GPUscout: Locating Data Movement-related Bottlenecks on GPUs

Soumya Sen
soumya.sen@tum.de
Technical University of Munich
Garching, Bavaria, Germany

Stepan Vanecek
stepan.vanecek@tum.de
Technical University of Munich
Garching, Bavaria, Germany

Martin Schulz
schulzm@in.tum.de
Technical University of Munich
Garching, Bavaria, Germany

Abstract

GPUs pose an attractive opportunity for delivering high-performance applications. However, GPU codes are often limited due to memory contention, resulting in overall performance degradation. Since GPU scheduling is transparent to the user, and GPU memory architectures are very complex compared to ones on CPUs, finding such bottlenecks is a very cumbersome process.

In this paper, we present a novel method of systematically detecting the root cause of frequent memory performance bottlenecks on NVIDIA GPUs that we call GPUscout. It connects three approaches to analyzing performance – static CUDA SASS code analysis, sampling warp stalls, and kernel performance metrics. Connecting these approaches, GPUscout can identify the problem, locate the code segment where it originates, and assess its importance.

This paper illustrates the capabilities and the design of our implementation of GPUscout. We show its applicability based on three commonly-used kernels, yielding promising results in terms of accuracy, efficiency, and usability.

CCS Concepts: • **Computing methodologies** → **Parallel computing methodologies**; • **General and reference** → **Metrics**; *Performance*; • **Computer systems organization** → *Parallel architectures.*

Keywords: High performance computing, Performance analysis, NVIDIA, GPU, CUDA, Data-movement, Profiler, SASS

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624208

ACM Reference Format:
Soumya Sen, Stepan Vanecek, and Martin Schulz. 2023. GPUscout: Locating Data Movement-related Bottlenecks on GPUs. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3624062.3624208

1 Introduction

Memory and data transfer speeds have been rising at a much slower pace than computing power, leading to severe performance limitations in applications caused by data transfers. Moreover, also from the energy perspective, the cost of data transfers is starting to dominate over the cost of computation in HPC systems [9]. Therefore, suboptimal memory access patterns in an application can have a huge negative impact both in terms of application performance and energy consumption. If the needed data is not provided quickly enough, the computation of a program cannot progress and is stalled. Hence, analyzing and optimizing data movements is crucial in increasing the overall performance of many (HPC) applications.

Throughout the last few years, GPUs have evolved into highly parallel computing units, allowing them to perform a wide range of general-purpose computations. This characteristic has made GPUs an interesting option when designing HPC systems, as scalable HPC algorithms often can make use of the wide parallelism GPUs offer. However, especially in such scenarios, memory access patterns are critical to support the high data needs of wide parallelism, and this can even amplify the effect of suboptimal memory accesses. Moreover, as GPU scheduling is transparent to the user, and the memory hierarchy is very complex, getting an understanding of data movements on the GPU is a complex task. This increases the need for effective and easy-to-use performance tools to detect and mitigate such scenarios.

GPU profilers are used extensively to measure the performance of CUDA-accelerated codes on NVIDIA GPUs. NVIDIA Nsight Compute [21], a successor to nvprof [19], is designed to assist CUDA kernel profiling with a powerful set of features driven by NVIDIA's own insights. Additionally, there are multiple community-driven projects that work on the premise of performance measurement of GPU kernel execution. Rice University's HPCToolkit Performance Tool [25] measures and analyses the performance of CPU

and GPU applications. It supports measuring CUDA applications using NVIDIA's CUPTI API. The TAU (Tuning and Analysis Utilities) parallel performance system [23] is another open-source toolset for instrumentation, measurement and analysis of HPC applications. Its approach to measuring performance was extended for NVIDIA GPU computations using the TAUcuda [14] measurement library. There are many other performance measurement and profiling tools with CUDA support, such as PAPI [3], Score-P/Scalasca [6, 8] or Vampir [15]. They provide profiling information at different granularities.

While the available tools can identify hotspots in a kernel, they lack closer analysis of data movements and memory behavior inside a GPU. Moreover, if suboptimal behavior is identified, they provide very little insight into the source of the problem. Nsight Compute, for instance, analyzes measurement data and proposes suggestions at kernel level. While it provides the most metrics compared to the other tools mentioned above, it fails to point at specific regions of code that can be optimized, nor advises it the user about what metrics are relevant for which optimizations. Other tools mentioned, like HPCToolkit and TAUcuda, support fine-grained suggestions based on instrumenting kernels to locate potentially problematic code sections. However, they do not associate the specific problem with the metrics derived, leaving this to the user. Moreover, deciding whether the problematic section has room for improvement and identifying the actual problem (and therefore also the potential solution) is left to the user. Finally, the main focus of these existing tools is mostly compute- rather than memory-related performance analysis.

In this paper, we present **GPUscout**, a performance optimization tool for CUDA codes. It analyzes CUDA kernels in great detail to pinpoint performance issues caused by suboptimal data movements within the complex multi-level memory hierarchies inside NVIDIA GPUs. GPUscout statically analyzes the GPU machine code to identify potentially problematic instruction patterns and combines these with carefully selected relevant kernel-wide metrics and sampled warp stall information, to provide a comprehensive analysis of the found issues, and to assess their severity. Identifying such memory-based bottlenecks can help users to take steps to avoid or reduce the associated performance penalty. The focus of the analysis is on **data movements inside the GPU** and thus other communication primitives, like CPU-GPU or GPU-GPU, are NOT considered in the current work. To the best of our knowledge, there does not exist another tool that combines static code analysis for NVIDIA GPUs with kernel-level metrics and warp stall sampling to precisely identify and provide detailed information about data-movement-based bottlenecks in NVIDIA GPU kernels.

The remainder of the paper is organized as follows. Section 2 reviews the relevant background for this work. The design and workflow of GPUscout are discussed in Section 3. Section 4 presents the methods implemented to detect potential memory-related bottlenecks. Next, Section 5 presents three kernels used to showcase the work with GPUscout. Section 6 describes the related work. Finally, Section 7 summarizes our work and presents some of the next steps in the development of GPUscout.

2 Background

GPUscout is built on top of existing APIs, further analyzing the data they provide. The fundamental background concepts presented in this section include the low-level Instruction Set Architectures (ISAs) – PTX and SASS, the NVIDIA CUPTI PC Sampling API, and the NVIDIA Nsight Compute CLI.

2.1 PTX and SASS

A CUDA kernel is a C/C++-like function that is executed in parallel on an NVIDIA GPU. When compiling CUDA kernels, the CUDA code is first transformed into an intermediate representation called Parallel Thread Execution (PTX). PTX ISA can be considered a low-level parallel execution assembly for a virtual GPU architecture. It has an unlimited register count and hence can be considered an abstraction of the underlying hardware [5]. The PTX representation is translated into yet another, lower-level, representation called SASS, which is then executed on the target GPU architecture. Hence a CUDA kernel can be characterized by two separate ISAs: PTX and SASS [2]. To get a better understanding of the performance characteristics and the data flow inside the kernel, it is necessary to analyze the sequence of the actual machine instructions – the SASS. The SASS assembly is obtained by disassembling the CUDA binary file using tools like `nvdisasm` and `cuobjdump` [18]. The disassembled file contains the instruction and the destination (+source) register numbers, which are later used for our analysis.

2.2 CUPTI PC Sampling API

The NVIDIA CUDA Profiling Tools Interface (CUPTI) [17] is capable of providing data for profiling and tracing tools for CUDA applications. CUPTI provides several APIs that allow users to interject tool code, read event counters, and gain more insights into the behavior of CUDA kernels.

GPUscout utilizes the PC Sampling API to sample PC (program counter) stalls, providing information about the various stall reasons and the affiliated CUDA code (and SASS) line number [1] with a low runtime overhead. Given the source code line information, the collected samples can be used to identify stalls in certain hotspot regions within a kernel.

2.3 Nsight Compute Metrics

NVIDIA's Nsight Compute CLI (`ncu`) [21], a successor to `nvprof` [19], provides a simple and customizable way to

[1]The source code attribution requires compilation flags `-g -generate-line-info`.

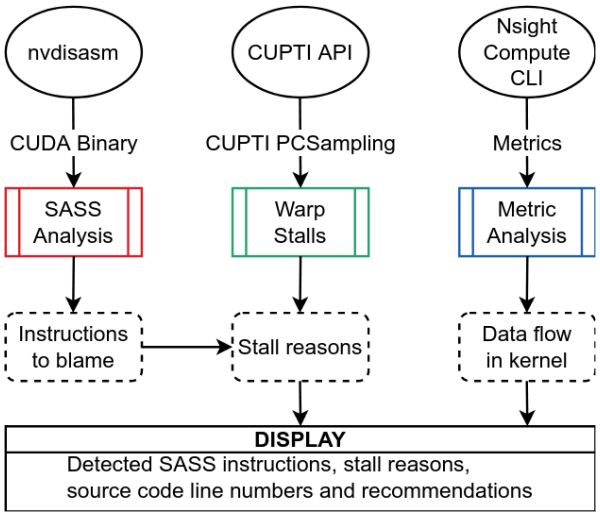

Figure 1. A simplified representation of GPUscout

collect various performance counter values. The metrics are collected with kernel granularity. Compared to alternatives like TAUcuda [14], ncu offers more metrics. Apart from using the native counter values, combining various performance counters provides more advanced analyses that expose the GPU memory system in greater detail.

For example, for all the streaming multiprocessors (SMs), the estimated number of queries to the L2 cache due to local memory can be expressed as

```
#SMs * (% cache miss) * (local memory instructions)
```

where the used metrics can be collected from ncu. This aggregated information helps observe the traffic between the CUDA cores and the L2 cache.

3 Design of GPUscout

The primary goal of GPUscout is to identify and provide accurate and detailed information about code regions of GPU kernels with poor performance caused by suboptimal data movements. To achieve this, GPUscout performs multiple different analyses of each tested kernel. Combining these analysis engines, we aim at identifying the memory-related bottlenecks in the source code, defining the problematic behavior, and providing detailed information about the observed patterns. The user is informed about the optimization to consider, is pointed directly at the problematic line in the CUDA code, and is provided with a detailed analysis of the problematic section, including kernel-level metrics.

Some recommended modifications from GPUscout might require the user to add a single keyword, while others might ask the user to restructure larger parts of their code. While we cannot claim that the provided recommendations will always improve the kernel's performance significantly, our experience shows that they very often do.

We use three analyses as the pillars of GPUscout:

1. **SASS Analysis**,
2. **Warp Stalls**, and
3. **Metric Analysis**.

Figure 1 presents a simplified representation of how the three above-mentioned pillars of GPUscout come together to provide a detailed analysis of the suboptimal data-access behavior in the kernel. The first pillar, the SASS Analysis, analyzes the generated SASS code to look for specific code patterns, indicating operations causing potentially problematic behavior. The analysis returns the exact locations of the detected patterns in the code. Using NVIDIA CUPTI libraries, we then collect PC samples to attribute warp stall reasons to the instruction detected by the SASS Analysis engine. Hence, the user learns how often on the instruction of interest which kind of warp stalls happen. Finally, a look at kernel-wide aggregated data helps to identify opportunities for memory improvements in the CUDA code or algorithm used. For this purpose, various metrics are collected to analyze the data movements and instruction execution in the kernel.

Static SASS Code Analysis. Static code analysis is the core part of GPUscout. It is responsible for identifying the bottlenecks and locating them in the source code. The analysis is performed at the assembly code of an NVIDIA GPU – the SASS – hence SASS analysis.[2] GPUscout operates directly on the disassembled SASS code without assuming the availability of the source CUDA program.

Each CUDA kernel is analyzed for various selected instruction and register usage patterns. Each analysis seeks a specific potentially suboptimal behavior (bottleneck). The SASS analysis then consists of multiple independent such checks for said patterns. A closer description of these analyses is provided in Section 4. Due to the modular architecture of GPUscout, all analyses are standalone, hence new bottleneck analyses can easily be added. Once an analysis has identified a potential bottleneck, the user is informed that such a pattern (potential bottleneck) was found. GPUscout points to the registers and corresponding source code line numbers causing the memory bottleneck. As the main output of the *SASS Analysis*, the user is advised to investigate potential optimizations at the given lines.

Warp Stalls. The CUPTI PCSampling API is used to fetch instruction stalls at the line numbers identified by the above-described static SASS code analysis. These stalls can occur due to various reasons, such as waiting for a branch to resolve, a memory operation to complete, or resources needed for execution. GPUscout offers more verbose explanations of the observed stalls, giving the user the needed context to understand the behavior. Knowing the reason for the stall can help assess the essence of the issue. For example, the stall reasons can point to a memory throttle due to a high number of pending memory operations, which indicates a

[2]Analogously to SASS, a PTX analysis is performed in Section 4.4.

drop in performance. This, in turn, can be minimized by combining several memory transactions into one, such as using vectorized loads as described in Section 4.1.

Performance metrics. As a second step, we collect multiple kernel-wide aggregate performance metrics at runtime to model the GPU execution flow and the overall program structure. This data helps users to assess the feasibility of the planned code changes, as well as the performance impact of the already-implemented modifications. GPUscout uses NVIDIA Nsight Compute [21] to collect requested performance metrics from the GPU, as it provides an easy and customizable interface to collect the necessary metrics. Since obtaining these metrics comes with significant overhead, the number of collected metrics is kept to minimum. We store the metrics and mangled kernel names to ensure consistency across all the metrics in each run of any given kernel.

These performance metrics detail a logical representation of the data movement through the various memory components of the GPU. For example, an L2 cache miss rate of 70 % denotes that a majority of the requests to the L2 cache are resolved at DRAM, indicating poor L2 cache utilization.

3.1 Workflow

The internal design of GPUscout is kept as modular and independent as possible to maintain and extend it easily in the future. Broadly speaking, the workflow of GPUscout can be split into four stages:

- **Configuration**: Before the data collection and the analysis begins, an initial setup takes place. The CUDA code needs to be compiled with the flags `-arch=sm_xx -g --generate-line-info`. The executable to analyze and the associated CUDA binary (`.cubin`) must be placed in the GPUscout subdirectory. GPUscout then disassembles these CUDA binaries and stores the generated SASS codes.
- **Static Code Instrumentation**: With the SASS code available, our static analysis engine examines the code to detect and locate possible bottlenecks. The results are stored for later use.
- **Dynamic Data Collection**: Once everything is set up, the actual data collection commences. We use the CUPTI PC Sampling API to obtain information about the warp stalls on instructions identified previously. Additionally, we collect selected metrics using the NVIDIA Nsight Compute CLI.
- **Data Evaluation**: In the final step, GPUscout displays the identified 'fault-causing' SASS instructions with the source code line number and warp stall reasons alongside specific information tailored to each identified bottleneck. Additionally, our tool informs the user about the data movement in the caches based on the collected metrics.

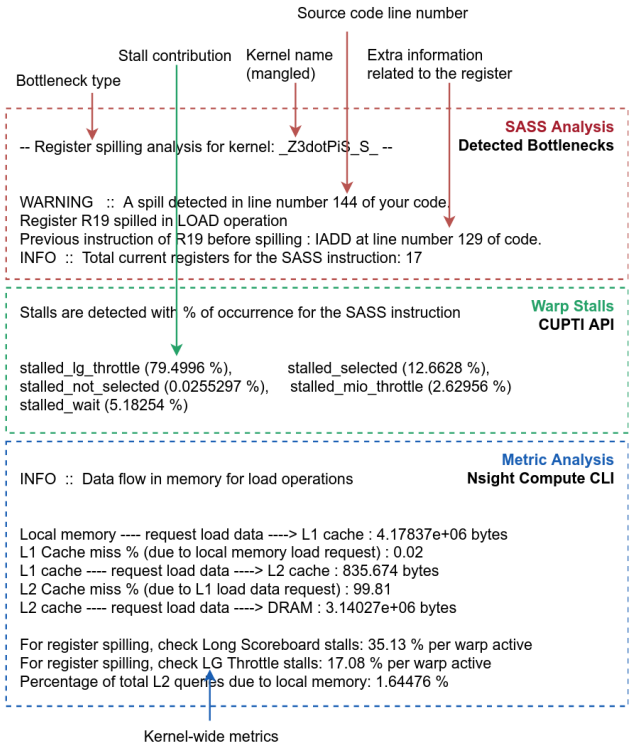

Figure 2. Sample output from GPUscout

Dry Run. GPUscout offers a `--dry-run` option. This command only inspects the SASS code to identify hotspots and memory bottlenecks, thereby making it possible to be executed without involving the GPU at all. Although this examination report misses *Warp Stalls* and kernel-wide *Metric Analysis*, it may serve as a starting point to validating the configuration and evaluating the kernel performance.

The `--dry-run` option can help users save time and resources by quickly spotting preliminary performance issues in the kernel without triggering the costly metric collection. Additionally, this option extends GPUscout's usability to older NVIDIA architectures not supported by ncu, such as Pascal, where GPUscout can still at least perform the static SASS analysis.

3.2 Presented Results

Figure 2 shows a sample output from GPUscout. The output is provided in a text-based form printed to the terminal. The information provided by the analysis can also be used to create visual and interactive presentations of the results in future iterations.

Analogous to the three analyses performed, the output can be split into three separate sections – the SASS analysis output, the Warp Stalls information, and the Performance metrics analysis.

The **SASS Analysis** shows a bottleneck detected due to register spilling into the local memory. When there are not

enough physical registers, a register is 'spilled' to the local memory of the GPU to make place for a new piece of data. The spilling, therefore, creates additional memory traffic. The amount of available registers depends on the GPU itself.

The warning details the register and the source code line numbers where the spill happened. The user, hence, knows where exactly in the kernel (i.e., in the source code) the problem occurs, simplifying the process of locating the bottleneck significantly. Any additional helpful data is presented alongside. In our example, the previous SASS instruction executed by the register hints at an `IADD` operation that caused the spill. Again, the source code line information is provided, so that the user knows where exactly to look. Additionally, GPUscout includes information regarding operations within a for-loop, the read-only nature of a register, or the live register pressure of an instruction.

Following the SASS analysis, GPUscout correlates the **Warp Stalls** extracted using the CUPTI API. The output shows the contribution of `lg_throttle` stall to be the largest offender for the assembly instruction identified by the SASS analysis. The `stalled_lg_throttle` indicates that a warp was stalled waiting for the L1 instruction queue handling local and global (LG) memory operations. This is typically caused by executing local or global memory operations too frequently.[3] Since register spilling increases the L1 traffic, reducing it should also reduce the `lg_throttle` stall. We can verify that by comparing the stall numbers of our updated implementation.

The final block in Figure 2, the **Metric Analysis**, outlines additional data based on kernel-wide metrics gathered using ncu. The metrics allow users to gain deeper insight into how the kernel is utilizing the GPU memory resources overall, thereby identifying potential areas of optimization. GPUscout provides information about the amount of data transfers through the CUDA global memory. In case shared or local memory is used, additional data regarding the corresponding memory requests and cache miss rates are collected. Alongside, GPUscout presents a set of computed metrics that should be paid attention to, given the recommended optimization. For the example at hand, increased memory traffic can impact performance adversely, especially for bandwidth-limited code. Therefore, the percentage of traffic to the L2 cache due to local memory requests is shown.

4 Bottleneck Detection

GPUscout employs the static SASS analysis to identify various frequently-occuring data-movement-based bottlenecks in the analyzed kernel. Each bottleneck is examined by a specialized routine that searches for specific instructions and patterns in the SASS. As a result of the SASS analysis, GPUscout provides information about the type of potential bottleneck, its location in the CUDA source code, and additional case-specific information. A provided recommendation is to be understood as highlighting a potential problem. It is up to the user to investigate whether the algorithm can accept the suggested change, and whether it will improve the performance (GPUscout provides metrics to support such an assessment).

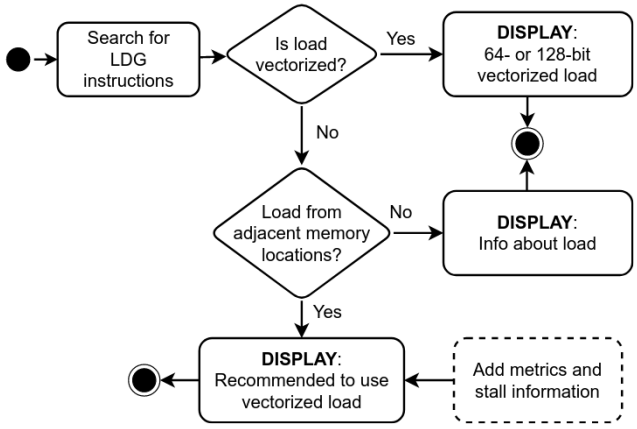

Figure 3. Workflow of vectorized load SASS analysis

This section sketches an overview of some of the employed strategies. To complement the SASS analysis, some interesting performance metrics and stall reasons are presented. Sections 4.1–4.3 present the analyses in more detail, while we sketch the essence of other analyses in Sections 4.4–4.7.

4.1 Use Vectorized Loads

A CUDA kernel can load and store data from global memory via 32-, 64-, or 128-bit transactions that are aligned to their size. A 32-bit load is denoted as an `LDG.E` instruction in the SASS code. Since each memory operation translates to a transaction and requires one instruction, loading larger amounts of data in one piece can improve the memory bandwidth utilization. Alternatively, if one CUDA kernel processes multiple elements, we can reduce the number of CUDA kernels spawned. To achieve that, vectorized load operations (`LDG.E.{64,128}`) can be used. This allows obtaining multiple pieces of data in a single fetch, resulting in only a fraction of the load instructions executed.

SASS Analysis. GPUscout identifies opportunities to convert global loads into vectorized loads by analyzing the SASS assembly code. Figure 3 illustrates the decision-making process of recommending the user to modify the code to use vectorized loads. GPUscout begins by searching for non-vectorized 32-bit loads (`LDG.E` instruction). The key deciding factor is whether multiple 32-bit load operations access data from neighboring memory locations. If so, the user is advised to use vectorized loads for the corresponding registers. The

[3] A more verbose interpretation of the stall reasons can be looked up in GPUscout 's manual.

report, of course, shows the corresponding source code line number.

Metrics. The register pressure is defined as the ratio of the required to available registers. Using vectorized loads may increase the register pressure, as multiple registers are filled at once. An increased register pressure may lead to a decreased occupancy on an SM.[4] This may negatively impact performance. Hence, GPUscout provides information about register pressure to help the user decide whether increasing an already high register pressure could potentially bring even more harm than what the vectorizing benefit is. GPUscout presents the number of additional registers needed by each SASS instruction, showing if/how much each instruction affects the overall register pressure. Finally, after implementing the modifications, the user can compare the new register pressure to the old values.

Warp Stalls. For a non-vectorized load, a high percentage of *long scoreboard* dependency stall points to a high number of cycles waiting on L1TEX operation (global or texture memory operation). Vectorized loads can reduce the number of stalled cycles by increasing the memory access bandwidth. Therefore, checking the *long_scoreboard* stall metric helps to assess the severity of the issue posed by non-vectorized loads.

4.2 Register Spilling

Register spilling occurs when a program needs more registers than are available. The data from registers must be temporarily stored (spilled) to the local memory so that other data can be loaded to the particular register. This causes an increase in instruction count and memory traffic, leading to decreased performance.

SASS Analysis. An instruction `STL R1, R2` indicates spilling(storing) the register `R2` to the local memory at the address given by `R1` (analogously for load `LDL`). To understand the cause of the spill, we need to find out which SASS instruction is to blame for the spill. Since spilling occurs if there are not enough registers to hold a variable, an optimistic assumption would be that the data written to the spilled register was due to an arithmetic operation. GPUscout, therefore, presents the registers that spilled, the relevant source code lines, and the last operation that caused the spill to happen.

Metrics. The spills based on high register pressure lead to supplementary data movement between registers and local memory. GPUscout collects this extra register pressure information and thus helps the user optimize the kernel to limit spills.

[4]Low occupancy results in inefficient issue of instructions because there are not enough available warps to hide latency between dependent instructions.

Since suboptimal memory usage negatively affects performance, our tool presents a detailed analysis of the data migration through the GPU caches. The kernel requests data, among others, from global and local memory. These requests are first received by the L1 cache. On an L1 cache miss, the request is forwarded to the multi-banked L2 cache. L1 cache miss percentages due to global and local memory data requests are obtained as separate `ncu` metrics. Using this miss percentage, the amount of data requested from the L2 cache is calculated as well.

```
({L1,L2} miss %) * (bytes requested from cache)
```

Similarly, on an L2 cache miss, the request continues to the slower DRAM. To further analyze the impact of register spilling on the GPU memory, the total number of data requests between the streaming multiprocessors (SMs) and L2 cache is measured and presented. Based on this, the user can assess whether register spills occur in a bandwidth-limited code, hence if reducing these spills can significantly increase performance.

Warp Stalls. It is mentioned in [20] that warps can be stalled waiting for a scoreboard dependency due to local memory operation. Moreover, the warp may stall waiting for local memory operations to fill the L1 instruction queue too. So, `long_scoreboard` and `lg_throttle` (Local and Global memory throttle) stalls are shown to the user to explain the impact of the register spill. On containing the spill, the user should observe a reduced portion of the aforementioned stalls.

4.3 Use Shared Memory

Shared memory offers lower latency than global memory. For shared memory to be useful, the data must be reused many times. Otherwise, the overhead of transferring the data to/from the shared memory will outweigh the benefits. GPUscout detects data that is accessed repeatedly as potential candidates for shared memory optimization.

SASS Analysis. The decision-making process to recommend the usage of shared memory instead of global memory is illustrated in Figure 4. For each address being accessed from global memory, GPUscout counts how many times the data from the particular address is loaded repeatedly. A second counter detects the number of arithmetic instructions involving said register. Additionally, the presence of the register inside a for loop denotes repeated calls to request data from the global memory, even amplifying the problematic behavior. Frequent arithmetic instructions on a register, especially inside a for-loop, are marked as likely candidates for using shared memory.

Metrics. Number-of-way bank conflicts are relevant when using shared memory. It describes how many threads access the same bank simultaneously, as the accesses will be

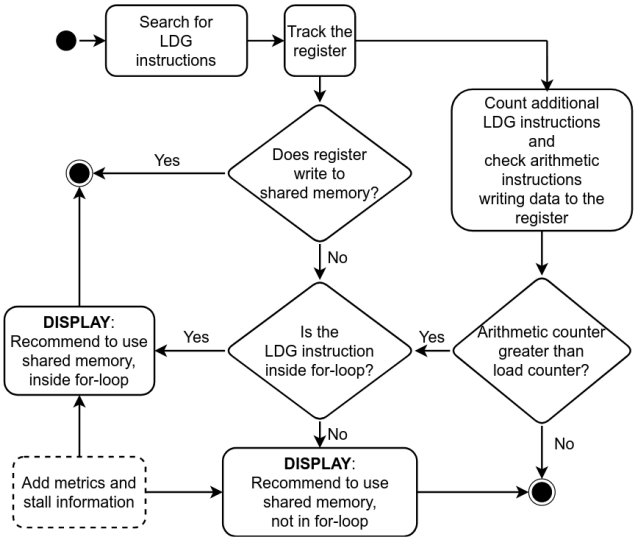

Figure 4. Simplified flowchart on the methodology to recommend using shared memory

serialized. Since ncu does not currently provide this metric (only number of bank conflicts is available), GPUscout approximates this value. It is computed as `# of shared memory load transactions / # of shared memory load accesses`, which are both obtained from ncu. One transaction per access indicates no bank conflicts, 32 transactions denote a 32-way bank conflict where all the accesses are serialized. Hence, the higher the ratio, the worse the pipeline performance, which is consistent with a low shared memory efficiency metric from ncu.

4.4 Use Shared Atomics

An atomic operation performs a read-modify-write operation by non-interfering threads on data residing in the global or shared memory [16]. Global atomic is a kernel-wide serialization, typically getting resolved in the L2 cache. Shared atomics, however, results in serialization at the block level, compared to device-wide. Of course, due to its nature, shared atomics only work as a synchronization point within one thread block, i.e., one SM. GPUscout displays the atomic instructions count for global and shared memory alongside the corresponding source code line numbers. Frequent global atomic instructions compared to shared atomics typically indicate a potential memory bottleneck. Therefore, GPUscout warns of global atomics especially detected in a for-loop, where repeated serialization even amplifies the performance degradation.

In such a scenario, `lg_throttle` warp stall will occur often. Using shared atomic can reduce this problem. On the other hand, shared atomics involve increased utilization of memory input/output (MIO) pipelines, potentially leading to a high number of *MIO* stalls. The user is therefore advised to watch out for *MIO* stalls after updating the atomics.

Finally, GPUscout also analyzes the atomic data movement in the GPU, usually resulting in 100 % L1 cache miss, and some atomics being resolved in the L2 cache, others in DRAM.

4.5 Use Read-only Cache

The NVIDIA nvcc compiler (starting from compute capability 3.5) offers a way to hint about pointer aliasing using the keyword `__restrict__`, thus hinting that the load operations may go through read-only caches. Using the cache designed for read-only data, the compiler can optimize more aggressively, especially the order of memory accesses, thereby reducing the number of memory accesses.

GPUscout implements a strategy to identify global memory accesses in the CUDA kernel that can potentially be marked with `__restrict__`. The SASS analysis observes for each such load instruction whether it is read-only throughout the kernel. For such registers, the user is advised to use the `__restrict__` keyword. The compiler optimizations due to restricted pointers can improve performance unless the corresponding register pressure is too high. Hence, GPUscout provides the register pressure information as well.

4.6 Use Texture Memory

Texture memory is a type of global memory with a dedicated cache, which is optimized for spatially-local accesses, where threads in a warp read memory addresses close by. To detect spatial locality, the SASS analysis of GPUscout seeks data accesses from adjacent global memory addresses, such as the following:

Listing 1. Example SASS for texure memory.

```
LDG.E.SYS R0, [R2] ;
LDG.E.SYS R5, [R4] ;
LDG.E.SYS R7, [R4+-0x8] ;
LDG.E.SYS R9, [R2+-0x8] ;
STG.E.SYS [R4], R9 ;
```

The example SASS code shows four loads from global memory. Loads to `R0` and `R9` fetch data from adjacent memory addresses `R2` and `R2+-0x8`. This proximity hints at some spatial locality to the read access pattern. If these registers are read-only, they are potential candidates for using texture memory. The same holds for loads to `R5` and `R7`.

Too many outstanding requests may fill the TEX pipeline, resulting in warp stalls. This can be observed through an increase in `tex_throttle` stalls. Additionally, with frequent texture operations, warps might stall waiting for a scoreboard dependency on texture memory data accesses. Therefore, the `long_scoreboard` stalls are to be observed.

4.7 Datatype Conversions

Datatype conversions are expensive operations on GPUs, as they increase the instruction count and might require

```
-- Vectorized load analysis for kernel: mixbench --

WARNING  :: Use vectorized load for register R2 at line 55 of your code.
It has 7 adjacent memory accesses by the compiler.


-- Shared memory analysis for kernel: mixbench --

WARNING  :: Use shared memory for register R9 at line 55 of your code.
It has 1 global load count and 43 computation instruction counts.
It seems to be in a for loop.
```

Figure 5. Tool output (Mixbench naive implementation)

frequent utilization of several pipelines on the GPU. Hence, conversions such as `F2F` (floating point to floating point) and `I2F` (integer to floating point) should be avoided, if at all feasible. To indicate that such conversions take place, GPUscout presents a total count of the conversions detected, alongside the corresponding line numbers in the source code.

5 Case Studies

We showcase the functionality of GPUscout by presenting three use-cases based on real-world applications and benchmarks. Our results demonstrate how the recommendation system operates. Furthermore, we reveal that significant performance gains can be achieved by adapting the kernels based on GPUscout's suggestions.

The analyses and measurements were carried out on an NVIDIA Tesla V100 GPU. The GPU (Volta microarchitecture) has 80 SMs × 64 threads and 16 GB DRAM[5].

5.1 Mixbench

Mixbench [10, 11] is a GPU benchmarking suite that aims at evaluating GPUs and CPUs on mixed operational intensity kernels. The CUDA implementation (`mixbench-cuda`), considered in this use-case, executes single-precision, double-precision, and integer-type multiply-add (MAD) operations.

We used GPUscout to examine the `benchmark_func` kernel. It returned the following suggestions:

1. Favoring shared memory and
2. Using vectorized global memory loads

for better performance, as shown in Figure 5.

A 32-bit global memory load fetches data from the register address `R2`. It also reads seven adjacent addresses. This can be improved by using a vectorized load for register `R2`, to load contiguous data more efficiently. For the double-precision implementation, GPUscout detected a 64-bit width vectorized read performed by the compiler. Apart from the register number, GPUscout also mentions line number 55, where this instruction originates. This line loads elements from `g_data` to `tmps`, which is a memory-intensive operation (cf. Listing 2).

[5]CUDA version 11.6.1, GCC version 11.2.0 and Linux kernel 5.3.18.

The second recommendation regarding shared memory corresponds to the same line number, as register `R9` is involved in multiple loads and computationally expensive arithmetic operations. Additionally, the instruction is in a for-loop, hence potentially repeating many times. This is what triggered the second bottleneck warning, asking the user to use shared memory to profit from its lower latency on repeated accesses.

We follow GPUscout's first suggestion – to use 128-bit wide vectorized memory access to load the data. Listing 2 shows the CUDA kernel modification for single-precision floating point data, where `float4* x` is 128-bit aligned using `reinterpret_cast<float4*>`. Since, the hardcoded vector size in the original benchmark is divisible by 4, we do not have to consider any potential remainder-loop. The previously reported memory-intensive load operation at line 55 is modified to use vector datatypes, namely `int4`, `double4`, or `float4`. Since the compiler lacks support of MAD operations for vector datatypes, the corresponding functions had to written. Consequently, the loop now runs for a quarter times since four elements are now handled in each iteration.

Listing 2. Code modification (Mixbench).

```
[OLD]
for(int j=0; j<granularity; j++)
 tmps[j] = g_data[...];
 ...
 for(int i=0; i<compute_iterations; i++)
  tmps[j] = mad(tmps[j],tmps[j],seed);

[NEW]
for(int j=0; j<granularity/4; j++)
 reinterpret_cast<float4*>(tmps)[j] =
  reinterpret_cast<float4*>(g_data)[...];
 ...
 for(int i=0; i<compute_iterations; i++)
  reinterpret_cast<float4*>(tmps)[j] =
   mad(reinterpret_cast<float4*>(tmps)[j],
    reinterpret_cast<float4*>(tmps)[j],seed);
```

We rerun the analysis on the updated kernel. Compared to the baseline, GPUscout showed a decrease in *long scoreboard* stalls from 70 % to 62 % per active warp for the modified kernel. This indicates fewer stalls due to waiting for a scoreboard dependency on a global memory load operation, as vectorized loads fetch multiple data elements in one transaction. Additionally, due to an increase in register pressure, the occupancy achieved dropped from 92 % to 83 %, thereby showing the effectiveness of the tool in pointing at potential caveats in using vectorized loads.

For an iteration count of 96, this simple optimization shows a **performance improvement of 3.77×, 3.86×, and 4.44×** for single-precision, double-precision, and integer `MAD` operations, respectively. This performance improvement is

primarily based on increased bandwidth utilization and a decreased number of instructions to execute during vectorized loads.

5.2 Heat Transfer Simulation

A 2D heat transfer simulation of an isotropic material is a computationally challenging problem. The Jacobi iterative solver [22] is a frequently-used approach to compute heat transfers on GPUs due to its highly parallel nature. It consecutively computes the time steps of our heat transfer problem. For each data point, an iteration of a 2D heat transfer is computed as:

$$T_{NEW} = T_{OLD} + k * (T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4 * T_{OLD})$$

So, the new value of each point depends on the neighboring cells.

We analyze such a Jacobi kernel with GPUscout, which recommended

1. using texture memory or shared memory,
2. using vectorized loads from the global memory,
3. favoring the `__restrict__` keyword, and
4. minimizing datatype conversions.

We update the 2D-stencil code to use texture memory (`tex2D()`) and rerun the GPUscout analysis. The report outlined that the warp stalls due to *TEX throttle* had considerably moved up to 24.65 % (from 0 % because of no TEX operations) due to high utilization of the TEX pipeline. The analysis of the naive implementation already warned us to look after this metric.

The metric analysis outlined that the texture memory requested 221 760 B of data to the texture cache. 11.5 % of those missed the texture cache and were forwarded to the L2 cache, which in the worst case would continue to DRAM. A high percentage of such cache misses would poorly impact the code performance.

After switching to texture memory, the throughput increases by 61.1 % compared to the naive implementation for a problem size of 8192×8192 elements. This leads to a **performance improvement** (kernel execution duration) **of 39.2 %**.

Using textures can greatly benefit the performance, however, the legacy texture API is cumbersome to use and maintain. A viable alternative, as also suggested by GPUscout, would be using shared memory instead, which is exposed in a more user-friendly way to CUDA developers. Multiple efforts, such as [1, 4, 24] present the algorithms and explore the benefits of using shared memory in stencil computations on GPUs.

Following the recommendation of GPUscout to add the `__restrict__` keyword had very little effect, improving the performance by only 0.3 %. This little performance gain can be attributed, for example, to some hardware-based optimizations happening to the original kernel, however, a closer analysis is needed in this case.

On the datatype conversion front, our tool points at six I2F (integer to floating point) datatype conversions at their respective source code line numbers. However, these conversions are unavoidable due to the nature of the algorithm.

5.3 SGEMM Matrix Multiplication

A single float precision General Matrix Multiply (SGEMM) kernel computes matrix-matrix operations of the form

$$C \leftarrow \alpha AB + \beta C$$

In this use-case, our starting point is a naive implementation, where each GPU thread computes a dot product of a row of matrix `A` and a column of matrix `B`, computing a single element of `C`.

GPUscout analysis of the SGEMM kernel detects registers that are read-only throughout the lifetime of the kernel, and are not aliased, making them suitable to be marked with `__restrict__` and `const` keywords. To make the change, the tool again provides the exact source code line numbers. GPUscout's second recommendation is to use shared memory. Again, the registers involved in multiple load and computation operations are shown, including their potential presence in a for-loop. The analysis also advises the user to pay attention to the shared memory bank conflicts and a higher number of *long scoreboard* and *MIO throttle* stalls when adjusting the code to use shared memory.

Based on the recommendations, we modify the kernel to use shared memory. Roughly, we load a tile of `A` and of `B` from global memory to the shared memory of each block. The computation then takes place on the shared memory, accumulating the partial results for each tile of `C`.

Comparing the metrics to the baseline ones, the *long scoreboard* stalls rise from 7.8 % to 30.6 % and *MIO throttle* stalls from 0.03 % to 4.5 %, showcasing the effective warning system of GPUscout. The increase in the *MIO throttle* stall is associated with high utilization of the MIO pipeline due to shared memory instructions. However, the measured increase is not enough to cause a significant performance drop. For a matrix size of 10240×10240, the **total kernel runtime improves by a factor of 54×** compared to the baseline.

Next, we analyze the new shared memory kernel, and it newly recommends a vectorized load optimization. Therefore, we further modify the kernel to use vectorized 128-bit-aligned (`<float4>`) loads. GPUscout warns of a rise in register pressure from using 25 registers (original) to 72 (vectorized), hence reducing the overall occupancy. This optimization yields an **additional performance improvement of 8.5 %** for the matrix size used.

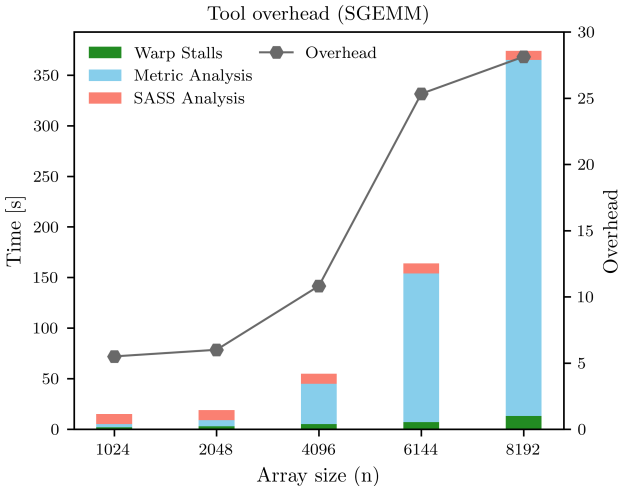

Figure 6. GPUscout measurement overhead

5.4 Performance Analysis

As the analysis of kernels with GPUscout needs to utilize the device (GPU) for some time, we provide basic information about the overhead it comes with. As described in Section 3, there are three major tasks when analyzing a kernel – the static SASS Analysis, warp stall sampling, and collecting the performance metrics. Each of these components in the workflow adds to the total overhead of GPUscout.

Figure 6 presents the time needed for analysis of the SGEMM kernel from Section 5.3 for different matrix sizes. Each of the three pillars is presented separately. We can see that the collection of metrics through NVIDIA Nsight Compute is the most prominent contributor to the tool performance overhead, especially for larger problem sizes. The overhead is proportional to the sampling frequency and the number of metrics collected. For this reason, the number of metrics is kept as low as possible. The time spent with the PC Stall Sampling analysis also increases with the problem size, although being significantly less compared to the metrics collection. On the other hand, the SASS analysis is independent of the kernel execution duration. Instead, it depends on the number and complexity of bottlenecks inspected by GPUscout and the size of the SASS files being analyzed. As a result, this value remains relatively constant, making it a dominant factor for short kernels, but negligible otherwise.

Figure 6 presents the overhead of GPUscout compared to the bare kernel execution time. As the array size increases, the overhead rises too, reaching a factor of 28× for a matrix size 8192×8192. This is caused by the rapid rise in metric collection duration using Nsight Compute, which overshadows the contribution from the other sections of GPUscout.

To avoid the overheads, the `--dry-run` option saves time and resources by skipping the expensive metric collection section of the tool, as presented in Section 3.1.

6 Related Work

Performance analysis and optimization tools are integral instruments in the development of GPU applications. Nowadays, there is a rich ecosystem of tools that assist developers in adapting and optimizing their GPU applications. Some of the tools are vendor-provided and hence tied to a particular GPU platform, while others are not. Some tools are commercial, while others are open-source. As adding GPU support in performance analysis tools is a recent development, the support for various GPU features varies considerably.

NVIDIA's proprietary Nsight suite offers comprehensive profiling and optimization details. Nsight Systems and Nsight Compute are two tools dividing the workflow into system-level workload overview and CUDA kernel-level profiling. NVIDIA Nsight Compute (`ncu`) [21] is an interactive cross-platform low-level kernel profiler for CUDA programs. It provides a detailed performance overview via a user interface and/or a command line interface (CLI) by sampling metrics from the kernel. These metrics only provide information at a kernel-level granularity, hence do not provide details about particular code segments.

Rice University's HPCToolkit Performance Tool [25] is capable of registering callbacks to monitor asynchronous GPU operations using the NVIDIA CUPTI Activity API. Additionally, it supports coarse-grained and fine-grained profiling of GPU activities. HPCToolkit comes with a slightly larger overhead than `nvprof` because, unlike `nvprof`, HPCToolkit gathers CPU call stack data on each GPU profiling pass as well. Unfortunately, there has not been much effort invested in improving the tool's measurement overhead [25]. GPA [26], a similar performance advisor for NVIDIA GPUs, uses PC Sampling and dynamic backward slicing to detect the stall reasons. However, unlike GPUscout, it lacks metrics data to understand the influence of the GPU hardware.

DrGPU [7], a top-down GPU profiler, utilizes NVIDIA Nsight Compute to collect the required hardware metrics and builds an analysis-tree with rich performance insights. This helps DrGPU to show memory-access related performance bottlenecks. In contrast to our tool, it misses ingesting CUDA SASS code information and therefore does not provide actionable optimization advice at an instruction-level.

TAUcuda measurement library [14] enables capturing CUDA events asynchronously in profile and trace forms that provide static and dynamic performance measurement and analysis of GPU kernels. Static analysis involves disassembling CUDA binary files to observe instruction mixes and source line information. The dynamic analysis uses CUPTI source-code locator information and metrics for hardware counter sampling [12]. Currently, it only supports default counters, like `occupancy %`, `registers per thread`, etc. With GPUscout, we employ a larger range of metrics to provide a more detailed and comprehensive analysis of the kernel performance. Moreover, the design of the TauCUDA

package is based on an experimental Linux CUDA driver from NVIDIA. As a result, the practical use of the tool depends heavily on NVIDIA's tools support in future Linux production systems [13].

Score-P measurement infrastructure [8] is a scalable suite for profiling and event tracing of HPC applications. It is primarily an instrumentation-based tool that is limited to basic functionality on GPUs. It has connections to other popular tools like Scalasca [6], Vampir [15], and TAU. It enables recording CUDA function calls and GPU events with the CUPTI API. Although GPUscout adopts the use of CUPTI PCSampling API too, in addition, we correlate the stalls to the inefficient regions of source code and focus on analyzing the SASS code as well. This helps to derive several optimization recommendations for tuning the kernel performance.

The main added value of GPUscout, compared to the above-mentioned tools, is offering a systematic approach to decipher the CUDA binary code and analyze patterns of data movements between the cores and the GPU memory. This, combined with PC stalls and performance metrics, enables detecting the opportunities for optimizations that would otherwise be difficult to foresee merely at a source code level.

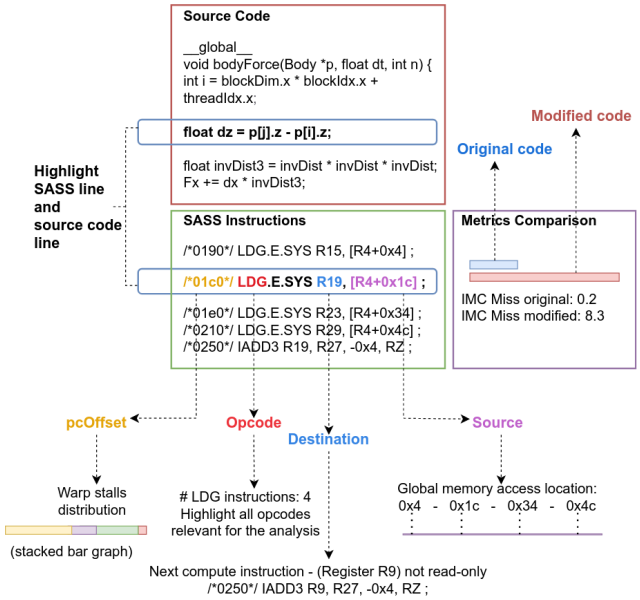

Figure 7. Schematic sketch of visualizing the data collected

7 Conclusion and Future Work

Due to the complex memory architecture and programming model of (NVIDIA) GPUs, optimizing workloads performed on GPUs in HPC applications is a challenging task. To help the developers with this task, GPU profilers and performance optimization tools were created. Although they can provide lots of information about kernel behavior, including metrics, aggregated sample values, etc., it remains very difficult to interpret the provided information to identify the root cause of potential problems and locate these directly in the source code.

To address this gap, in this paper, we introduced GPUscout, which provides a framework for a comprehensive analysis of NVIDIA GPU kernels and combines static analysis with measurement-based data to provide additional context unveiling the performance and behavior of the kernel. The focus of GPUscout is to provide precise and useful information, hence the problem description and source code line number are always attached. The analysis of GPUscout consists of three individual parts – the analysis of the SASS code, collection of warp stall sample aggregates, and collection of kernel-wide metrics – that are linked together to provide the needed information to the user.

We highlighted three use-cases in which we utilized GPUscout to examine various kernels. It presented several optimization suggestions. We implemented multiple of them and saw significant performance improvements in the majority of cases. The suggested and implemented optimizations included using shared or texture memory, vectorized loads, or the `__restrict__` keyword. These use-cases demonstrate the usefulness of GPUscout and present how the analysis engines come together to help interpret the kernel performance data and improve the code.

Future Work. In the future, we plan to add a more visual way of presenting the findings and relevant data in an easily understandable way. A sketch of the planned frontend is illustrated in Figure 7. The main window can be divided broadly into the 'Source Code' view (presenting the CUDA code) and the 'SASS Instructions' view (presenting the SASS instructions). These windows can easily be correlated with each other through the code line/SASS instruction mapping. Another section called 'Metrics Comparison' will focus on the data movement in the GPU hierarchy. It will point at metrics to observe after modifying the code, and hence, a new-versus-old comparison of the obtained metric values will be available here, showing how selected metrics rise/fall due to the change. Such a simple, yet powerful interactive visualization can move the analysis provided by GPUscout to a new level by presenting the findings in a more user-friendly and interactive way.

Additionally, due to the modular nature of GPUscout, more SASS analyses can be added very easily, detecting much more potential optimization scenarios than now.

Finally, we plan to examine options of injecting PTX instructions around specific code regions of interest to collect further metrics characterizing memory bottlenecks.

Acknowledgments

The DEEP-SEA project has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606. National contributions from the involved state members match the EuroHPC funding.

Part of the performance results have been obtained on systems in the test environment BEAST (Bavarian Energy Architecture & Software Testbed) at the Leibniz Supercomputing Centre.

References

[1] Ronan Amorim, Gundolf Haase, Manfred Liebmann, and Rodrigo Santos. 2009. Comparing CUDA and OpenGL implementations for a Jacobi iteration, In 2009 International Conference on High Performance Computing and Simulation. *Proceedings of the 2009 International Conference on High Performance Computing and Simulation, HPCS 2009*, 22 – 32. https://doi.org/10.1109/HPCSIM.2009.5192847

[2] Lorenz Braun and Holger Fröning. 2019. CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 73–81. https://doi.org/10.1109/PMBS49563.2019.00014

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.* 14, 3 (aug 2000), 189–204. https://doi.org/10.1177/109434200001400303

[4] José Cecilia, José García, and Manuel Ujaldon. 2010. CUDA 2D Stencil Computations for the Jacobi Method, In Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing - Volume Part I (Reykjavík, Iceland). *Para 2010 - State of the Art in Scientific and Parallel Computing* I, 173–183. https://doi.org/10.1007/978-3-642-28151-8_17

[5] Gautam Chakrabarti, Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, and Jian-Zhong Wang. 2012. CUDA: Compiling and optimizing for a GPU platform. *Procedia Computer Science* 9 (12 2012), 1910–1919. https://doi.org/10.1016/j.procs.2012.04.209

[6] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 2010. The Scalasca Performance Toolset Architecture. *Concurr. Comput.: Pract. Exper.* 22, 6 (apr 2010), 702–719.

[7] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A Top-Down Profiler for GPU Applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23)*. Association for Computing Machinery, New York, NY, USA, 43–53. https://doi.org/10.1145/3578244.3583736

[8] Andreas Knüpfer, Christian Feld, Dieter Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir.* Springer, Berlin, Heidelberg, 79–91. https://doi.org/10.1007/978-3-642-31476-6_7

[9] Peter Kogge and John Shalf. 2013. Exascale Computing Trends: Adjusting to the New Normal for Computer Architecture. *Computing in Science & Engineering* 15 (11 2013), 16–26. https://doi.org/10.1109/MCSE.2013.95

[10] Elias Konstantinidis. 2015. mixbench. https://github.com/ekondis/mixbench,. commit: 8a3585e3cf32a062192396cbc560afe6abb566d0.

[11] Elias Konstantinidis and Yiannis Cotronis. 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parallel and Distrib. Comput.* 107 (04 2017), 37–56. https://doi.org/10.1016/j.jpdc.2017.04.002

[12] Robert V. Lim, Allen D. Malony, Boyana Norris, and Nicholas Chaimov. 2015. Identifying Optimization Opportunities Within Kernel Execution in GPU Codes. In *Euro-Par Workshops*.

[13] Allen D. Malony, Scott Biersdorff, Wyatt Spear, and Shangkar Mayanglambam. 2010. An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using CUDA. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1810085.1810105

[14] Shangkar Mayanglambam, Allen D. Malony, and Matthew J. Sottile. 2009. Performance Measurement of Applications with GPU Acceleration using CUDA. In *International Conference on Parallel Computing*.

[15] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer* 63, Vol. XII, 1 (1996), 69–80. https://juser.fz-juelich.de/record/189233

[16] NVIDIA. 2020. CUDA, release: 10.2.89. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2023-04-15.

[17] NVIDIA. 2022. CUDA Profiling Tools Interface (CUPTI), release: 11.8.0. https://docs.nvidia.com/cuda/cupti/index.html. Accessed: 2023-04-15.

[18] NVIDIA. 2023. CUDA Binary Utilities, release: 12.0. https://docs.nvidia.com/cuda/cuda-binary-utilities/. Accessed: 2023-04-15.

[19] NVIDIA. 2023. CUDA Profiler, release: 12.1. https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf. Accessed: 2023-04-15.

[20] NVIDIA. 2023. Kernel Profiling Guide, release: 2022.4.1. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html. Accessed: 2023-04-15.

[21] NVIDIA. 2023. Nsight Compute CLI, release: 2022.4.1. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html. Accessed: 2023-04-15.

[22] Huabin Ruan, Xiaomeng Huang, Haohuan Fu, and Guangwen Yang. 2013. Jacobi Solver: A Fast FPGA-based Engine System for Jacobi Method. *Research Journal of Applied Sciences, Engineering and Technology* 6 (12 2013), 4459–4463. https://doi.org/10.19026/rjaset.6.3452

[23] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. https://doi.org/10.1177/1094342006064482

[24] Siham Tabik, Maurice Peemen, Nicolas Guil, and Henk Corporaal. 2015. Demystifying the 16 x 16 thread-block for stencils on the GPU. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3591. *Concurrency and Computation: Practice and Experience* 27, 18 (2015), 5557–5573. https://doi.org/10.1002/cpe.3591 Accessed: 2023-04-15.

[25] Keren Zhou, Laksono Adhianto, Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Mark Krentel, Yumeng Liu, Xiaozhu Meng, and John Mellor-Crummey. 2021. Measurement and Analysis of GPU-Accelerated Applications with HPCToolkit. *Parallel Comput.* 108, C (dec 2021), 12 pages. https://doi.org/10.1016/j.parco.2021.102837

[26] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 115–125. https://doi.org/10.1109/CGO51591.2021.9370339

Received 19 Aug 2023; revised 30 Sep 2023; accepted 30 Sep 2023