose, the methodology reflects the fraction of performance lost at each component, with respect to

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

an ideal execution (ideal IPC). Figure 5 (top and bottom) provides these results, identifying the main performance bottlenecks. In the first place, the low value of reported IPC (Retire) in both architectures is remarkable. Applications are in general not able to exploit the performance provided by the microarchitecture. Additionally, it can be observed that those applications able to obtain an acceptable performance are the same in both cases (*srad_v2*, *heartwall*, *hotspot3D* or *pathfinder)*, which seems to indicate that Turing architecture does not suppose radically new architecture compared to Pascal.

Looking at the different sources of performance loss in the microarchitecture, divergence has a negligible impact, and most of the performance is lost in the Backend. In this case, both architectures show slightly different results. While, on average, Pascal loses almost 20% of performance in its Frontend, this value is reduced to less than 10% in Turing. However, this improvement does not translate directly into better performance, because Turing suffers from a larger degradation in its Backend

The level 1 analyzed gives an overview of the GPU behavior. However, in order to clarify the specific parts where bottlenecks have occurred, it is necessary to go down the hierarchy. We will expand the detail of our analysis on the Turing architecture, since the number of available metrics is higher in the more modern architecture. Figure 6 shows the level 2 Top-Down analysis performed on the Turing architecture for the applications of the *Rodinia* benchmark. Results have been normalized to Total IPC degradation for each application, in order to clarify the contribution of each microarchitecture component.

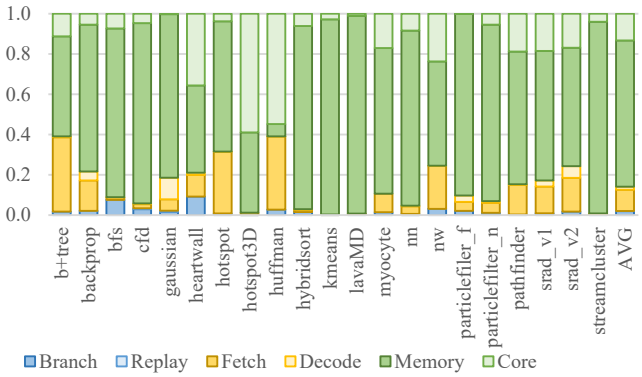

Figure 6. Top-Down level 2 analysis of Rodinia apps on Turing

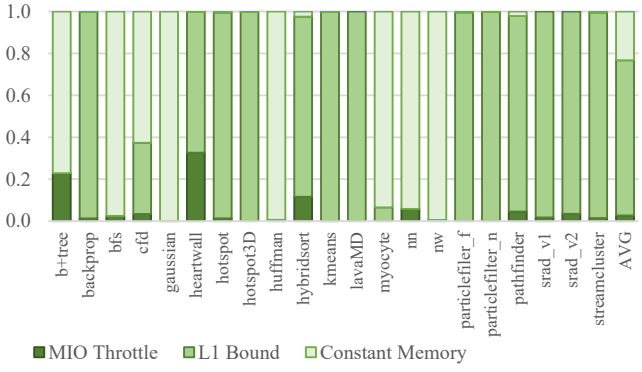

Figure 7. Top-Down level 3 analysis of Rodinia apps on Turing

In general, memory hierarchy is the main contributor to performance degradation, accounting for approximately 70% of the overall loss on average. The lack of available functional units (Core) and the problems with instruction fetch (Fetch) also have a relevant contribution to performance degradation, although their impact is much lower in comparison. In cases where divergence is relevant, it is mainly due to warp inefficiency, possibly due to conditional jumps. Finally, in the applications where the Frontend has a negative impact on performance, it seems that it is because the instruction fetch cannot be performed (perhaps because of problems in the instruction cache), and not so much because of conflicts in the register bank. With these results, Top-Down enables the establishment of a clear conclusion concerning the microarchitecture evaluated, identifying the memory hierarchy as its main bottleneck. Since it seems that the impact of the memory hierarchy on performance is generalized, and since this is where our tool offers more levels of detail, we perform a level 3 analysis.

Figure 7 shows a clear performance loss in L1 cache memory on average. These stalls can be caused by data dependencies or lack of resources in the Load/Store Unit instruction queue. Additionally, in some cases, applications such as *myocyte* and *nn* suffer bottlenecks in the constant memory. Finally, the impact of memory input/output (MIO Throttle) system, which reflects waits due to not having free entries in the memory queue (FIFO) seems to have little impact on performance.

C. Altis Benchmark

Altis [16] is a recent benchmark, an evolution of two previous suites, *Rodinia* and SHOC [17]. In particular, *Altis* incorporates several features used in modern and popular DNNs. In addition, the applications included have been retrofitted with the new features of GPUs with newer CUDA versions, e.g., Unified Memory, Dynamic Parallelism, HyperQ, GridSync…

As in the case of *Rodinia*, a Top-Down analysis is performed on *Altis* Benchmark for the most recent architecture, Turing. Figure 8 shows the results obtained. There is a prevalence of performance losses in the Backend, followed by the Frontend, with a notable difference between the two. Divergence, on the other hand, has little impact on the average performance loss.

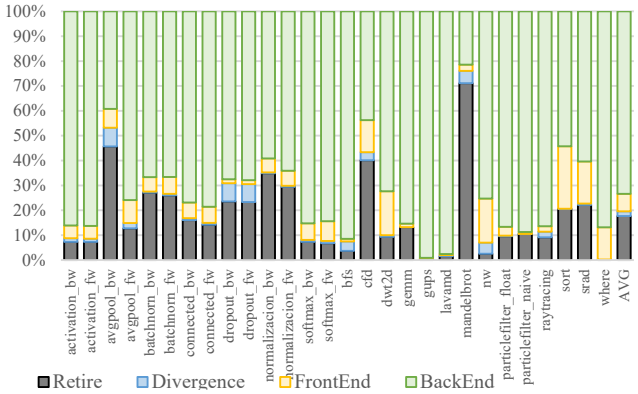

Figure 8. *Altis* Benchmark Top-Down on NVIDIA Turing GPU

Comparing the results with *Rodinia* on the same microarchitecture, they seem quite similar. The prevalence of performance loss is clearly in the backend, as was the case in

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

Rodinia. In both cases the Frontend constitutes the second source of performance loss, although the average performance (Retire) seems to be higher in *Altis* applications, with applications around 40% efficiency, and some reaching 70% of the theoretical maximum performance of the machine (*mandelbrot*). Regarding applications that are the same in both benchmarks, the results they offer are practically the same, even though *Altis* seems to have refitted some algorithms, e.g., *bfs* or *nw* applications obtain very similar performance in both benchmarks, on the same architecture. However, it is possible to appreciate differences in behavior in applications such as *cfd*, in which the GPU exhibits better performance.

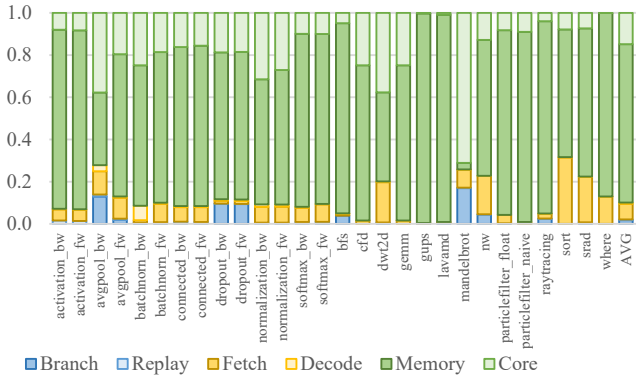

Figure 9. Top-Down level 2 analysis of *Altis* apps on Turing.

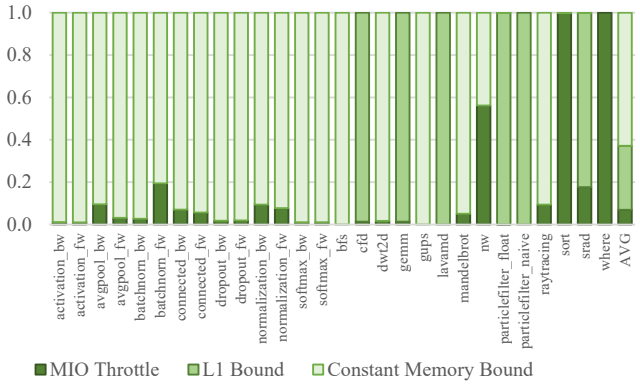

Figure 10. Top-Down level 3 analysis of *Altis* apps on Turing.

We will expand again the detail of our analysis of Altis apps on the Turing architecture. Figures 9 and 10 show the level 2 and level 3 Top-Down analysis performed. Similarly, results have been normalized to Total IPC degradation for each application, in order to clarify the contribution of each microarchitecture component. In level 2 results a consistent behavior for both suites can be appreciated. The memory hierarchy, on average, dominates performance degradation, accounting for ~70% of the overall loss. In contrast, from level 3 results, we observe that the applications from *Altis* benchmark seem to impose higher pressure on the constant cache, which is now the main contributor to performance loss. Machine learning apps seem to be the main culprits to this result. Thanks to the methodology proposed it is easy to detect that Deep Learning apps seem to have a performance bottleneck on Constant caching, which could be relevant for future microarchitectural improvements.

D. Dynamic Analysis

The tests performed so far consider a single Top-Down result for the whole application execution. These numbers are estimated measuring every kernel execution and calculating average values, weighted by the length of each kernel. In some cases, averaging could hide inner kernel performance bottlenecks. To illustrate this with an example, we analyze the behavior of two kernels over time (different invocations during workload execution). Figures 11 and 12 show the level 1 Top-Down analysis of srad_cuda_1 and srad_cuda_2, belonging to the *srad* application, showing their temporal evolution.

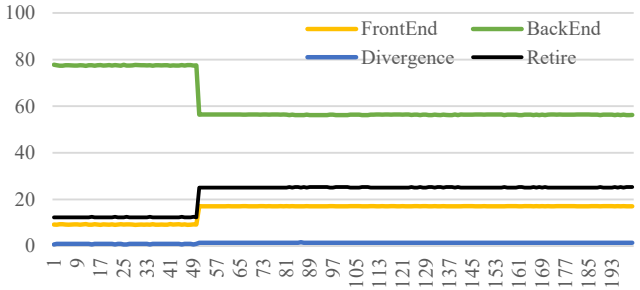

Figure 11. Level 1 Top-Down evolution of Altis *srad* kernel srad_cuda_1 on Turing architecture

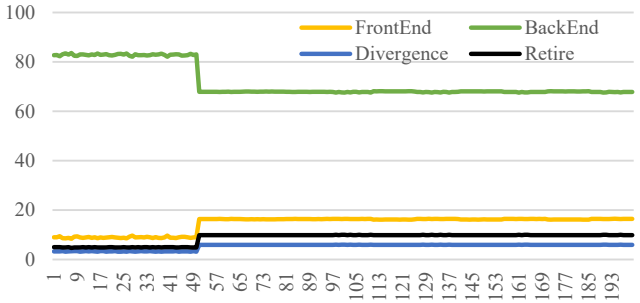

Figure 12. Level 1 Top-Down evolution of Altis *srad* kernel srad_cuda_2 on Turing architecture

In both kernels there are clearly two phases. A first one, shorter, from the beginning until invocation 50, and a second one from this moment to the end. Stalls in the Backend dominates the first execution phase in both kernels. While in the second phase, the pressure on the Backend seems to go down, especially in srad_cuda_1, where performance improves more noticeably. In both cases, as the second phase arrives, the pressure on the Frontend increases. In these cases, global Top-Down values could hide the pressure on the BackEnd observed in the first phase, and it could be interesting to split applications into multiple workloads, one for each phase, in order to reflect this temporal evolution. Although it seems evident, given the nature of the applications, that the main bottleneck in a GPU is in the backend, it should be noted that the distribution of hardware resources between backend and frontend is much more unbalanced than in a general purpose processor. For this reason, if applications are not tuned according to this imbalance the frontend can turn into an involuntary bottleneck. In fact, our experiments show that some applications show a performance degradation over 20% in the frontend.

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

We have observed similar behavior in more applications from the benchmarks evaluated, but the division of applications into multiple workloads according to the phases observed will be addressed in future versions of the framework.

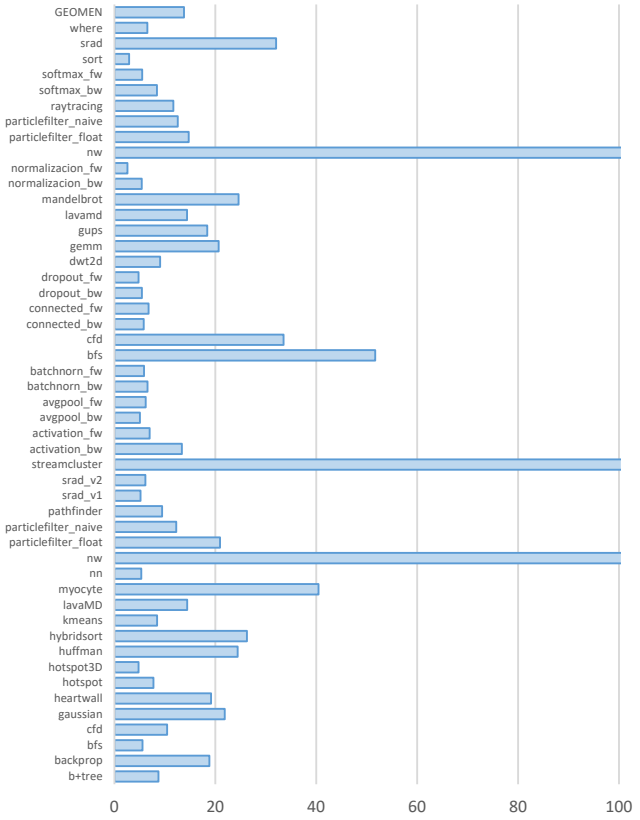

Figure 13. Overhead of GPU Top-Down analysis on Turing architecture running Rodinia and Altis benchmarks (~13 Times slower)

E. Overhead

As mentioned in Section II.A, available hardware counters are limited, and if the number of values to be measured exceeds this capacity, it is necessary to run the kernels several times in order to obtain the desired results. Additionally, in each repetition, there is a flush of some hardware components to mitigate the impact of the previous run, so each run executes under similar conditions. This replication occurs for every kernel and therefore can add considerable overhead that slows down the time required to obtain performance results.

We compare the time employed by the Turing system to run the raw benchmark execution, with the Top-Down level 3 analysis. The observable trend (Figure 13) is an average overhead of 13 between native execution and Top-Down, which seems reasonable considering that each kernel must be executed 8 times. Under certain conditions, such as an elevated number of kernel executions and applications with a large working-set (which increases the time required to flush memory elements between replays), the overhead required to collect desired metrics is unpractical. In these cases, there is certain room for improvement if measurements are limited to a subgroup of kernel executions (the size of the subgroup should be large enough to provide statistically sound results). Additionally, the low-level library (CUPTI) could also be helpful for this issue, providing a finer-grain control over replays, which could be performed on a specific section of code only.

VI. Related Work

As mentioned previously, the performance evaluation process follows two main approaches, depending on which will be the source of improvement. First, given a particular HW, we can work to tune software to extract the maximum performance benefit from microarchitectural features. Second, given a broad and representative group of applications (SW), we can evolve hardware features to adapt to novel software requirements.

Software optimization relies on performance analysis tools [18] (profilers) which collect runtime information (hardware events and metrics are a part of this information) to expose code bottlenecks. Major GPU vendors provide their own tools, such as nsight systems from NVIDIA [13] or AMD [19]. There are multiple alternative tools, not limited to a single vendor or programming mode. ARM MAP [20] offers a kernel analysis including a line-level breakdown of warp stalls helping the developer to tune code accordingly. Open source tools such as TAU [21] and HPCToolkit [22] perform basic performance analysis. In the case of HPCToolkit, a recent enhanced version [23] extends callgraphs with event and metric counting.

The use of this kind of profiling tools is usual, with examples such as understanding the performance bottlenecks of current sparse matrix multiplication algorithms [24], irregular graph algorithms [25], or implementing Roofline performance analysis [26], a methodology which provides insights into the efficiency of an application utilization of the memory subsystem. Performance analysis through these tools in most cases focuses on exposing code bottlenecks. In this process, Top Down could help to identify which parts of the microarchitecture are causing runtime problems, but it could go one step further, highlighting common hardware bottlenecks for multiple applications (significant enough to be representative of a general-purpose scenario (GPGPU).

GPU microarchitectures evolve trying to adapt hardware features to software requirements. In this process, it is important to correctly identify those bottlenecks common to a significant number of applications and evaluate each microarchitecture modification in detail prior to its adoption in a new SM generation. In this process, simulation frameworks are the most appropriate tool, GPGPUsim [9] and Multi2Sim [14] being two of the most relevant in the GPGPU field. These tools have been employed in previous studies for characterization purposes. The authors in [9] characterize the performance impact of several microarchitectural designs using GPGPUSim. In [27] the authors propose a set of microarchitecture-agnostic GPGPU workload characterization metrics and use these metrics to study multiple benchmarks. Moreover, in [28] a set of low IPC applications is analyzed, identifying the benchmarks' key architectural bottlenecks and applying an analytic model to predict the performance impact of mitigating each bottleneck.

The limited accuracy and elevated computation cost are a drawback that impairs the utilization of simulation tools for characterization purposes. This is where the proposed

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.

methodology could be extremely helpful, providing a much faster characterization process and organizing microarchitectural events in a consistent and hierarchical way.

VII. Conclusions and Future Work

In this work we demonstrate that, with the available events and metrics in NVIDIA GPUs, a Top-Down methodology can be implemented. Following a similar approach to intel's Top-Down, we define a hierarchical structure of Streaming Multiprocessor pipeline and then infer through hardware events the contribution of each component to performance degradation. The Depth of the pipeline hierarchy was limited by the available events and metrics. If PMU evolves including more detailed metrics for the memory hierarchy or functional unit utilization, the hierarchy could be completed at its deepest level.

This methodology has demonstrated its versatility with two experiments. First, we compared two different microarchitectures, observing the significant differences between Frontend and Backend as a source of performance degradation. Second, we also verified that the main performance bottleneck in SM pipeline is still the Backend.

It is possible that the evaluation of individual kernels instead of whole applications could increase the information provided by the tool. Currently the application can offer the results at a kernel level, making possible to increase the information provided by the tool. Unfortunately, in some applications where the number of kernels executed largely exceeds 100,000 it is impractical to achieve full results. Sampling based approaches can be useful here. Despite not employed in the experiments, *ncu* provides the tools to limit the number of times a kernel is executed to provide measured metrics. An alternative solution would be the utilization of a lower-level interface such as a CUPTI library to implement the methodology. This could enable increased control of execution overhead, through the re-definition of measurement intervals. We will work on this issue in future versions of the tool.

Acknowledgment

The authors would like to thank Jose Angel Herrero for his valuable assistance with the computing environment HPC cluster Calderon within the datacenter 3Mares. We also appreciate all the comments made by reviewers which helped to improve the quality of the paper.

References

[1] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of GPU-accelerated systems," 2012.

[2] D. Steinkraus, I. Buck, and P. Y. Simard, "Using GPUs for machine learning algorithms," 2005, doi: 10.1109/ICDAR.2005.251.

[3] K. Kurowski, M. Kulczewski, and M. Dobski, "Parallel and GPU based strategies for selected CFD and climate modeling models," *Environ. Sci. Eng.*, 2011, doi: 10.1007/978-3-642-19536-5_57.

[4] D. Leutwyler, O. Fuhrer, X. Lapillonne, D. Lüthi, and C. Schär, "Towards European-scale convection-resolving climate simulations with GPUs: A study with COSMO 4.19," *Geosci. Model Dev.*, 2016, doi: 10.5194/gmd-9-3393-2016.

[5] J. A. Dev, "Bitcoin mining acceleration and performance quantification," 2014, doi: 10.1109/CCECE.2014.6900989.

[6] I. Medina *et al.*, "Highly sensitive and ultrafast read mapping for RNA-seq analysis," *DNA Res.*, 2016, doi: 10.1093/dnares/dsv039.

[7] R. Wilton, T. Budavari, B. Langmead, S. J. Wheelan, S. L. Salzberg, and A. S. Szalay, "Arioc: High-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space," *PeerJ*, 2015, doi: 10.7717/peerj.808.

[8] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 35–44, doi: 10.1109/ISPASS.2014.6844459.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2009, pp. 163–174, doi: 10.1109/ISPASS.2009.4919648.

[10] Z. Yu *et al.*, "Accelerating GPGPU architecture simulation," 2013, doi: 10.1145/2494232.2465540.

[11] "Cutpi user's guide," 2011. https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUPTI_Users_Guide.pdf.

[12] NVIDIA, "Profiler user's guide," *DU-05982-001_v11.4*, 2021. https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf.

[13] NVIDIA, "Nsight Compute Command Line Interface," *v2021.2.1*, 2021. https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf.

[14] X. Gong, R. Ubal, and D. Kaeli, "Multi2Sim kepler: A detailed architectural GPU simulator," 2017, doi: 10.1109/ISPASS.2017.7975298.

[15] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

[16] B. Hu and C. J. Rossbach, "Altis: Modernizing GPGPU Benchmarks," 2020, doi: 10.1109/ISPASS48437.2020.00011.

[17] A. Danalis *et al.*, "The Scalable Heterogeneous Computing (SHOC) benchmark suite," 2010, doi: 10.1145/1735688.1735702.

[18] M. Knobloch and B. Mohr, "Tools for GPU computing-Debugging and performance analysis of heterogenous HPC applications," *Supercomput. Front. Innov.*, 2020, doi: 10.14529/js200105.

[19] P. Mistry and B. Purnomo, "Profiling OpenCL kernels using wavefront occupancy with radeon GPU profiler," 2019, doi: 10.1145/3318170.3318182.

[20] C. January, J. Byrd, X. Oró, and M. O'Connor, "Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead," in *Tools for High Performance Computing 2014*, 2015.

[21] S. Mayanglambam, A. D. Malony, and M. J. Sottile, "Performance measurement of applications with GPU acceleration using CUDA," 2010, doi: 10.3233/978-1-60750-530-3-341.

[22] L. Adhianto *et al.*, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurr. Comput. Pract. Exp.*, 2010, doi: 10.1002/cpe.

[23] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of GPU-accelerated applications," 2020, doi: 10.1145/3392717.3392752.

[24] S. AlAhmadi, T. Muhammed, R. Mehmood, and A. Albeshri, "Performance characteristics for sparse matrix-vector multiplication on gpus," in *EAI/Springer Innovations in Communication and Computing*, 2020.

[25] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," 2013, doi: 10.1109/IISWC.2013.6704684.

[26] J. Li, M. Lakshminarasimhan, X. Wu, A. Li, C. Olschanowsky, and K. Barker, "A Sparse Tensor Benchmark Suite for CPUs and GPUs," 2020, doi: 10.1109/IISWC50251.2020.00027.

[27] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," 2010, doi: 10.1109/IISWC.2010.5649549.

[28] E. Blem, M. Sinclair, and K. Sankaralingam, "Challenge Benchmarks That Must be Conquered to Sustain the GPU Revolution," *Emerg. Appl. Many-Core Archit.*, 2011.

Authorized licensed use limited to: COLLEGE OF ENGINEERING - Pune. Downloaded on June 16,2025 at 06:40:03 UTC from IEEE Xplore. Restrictions apply.