Understanding Performance Problems in CUDA Programs

YUYANG BI, Fudan University, China
JUNMING CAO, Fudan University, China
YOU LU, Fudan University, China
BIHUAN CHEN*, Fudan University, China
TIANWEI GAN, Fudan University, China
DINGJI WANG, Fudan University, China
XIN PENG, Fudan University, China

With the wide adoption of GPUs, CUDA programming has become essential for leveraging GPU parallelism. However, its complex programming model poses challenges in performance optimization. Consequently, CUDA programs often suffer from performance problems. In that sense, it is crucial to understand the performance problems specific to CUDA programming. Unfortunately, no systematic study has been conducted in literature.

To bridge this gap, we conduct the first systematic study to 1) characterize the symptoms and root causes of 216 performance problems collected from 55 StackOverflow posts and 122 NVIDIA forum posts, and 2) measure the speedup of fixing performance problems, and assess the capability of existing performance analysis methods in identifying performance problems, using a dataset of 69 reproduced performance problems. Our findings provide practical guidance for developers, and opportunities for researchers to advance performance analysis.

CCS Concepts: • **Software and its engineering** → **Software performance**.

Additional Key Words and Phrases: CUDA Programs, Performance Problems, Performance Analysis

ACM Reference Format:
Yuyang Bi, Junming Cao, You Lu, Bihuan Chen, Tianwei Gan, Dingji Wang, and Xin Peng. 2026. Understanding Performance Problems in CUDA Programs. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE016 (July 2026), 22 pages. https://doi.org/10.1145/3797143

1 Introduction

With the increasing demand for computational power, GPU computing has become a core technology in modern high-performance computing due to GPUs' parallel computing capability [12]. Specifically, NVIDIA's CUDA (Compute Unified Device Architecture) [30] is a parallel computing platform and programming model. It allows developers to fully leverage the potential of GPUs and write CUDA programs that can efficiently run on GPUs. With continuous advances, CUDA's influence in various areas, especially in deep learning and scientific computing, has become indispensable [29].

*Bihuan Chen is the corresponding author.

Authors' Contact Information: Yuyang Bi, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, 24110240003@m.fudan.edu.cn; Junming Cao, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, 21110240004@m.fudan.edu.cn; You Lu, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, ylu24@m.fudan.edu.cn; Bihuan Chen, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, bhchen@fudan.edu.cn; Tianwei Gan, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, twgan25@m.fudan.edu.cn; Dingji Wang, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, wangdj25@m.fudan.edu.cn; Xin Peng, Fudan University, College of Computer Science and Artificial Intelligence, Shanghai, China, pengxin@fudan.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE016
https://doi.org/10.1145/3797143

Motivation. Performance is a key metric for evaluating the efficiency of programs in GPU computing [52]. While powerful for leveraging GPU parallelism, CUDA programming presents significant challenges in performance optimization because of its complex programming model and steep learning curve. The intricacies of thread hierarchy, memory management, and kernel execution demand a deep understanding of both the hardware architecture and the software tools involved. This complexity leads to high learning cost, especially for developers accustomed to traditional CPU-based programming model. Consequently, developers face a barrier to entry when attempting to optimize the performance of their CUDA programs for GPUs.

Moreover, traditional performance optimization techniques are primarily designed for serial programming on CPUs, and do not scale effectively with CUDA's highly parallel architecture. For example, optimizations for CPU single-thread are designed to enhance the serial execution of individual cores [11], whereas CUDA programs leverage a massive number of threads, posing challenges in inter-thread coordination, scheduling, and large-scale parallelism [13]. CPUs typically leverage well-defined cache levels (i.e., L1, L2, L3) to improve data access speed [38]. In contrast, CUDA employs a distinct memory hierarchy, requiring careful optimization of memory coalescing and access patterns. CPUs use branch prediction to minimize performance losses caused by conditional jumps, reducing pipeline stalls [22]. However, CUDA lacks advanced branch prediction mechanisms, limiting the effectiveness of traditional branch optimization techniques. CPUs often utilize SIMD (Single Instruction, Multiple Data) techniques for optimization, where a single instruction is executed on multiple data points [17]. In contrast, CUDA adopts SIMT (Single Instruction, Multiple Threads) model, where multiple threads execute the same instructions simultaneously while processing different data points. Such gaps in performance optimization techniques exacerbate the challenges developers face when trying to optimize the performance CUDA programs.

Therefore, CUDA programs could suffer from performance problems that not only affect the efficiency of computational tasks but also lead to the underutilization of hardware resources. It is crucial to characterize the performance problems specific to CUDA programming, which helps lower the barrier to entry for CUDA program optimization and inspire automated performance optimization techniques. Unfortunately, no systematic study has been conducted in literature.

Our Empirical Study. To bridge this knowledge gap, we conduct the first systematic study to understand performance problems in CUDA programs. Specifically, we collect 216 performance problems from 55 StackOverflow posts and 122 NVIDIA forum posts. We manually investigate these performance problems to build a taxonomy of their symptoms (**RQ1**) and root causes (**RQ2**). Moreover, we construct a dataset of 69 performance problems by reproducing the collected performance problems, which covers most symptoms and root causes. Using this dataset, we quantify the speedup of fixing performance problems (**RQ3**), and assess the capability of existing performance analysis methods in identifying these performance problems (**RQ4**).

- **RQ1**: What are the symptoms of performance problems?
- **RQ2**: What are the root causes of performance problems?
- **RQ3**: How much speedup can be achieved by fixing performance problems?
- **RQ4**: To what extent can performance analysis methods identify performance problems?

These research question analyses provide useful findings for both developers and researchers. For example, 81.0% of the performance problems exhibit a long kernel execution time. 61.1% of the performance problems are introduced by memory issues, warp issues, and API misuses. Fixing these performance problems results in varying levels of performance improvement, achieving an average speedup of 2.14. Existing performance analysis techniques Nsight Compute [34] and Nsight Systems [35] have a limited capability in identifying performance problems; i.e., they are respectively only

applicable to 53.6% and 13.0% of the performance problems. 27.5% of the performance problems are not covered by CUDA programming guide [8].

These findings highlight the need for developers to deepen their understanding of the CUDA programming model, prevent memory-related and warp-related performance problems, and master the use of existing API libraries. These findings also provide researchers with research opportunities, e.g., semantic-aware performance analysis techniques to identify API-related and algorithm-related performance problems, intelligent API recommendation techniques to recommend efficient APIs, and automated performance optimization techniques to fix performance problems.

Contribution. This work makes the following contributions.

- *Empirical Collection of Performance Problems.* We collected 216 performance problems in CUDA programs from two sources, i.e., 55 StackOverflow posts and 122 NVIDIA forum posts.
- *Taxonomy of Symptoms and Root Causes.* We constructed a systematic taxonomy of the symptoms and root causes of these performance problems.
- *Dataset Construction and Evaluation.* We constructed a dataset through reproducing 69 of these performance problems, measured the speedup of fixing performance problems, and assessed the capability of existing performance analysis methods in identifying performance problems.

2 Preliminaries on CUDA Programming Model

To simplify the development of parallel programs on GPUs, the CUDA programming model extends C/C++, and provides a certain level of abstraction for the unique hardware architecture of GPUs. According to the CUDA C++ programming guide [28], the core concepts of the CUDA programming model are introduced as follows for the ease of paper presentation.

Kernel. A kernel is a special C++ function that is executed in parallel by different CUDA threads. When a CUDA program is executed, CUDA launches multiple threads to execute the kernel. These threads typically execute the full kernel code, but each thread processes different data. Each thread can access specific data using its unique thread identifier, thus enabling parallel data processing.

Thread Hierarchy. Threads are organized into a hierarchy of three levels, i.e., threads, thread blocks, and grids. Multiple threads form a thread block, and multiple thread blocks form a grid. Typically, each kernel uses a single grid, and the grid size and block size for each kernel are specified when the kernel is launched. Thread blocks are independent, while threads within a thread block can share data and synchronize. Each thread block typically executes on a streaming multiprocessor (SM), a hardware unit in a CUDA device responsible for scheduling and executing threads in parallel.

CUDA follows the SIMT (Single Instruction, Multiple Threads) architecture, where multiple threads execute the same instructions simultaneously while processing different data. In SIMT, threads are grouped into warps, and each warp typically contains 32 threads that execute instructions in parallel. During kernel execution, all threads in a thread block are divided into multiple warps, and the scheduling of these warps is handled by the hardware. The number of thread blocks in a CUDA program can far exceed the number of SMs, and the scheduling of thread blocks is managed by the CUDA runtime system. Generally, the thread hierarchy, combined with the SIMT execution, allows CUDA to flexibly manage a large number of parallel tasks.

Memory Hierarchy. GPUs have multiple types of memory with varying access speeds, capacities, and accessibility ranges. Fig. 1 is the memory hierarchy of the CUDA programming model. Each thread has its own registers and local memory. The register space is the smallest and fastest, and variables defined within a thread are preferentially stored in registers. If the register space is limited, they are stored in the larger but slower local memory. Shared memory is visible in each thread block, where data is accessible by all threads within the thread block. Global memory is globally visible and can be accessed by all threads. It typically has a larger capacity but a slower read

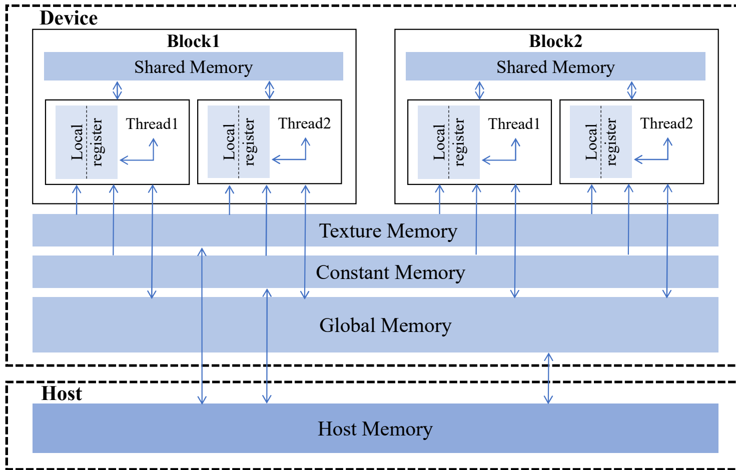

Fig. 1. Memory Hierarchy [28]

speed. Constant memory and texture memory are also part of the global memory, but are optimized for specific access patterns. Constant memory is read-only for threads on the GPU, but can be modified on the host, making it suitable for storing constants that do not change during runtime. Texture memory has a special caching mechanism and is suitable for image processing tasks.

Heterogeneous Programming Model. For CPU-GPU heterogeneous computing, a typical CUDA program execution often has six steps, illustrated by an example of a CUDA program in Fig. 2.

- *Host Code Execution.* When a CUDA program begins execution, the regular serial code runs first on the host (CPU), including tasks such as initialization and data loading. As illustrated at Lines 8–14 of Fig. 2, the host program sets parameters required by the kernel, and allocates memory.
- *H-D Data Transfer.* In a CUDA program, data is typically transferred from the host to the device (GPU). As shown at Lines 15-21 of Fig. 2, the data transferred may include the input data to be processed, the parameters, etc. This is done through CUDA's data transfer API, which copies data from host memory to device memory.
- *Kernel Launch.* Once the data is prepared and transferred from the host to the device, the CUDA kernel is launched. As shown at Lines 22–26 of Fig. 2, the launch involves scheduling the kernel code for execution on GPU and assigning tasks to each thread of the kernel.
- *Kernel Execution.* After the kernel is launched, the GPU executes the kernel code. As shown at Lines 1–6 of Fig. 2, each thread performs the same operations but processes different data. The kernel execution mainly occurs on the GPU, and it is the core computation of the CUDA program.
- *D-H Data Transfer.* After kernel execution, to obtain the output data, the result must be transferred from the device memory back to the host. As illustrated at Lines 29–30 of Fig. 2, this step is also done through CUDA's data transfer API.
- *Host Code Continues Execution.* The program returns to the host to continue with other tasks, such as processing the returned data, displaying results, or performing subsequent computations.

Thus, the total execution time of a CUDA program is the sum of the time for host code execution, the time for H-D and D-H data transfer, the time for kernel launch, and the time for kernel execution.

3 Empirical Study Design

Our goal is to understand performance problems in CUDA programs. To achieve this goal, we design four research questions, which are introduced in Sec. 1. Specifically, the symptom analysis in **RQ1**

```
// 4. Kernel execution
__global__ void vectorAdd(const float *A, const float *B, float *C, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n)
        C[idx] = A[idx] + B[idx];
}
int main() {
    // 1. Host code execution
    float *h_A, *h_B, *h_C;
    h_A = new float[N]; h_B = new float[N]; h_C = new float[N];
    for (int i = 0; i < N; i++) {
        h_A[i] = i * 1.0f;
        h_B[i] = (N - i) * 1.0f;
    }
    // 2. H-D data transfer
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, N * sizeof(float));
    cudaMalloc((void**)&d_B, N * sizeof(float));
    cudaMalloc((void**)&d_C, N * sizeof(float));
    cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice);
    // 3. Kernel launch
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();
    // 5. D-H data transfer
    cudaMemcpy(h_C, d_C, N * sizeof(float), cudaMemcpyDeviceToHost);
    // 6. Host code continues execution
    std::cout << "Result: " << h_C[0] << ", " << h_C[N-1] << std::endl;
    // free memory
    delete[] h_A; delete[] h_B; delete[] h_C;
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    return 0;
}
```

Fig. 2. An Example of Heterogeneous Programming Model

is designed to categorize the observable symptoms of performance problems. It can characterize the impact of performance problems, and provide insights for detecting performance problems. The root cause analysis in **RQ2** is designed to categorize the underlying causes of performance problems. It can offer insights for localizing performance problems. The speedup analysis in **RQ3** is designed to measure the performance improvement brought by fixing performance problems. It can demonstrate the severity of performance problems and reflect the necessity of performance optimization. The approach assessment in **RQ4** is designed to evaluate existing approaches for identifying performance problems. It can shed light on advanced performance problem detection and localization methods. These RQs can also provide hints for developing high-performance CUDA programs.

3.1 Performance Problem Collection

We collected performance problems in CUDA programs from two sources, i.e., StackOverflow and the official NVIDIA forum. These sources were selected based on three considerations, i.e., scale (sufficient number of posts), representativeness (real-world developer questions), and relevance to CUDA. StackOverflow, the largest developer Q&A site, features a dedicated CUDA tag, and is also used in prior work [45, 47] whose goal is to categorize GPU programming challenges, and investigate the usage of formal methods in GPGPU programming, which is different from ours. The NVIDIA forum specifically focuses on CUDA-related issues. In particular, their representativeness comes

from the fact that they capture real-world CUDA performance discussions from developers with diverse experience levels, including high-reputation contributors on StackOverflow and active participation from NVIDIA engineers on the official forum. This helps mitigate the bias toward only beginner-level or easily reported problems. StackOverflow posts were collected via official API, while NVIDIA forum posts were obtained through web scraping due to the lack of API. We carried out the following collection process on both sources.

CUDA-Related Post Selection. We first attempted to identify CUDA-related posts. To this end, we selected the posts that had the CUDA tag, but further excluded old posts that were published before January 1, 2020 (which ensured that our analysis focused on recent and currently widely used CUDA version). To ease manual analysis, we removed the posts that lacked source code in their problem descriptions. Furthermore, to concentrate on meaningful discussions, we filtered out posts that did not have any answers. After applying these selection criteria, we obtained a total of 1,225 posts from StackOverflow and 1,792 posts from the NVIDIA forum.

Performance Problem Post Selection. Rather than directly adopting performance-related keywords from prior research on traditional system performance analysis [24, 39, 42, 51], we formulated a custom set of keywords to comprehensively search performance-related posts. We first randomly sampled 50 posts labeled with the "performance" tag from the total 3,017 posts, and manually examined them to extract performance-related keywords. We continued this iterative process of random sampling and manual keyword extraction for five additional rounds until no new keyword was extracted. Altogether, 300 sampled posts contributed to the final set of keywords used for searching performance-related posts, including "performance", "slow", "fast", "speed up", "accelerate", "latenc", "contention", "optimiz", "efficient", "stride", and "reduc". Our keyword selection prioritized minimizing false negatives to avoid missing relevant posts. The resulting keyword set achieved a precision of 51.58%, and a recall of 100.00%, based on evaluation over the 300 sampled posts. Using this keyword set, we searched the problem descriptions from the 3,017 posts obtained in the CUDA-related post selection step, yielding 268 candidate performance problem posts from StackOverflow and 686 candidate performance problem posts from the NVIDIA forum.

Performance Problem Identification. Finally, we manually confirmed the 954 candidate performance problem posts to ensure that they discussed CUDA-related performance problems. Some posts merely contained performance-related keywords without discussing actual performance problems, some posts reflected misunderstandings or incorrect assumptions about performance problems, while others touched on performance but lacked valid solutions, making them unsuitable for characterizing performance problems. Two of the authors conducted the confirmation independently and resolved disagreements with a third author. The inter-rater agreement, measured using Cohen's Kappa coefficient [7], reached 0.8487. Any discrepancies were resolved through discussion. Ultimately, we identified 55 performance problem posts from StackOverflow and 122 from the NVIDIA forum, resulting in a final collection of 216 performance problems in CUDA programs. Notice that some posts contain multiple performance problems (i.e., 20 posts contain 2 problems, 8 posts contain 3 problems, and 1 post contains 4 problems), and we split them for separate analysis. Therefore, our analysis is based on each performance problem but not each post.

3.2 Symptom and Root Cause Labeling

To answer **RQ1** and **RQ2**, two of the authors labeled each of the 216 performance problems from two aspects, i.e., symptoms and root causes, by carefully going through all the content in the posts, including the titles, problem descriptions, comments, answers, and reference links mentioned in the discussion. Overall, we started with initial categories based on domain knowledge of CUDA performance optimization, incrementally added new categories when existing ones could not cover a case, and merged semantically similar categories after reviewing all cases to construct the final

taxonomy. Specifically, if the post explicitly mentioned the symptom of a performance problem, the symptom was directly determined. Otherwise, if the post only mentioned long program execution time, we classified it as either kernel execution time or data transfer time based on the parts of the program that were reported to have high time overhead. The root cause of a performance problem was identified by comparing the buggy code version in the problem description and the fixed code version in the valid answers. This process ensures that each identified root cause is grounded in concrete evidence by linking it to an explicit code change and its corresponding performance improvement. To further ensure the reliability of labeling, both annotators independently labeled all cases and resolved disagreements through discussion. The Cohen's Kappa coefficient was 0.9615 and 0.9652 for the labeling of symptom and root cause.

3.3 Dataset Construction via Reproduction

To answer **RQ3** and **RQ4**, we built a dataset by reproducing the performance problems. Our reproduction was carried out on a machine equipped with a 16-core Intel i7-7820X CPU (3.60GHz), an NVIDIA GTX 3090 GPU, 128GB RAM, and 1TB SSD. We utilized a CUDA Docker image to set up the runtime environment, with only the NVIDIA GPU driver installed on the physical machine. For each performance problem, we followed three steps to reproduce it.

Reproduce Buggy Code. We first extracted the original kernel exhibiting performance problems as provided by the post. Since developers often include only the buggy code directly related to the problem, these buggy code snippets are frequently incomplete. If the original buggy code was complete, we used it directly. Otherwise, we supplemented the provided code snippet with additional necessary components, guided by the problem description and provided answers, to form a complete and runnable buggy code.

Reproduce Fixed Code. For the fixed code, we relied on the fixed kernel provided in the post when available. If only a fix for a part of the code block was provided, we replaced the corresponding segment in the original kernel with the fixed code block and made necessary modifications to other parts affected by the change. As with the buggy code, if the fixed code snippet was incomplete, we completed it based on the accompanying context.

Run and Measure Speedup. We executed both the buggy code and the fixed code to replicate the reported symptom. Kernel launch parameters or other aspects of the code were adjusted as needed to reproduce the described behavior, given that our hardware environment may differ from that of the post. Then, we measured the speedup of the fixed code across four input data scales (i.e., 2^{20}, 2^{23}, 2^{26}, and 2^{29} bytes), averaging the speedup over 10 repeated runs. When measuring the speedup, we used Nsight Systems [35] to accurately measure the execution time of CUDA programs.

Through four person-months of manual effort, we successfully reproduced 69 of the 216 performance problems. The main reasons for reproduction failures were two-fold. First, developers provided very short and concise code snippets in the posts, making it infeasible to complete the buggy or fixed code. Second, some performance problems required specific hardware configurations that were different from our reproduction environment. To facilitate future research on performance problems of CUDA programs, our current dataset is a set of performance problems with their environment configuration, input data, buggy version, fixed version, performance changes after fixing, and reproduction steps. Its goal is to 1) measure speedup after fixing CUDA performance problems and 2) assess tool effectiveness in detecting or fixing them.

4 Empirical Study Results

We present our analysis results of the four research questions.

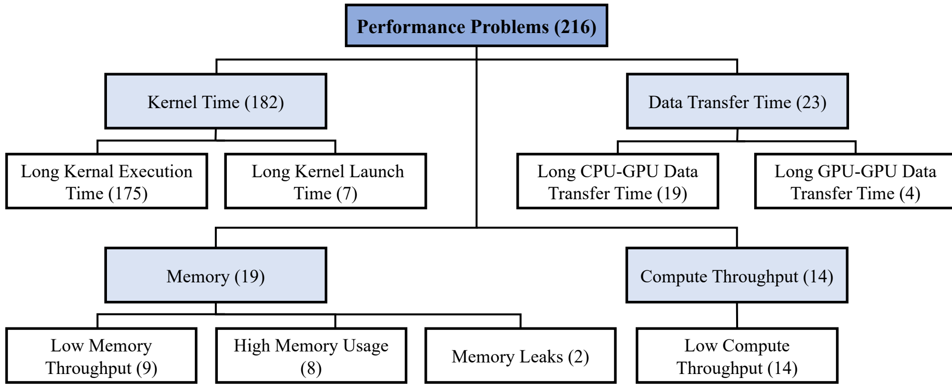

Fig. 3. Taxonomy of Performance Problem Symptoms

4.1 Symptom Analysis (RQ1)

As shown in Fig. 3, the taxonomy of performance problem symptoms is divided into four high-level categories, i.e., *kernel time*, *data transfer time*, *memory*, and *compute throughput*, with eight inner categories. One performance problem in a CUDA program may exhibit multiple symptoms.

Kernel Time. This category covers performance problems with a long kernel execution or launch time, which accounts for 182 (84.3%) of the performance problems. In detail, 175 (81.0%) of the performance problems manifest a long kernel execution time. Among them, 61 (28.2%) performance problems indicate that specific code blocks within the kernel have a long execution time. Although all code blocks contribute to the overall kernel execution time, the posts explicitly mention that certain code blocks incur exceptionally high time overhead, thereby exacerbating the kernel execution time. Besides, 7 (3.2%) of the performance problems exhibit a long kernel launch time prior to execution, which refers to the interval between the invocation of the CUDA kernel and its actual execution, a delay that is notably prolonged.

Data Transfer Time. This category includes performance problems with a long data transfer time, accounting for 23 (10.6%) of the performance problems. In particular, 19 (8.8%) of the performance problems exhibit a long host-device data transfer time, i.e., time for data transfer between CPU and GPU devices. 4 (1.9%) of the performance problems manifest a long device-device data transfer time, i.e., time for data transfer between multiple GPU devices. Due to the inherent nature of host-device heterogeneous computing, device memory needs to be allocated on the host prior to kernel execution, and the data is transferred from CPU to GPU or between GPU devices. This incurs significant time overhead in some cases.

Memory. This category includes performance problems demonstrating poor memory performance, accounting for 19 (8.8%) of the performance problems. Specifically, 9 (4.2%) of the performance problems manifest low memory throughput, indicating the inability to fully utilize the available memory resources on GPU devices. 8 (3.7%) of the performance problems exhibit a high memory usage that exceeds expectation, while 2 (0.9%) of the performance problems exhibit memory leaks.

Compute Throughput. This category consists of performance problems that exhibit low compute throughput. Compute throughput refers to the rate at which a GPU executes computations, typically measured in operations per second. A higher compute throughput reflects a better utilization of GPU compute resources. 14 (6.5%) of the performance problems manifest a low compute throughput, resulting in low compute resource utilization.

Summary. About 91.7% of the performance problems slow down the kernel execution and the data transfer. Such symptoms are specific to CUDA programs, and thus deserve investigation.

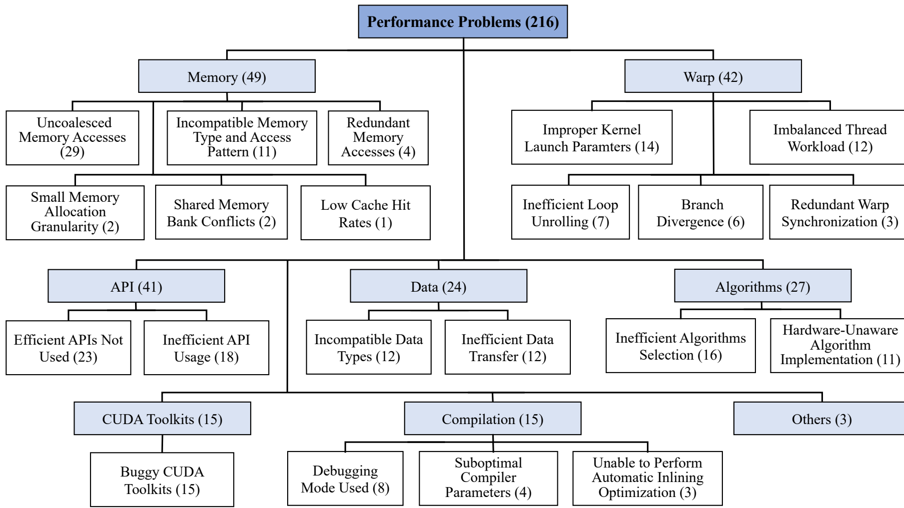

Fig. 4. Taxonomy of Performance Problem Root Causes

4.2 Root Cause Analysis (RQ2)

Fig. 4 shows the taxonomy of performance problem root causes. It is organized into seven high-level categories, i.e., *memory*, *warp*, *API*, *data*, *algorithms*, *CUDA toolkits*, and *compilation*, with 21 inner categories, accounting for 213 (98.6%) of the performance problems. The remaining 3 performance problems are caused by minor and uncommon issues, and thus are classified as *Others*.

Memory. This category includes performance problems caused by improper memory usage, which accounts for 49 (22.7%) of the performance problems. Specifically, *uncoalesced memory access* is the most common inner category along with other five inner categories.

- ***Uncoalesced Memory Accesses*** (29, 13.4%). Memory coalescing is a mechanism to consolidate memory accesses from multiple threads into a single operation when they access contiguous or adjacent memory addresses. It enhances memory bandwidth utilization and reduces memory access latency. Conversely, when threads access non-contiguous or non-adjacent memory addresses, memory accesses cannot be coalesced, leading to uncoalesced memory accesses. As illustrated in Fig. 5[1], the parameter `numStrides` is set to 256, and the warp exhibits a pattern where $thread_0$ accesses `d_mem[i]`, $thread_1$ accesses `d_mem[256 + i]`, and so forth. Consequently, memory accesses performed in the warp remain uncoalesced, thereby reducing memory efficiency.
- ***Incompatible Memory Type and Access Pattern*** (11, 5.1%). In CUDA, various types of memory (see Sec. 2) exhibit distinct access characteristics and performance implications. The efficiency of CUDA programs is significantly influenced by the memory type and access pattern, and the incompatibility between them can cause performance degradation. For example, the inefficient placement of small, frequently accessed data in local memory resulted in a performance bottleneck[2].
- ***Redundant Memory Accesses*** (4, 1.9%). This refers to redundant or unnecessary memory access operations performed by threads during execution. For example, repeated reading or writing of the same data caused excessive memory operations[3], which wastes memory resources and also leads to unnecessary latency.

[1]https://forums.developer.nvidia.com/t/154716
[2]https://forums.developer.nvidia.com/t/208418
[3]https://forums.developer.nvidia.com/t/176128

```
for (int i = 0; i < numStrides; i++) {
    const int idx = threadIdx.x * numStrides + i;
    float v = d_mem[idx];
}
```

Fig. 5. An Example of Uncoalesced Memory Accesses

- ***Small Memory Allocation Granularity*** (2, 0.9%). Memory allocation on GPU devices incurs substantial time overhead, making efficient allocation strategy critical for performance optimization. This category refers to scenarios in which memory blocks are allocated with insufficient size, resulting in frequent memory allocation operations with significant time overhead.
- ***Shared Memory Bank Conflicts*** (2, 0.9%). In CUDA, shared memory is organized into multiple independent memory banks, enabling parallel access to different banks. When multiple threads access distinct addresses within different banks, no conflict arises, allowing for efficient memory operations. However, if multiple threads attempt to access addresses within the same bank simultaneously, bank conflicts occur. In such cases, the hardware must serialize these memory accesses, increasing latency and reducing both parallelism and overall memory access efficiency.
- ***Low Cache Hit Rates*** (1, 0.5%). The program's memory access fails to effectively utilize the GPU cache system, leading to frequent cache misses and increased memory access latency.

Warp. This category covers performance problems caused by low warp occupancy, accounting for 42 (19.4%) of the performance problems. They exhibit a fundamental issue of failing to leverage GPU's high concurrency capability with five inner categories.

- ***Improper Kernel Launch Parameters*** (14, 6.5%). Suboptimal configurations of grid size and block size directly affect warp execution efficiency. A grid size that is too small prevents full utilization of the GPU's SMs, leading to insufficient warp occupancy. Similarly, a block size that is too small increases the number of parallel thread blocks but reduces the number of active warps per SM, leading to inefficient utilization of warp schedulers and lower overall throughput. For example, the grid size was set to 1, meaning that only one single SM was utilized, despite the experimental setup using an NVIDIA RTX 3090 [25] with 82 SMs[4]; due to the insufficient number of warps, the GPU failed to schedule other warps to perform computations while waiting for memory accesses. As a result, the execution units remained idle during memory operations, leading to significant performance degradation.
- ***Imbalanced Thread Workload*** (12, 5.6%). The workload distribution among threads within a warp is uneven, which can lead to performance inefficiencies. Since CUDA groups 32 threads into a warp, if some threads in the warp perform significantly more computations than others, the warp must wait until the most heavily loaded thread completes execution, resulting in a lower warp occupancy and inefficient use of computational resources. For example, the majority of computations may be assigned to one thread, while the other threads in the same warp perform significantly fewer tasks[5]. Consequently, the warp remained active but underutilized, as most threads completed their tasks early while waiting for the most computationally intensive thread to finish execution.
- ***Inefficient Loop Unrolling*** (7, 3.2%). Loop unrolling is an optimization technique to reduce loop overhead by manually or automatically expanding loop iterations, thereby increasing instruction-level parallelism and mitigating branch divergence. However, excessive loop unrolling may lead to increased memory usage of the kernel, causing the GPU's SM resources to become tight and limiting the number of warps that can be executed in parallel.

[4]https://stackoverflow.com/questions/76745515
[5]https://stackoverflow.com/questions/76745515

```
for(int stride = 1; stride < blockDim.x; stride*=2){
    if((tid%(2*stride)) == 0){
        data[tid] += data[tid+stride];
    }
    __syncthreads();
}
```

Fig. 6. An Example of Branch Divergence

- ***Branch Divergence*** (6, 2.8%). Threads within the same warp execute conditional statements and select different paths. Unlike CPUs, GPU devices lack advanced branch prediction capability and can only stall threads that are not in the active path. As a result, the hardware must execute each path separately, with threads not on the active path remaining stalled. Fig. 6 shows an example of branch divergence[6]. When $stride = 1$, only even-numbered threads within the same warp enter the branch, while odd-numbered threads are stalled. When $stride = 2$, threads whose identifier $tid\%4 == 0$ enter the branch, while the other threads are stalled.
- ***Redundant Warp Synchronization*** (3, 1.4%). This category covers the performance problems caused by the unnecessary synchronization of threads within a warp, particularly in cases where data races do not occur. Notice that the original posters who provided the fixed code showed that the warp synchronization was redundant, confirming there was no data race. Warp synchronization enforces the execution of all threads within a warp in a predefined order to prevent issues such as data inconsistency. However, when applied unnecessarily, redundant synchronization introduces avoidable execution overhead, leading to performance degradation.

API. This category covers performance problems caused by improper use of library APIs with two inner categories, accounting for 41 (19.0%) of the performance problems. NVIDIA provides highly optimized libraries for common computational tasks, such as cuBLAS [33], an efficient linear algebra library, and Thrust [36], which encapsulates common parallel algorithms. These libraries are carefully designed to leverage GPU hardware features and optimal memory access patterns. However, developers sometimes fail to use these optimized APIs properly, which leads to suboptimal performance. Specifically, 23 (10.6%) of the performance problems are categorized as ***Efficient APIs Not Used***, where developers manually implement computational operations instead of using the corresponding efficient APIs. For example, matrix multiplication was implemented manually rather than utilizing the cuBLAS library[7]. Besides, 18 (8.3%) of the performance problems involve ***Inefficient API Usage***. Even when developers are aware of available APIs, they may lack a deep understanding of performance characteristics of these APIs. As a result, improper API invocation, misconfigured parameters, or selecting an API unsuitable for the given workload can introduce significant performance bottlenecks.

Data. This category covers root causes of performance problems related to data processing, accounting for 24 (11.1%) of the performance problems. It can be classified into two inner categories. Specifically, 12 (5.6%) of the performance problems is caused by ***Incompatible Data Types***. GPUs can exhibit varying capabilities in handling different data types. For example, NVIDIA RTX 3090 has significantly lower computational throughput for double-precision floating-point arithmetic compared to single-precision operations. These performance problems use data types that the GPU handles less efficiently. Moreover, 12 (5.6%) of the performance problems involve ***Inefficient Data Transfer*** patterns. When data in unified memory is not proactively moved to the GPU, frequent

[6]https://stackoverflow.com/questions/68959852
[7]https://forums.developer.nvidia.com/t/263823

page migrations occur with increased latency. Frequent small data transfers result in excessive low-volume memory transactions, reducing overall bandwidth efficiency. Computations and memory transfers are serialized rather than performed concurrently, resulting in non-overlapping kernel executions and data transfers, leading to suboptimal GPU resource utilization. Data exchange between GPUs follows a suboptimal path, and thus leads to inefficient multi-GPU system topology, increasing communication overhead and reducing performance.

Algorithms. This category covers performance problems caused by inefficient algorithms, accounting for 27 (12.5%) of the performance problems with two inner categories. Specifically, 16 (7.4%) of the performance problems suffer ***Inefficient Algorithms Selection***, resulting in a high time complexity for the CUDA program. For example, an inefficient reduction algorithm caused suboptimal performance[8]. Besides, 11 (5.1%) of the performance problems are caused by ***Hardware-Unaware Algorithm Implementation***, where the algorithm fails to adapt to the highly parallel GPU programming model. For example, an inefficient serial algorithm was used for a first-order recursive problem, ignoring the execution model and resource limitations of GPU and thus leading to suboptimal performance[9].

CUDA Toolkits. This category covers performance problems caused by bugs inherent in CUDA toolkits, accounting for 15 (6.9%) of the performance problems. Given the complexity of CUDA toolkit development, the presence of certain bugs is inevitable, leading to performance degradation even in correctly implemented CUDA programs. Some performance problems are caused by CUDA bugs related to specific CUDA versions and CUDA libraries.

Compilation. This category covers performance problems caused by compilation issues, accounting for 15 (6.9%) of the performance problems with three inner categories. Specifically, 8 (3.7%) of the performance problems use the debug mode. When debugging features are incorrectly enabled during compilation, the CUDA program is compiled into a version that includes additional debug information, e.g., symbol information, execution checks, and error-checking code. These additional overheads cause a noticeable reduction in the program's execution speed. 3 (1.4%) of the performance problems lack compilation hints, preventing the compiler from performing automatic inlining optimizations; and 4 (1.9%) of the performance problems have suboptimal compiler parameters.

Others. This category includes root causes that are less frequent, accounting for 3 (1.4%) of the performance problems. For example, a performance problem is not due to the problem with the program itself, but rather caused by GPU hardware issues. A performance problem occurs because GPU slot limitations prevent the theoretical bandwidth from being achieved, while another is due to inadequate cooling, leading to reduced power and thus impacting performance.

Summary. The root causes of performance problems in CUDA programs are quite diverse, which makes their localization challenging. Around 61.1% of the performance problems are introduced by issues related to memory usage, warp occupancy, and APIs, which involve key characteristics about CUDA programming model.

4.3 Speedup Analysis After Fixing (RQ3)

The first six columns of Table 1 report the number of performance problems across root causes and symptoms. The numbers before the parentheses indicate the number of successfully reproduced performance problems in our dataset, while the numbers in parentheses represent the total number of performance problems. Our dataset cover all root causes except for three of the 21 inner categories. The seventh column of Table 1 shows the speedup range after fixing the 69 performance problems in our dataset. Overall, a speedup of up to 2.0×, 3.0×, and 5.7× is achieved in 50%, 40%, and 30% of the

[8]https://stackoverflow.com/questions/69313189
[9]https://forums.developer.nvidia.com/t/248186

Table 1. Reproduced Performance Problems across Root Causes and Symptoms and Evaluation Results

Root Cause	Symptom				Total	Speedup	Ncu		Nsys	Doc.
	C. T. [a]	Mem. [b]	K. T. [c]	D. T. [d]			App.	Iden.	App.	App.
Memory	**0 (2)**	**3 (5)**	**20 (46)**	**1 (1)**	**23 (49)**	**[1.04, 72.88]**	**16**	**14**	**2**	**21**
Uncoalesced Memory Accesses	0 (1)	0 (1)	14 (29)	0 (0)	14 (29)	[1.05, 46.57]	14	13	0	14
Incompatible Memory Type and Access Pattern	0 (1)	0 (1)	3 (11)	0 (0)	3 (11)	[1.05, 72.88]	0	0	0	3
Redundant Memory Accesses	0 (0)	0 (0)	2 (3)	1 (1)	3 (4)	[1.04, 1.06]	1	0	0	1
Small Memory Allocation Granularity	0 (0)	3 (3)	0 (0)	0 (0)	2 (2)	[3.12, 5.17]	0	0	2	2
Shared Memory Bank Conflicts	0 (0)	0 (0)	1 (2)	0 (0)	1 (2)	[1.32, 1.56]	1	1	0	1
Low Cache Hit Rate	0 (0)	0 (0)	0 (1)	0 (0)	0 (1)	–	0	0	0	0
Warp	**2 (6)**	**2 (5)**	**17 (38)**	**0 (1)**	**17 (42)**	**[1.00, 2222.41]**	**16**	**13**	**6**	**16**
Improper Kernel Launch Parameters	1 (5)	0 (0)	4 (13)	0 (0)	4 (14)	[1.25, 32.76]	3	3	0	3
Imbalanced Thread Workload	1 (1)	2 (3)	6 (10)	0 (1)	6 (12)	[1.12, 2222.41]	6	6	6	6
Inefficient Loop Unrolling	0 (0)	0 (2)	4 (6)	0 (0)	4 (7)	[1.22, 2.44]	4	1	0	4
Branch Divergence	0 (0)	0 (0)	2 (6)	0 (0)	2 (6)	[1.98, 9.57]	2	2	0	2
Redundant Warp Synchronization	0 (0)	0 (0)	1 (3)	0 (0)	1 (3)	[1.00, 1.01]	1	1	0	1
API	**0 (2)**	**0 (2)**	**9 (31)**	**1 (9)**	**10 (41)**	**[1.01, 6214.44]**	**0**	**0**	**0**	**5**
Efficient APIs Not Used	0 (2)	0 (2)	5 (20)	0 (2)	5 (23)	[1.32, 2959.97]	0	0	0	5
Inefficient API Usage	0 (0)	0 (0)	4 (11)	1 (7)	5 (18)	[1.01, 6214.44]	0	0	0	0
Data	**1 (2)**	**0 (1)**	**7 (15)**	**1 (8)**	**8 (24)**	**[1.07, 32.78]**	**5**	**4**	**1**	**6**
Incompatible Data Types	1 (1)	0 (1)	7 (11)	0 (0)	7 (12)	[1.07, 32.78]	5	4	0	5
Inefficient Data Transfer	0 (1)	0 (0)	0 (4)	1 (8)	1 (12)	[1.09, 1.13]	0	0	1	1
Algorithms	**0 (1)**	**0 (2)**	**8 (26)**	**0 (0)**	**8 (27)**	**[1.00, 988.71]**	**0**	**0**	**0**	**2**
Inefficient Algorithms Selection	0 (1)	0 (2)	4 (15)	0 (0)	4 (16)	[1.00, 988.71]	0	0	0	0
Hardware-Unaware Algorithm Implementation	0 (0)	0 (0)	4 (11)	0 (0)	4 (11)	[1.38, 16.41]	0	0	0	2
CUDA Toolkits	**0 (1)**	**0 (2)**	**0 (12)**	**0 (1)**	**0 (15)**	**–**	**0**	**0**	**0**	**0**
Compilation	**0 (0)**	**1 (2)**	**2 (13)**	**0 (1)**	**3 (15)**	**[3.43, 173.64]**	**0**	**0**	**0**	**0**
Debugging Mode Used	0 (0)	1 (2)	2 (7)	0 (0)	3 (8)	[3.43, 173.64]	0	0	0	0
Suboptimal Compiler Parameters	0 (0)	0 (0)	0 (3)	0 (1)	0 (4)	–	0	0	0	0
Unable to Perform Automatic Inlining Optimization	0 (0)	0 (0)	0 (3)	0 (0)	0 (3)	–	0	0	0	0
Others	**0 (0)**	**0 (0)**	**0 (1)**	**0 (2)**	**0 (3)**	**–**	**0**	**0**	**0**	**0**
Total	**3 (14)**	**6 (19)**	**63 (182)**	**3 (23)**	**69 (216)**	**[1.00, 6214.44]**	**37**	**31**	**9**	**50**

a : Compute Throughput *b* : Memory *c* : Kernel Time *d* : Data Transfer Time

performance problems after fixing, respectively. Notice that posts involving multiple performance problems are split into separate performance problems, each performance problem in our dataset has one root cause, and its speedup is measured by fixing one root cause in isolation.

Fixing Uncoalesced Memory Accesses (14, 20.3%). The fix is to adjust the memory access pattern such that memory addresses accessed by the same warp are contiguous, enabling coalesced memory accesses. For example, the speedup after fixing one performance problem[10] reached 34.08.

Fixing Incompatible Memory Type and Access Pattern (3, 4.3%). The fix involves changing the type of memory used. In two performance problems[11,12], the memory was changed from shared memory to registers, resulting in small speedups of 1.03 and 1.17, respectively. However, in another one[13], the memory was changed from local memory to registers, achieving a large speedup of 70.51.

Fixing Redundant Memory Accesses (3, 4.3%). The fix is to find redundant memory read and write operations in the code and remove them. The performance loss caused by redundant memory access is relatively small, and thus the speedup after fixing is also low.

Fixing Small Memory Allocation Granularity (2, 2.9%). The fix is to merge multiple memory allocations into a single one. One performance problem[14] caused an "out of memory" error even at

[10]https://stackoverflow.com/questions/76021386
[11]https://stackoverflow.com/questions/72769216
[12]https://forums.developer.nvidia.com/t/209144
[13]https://forums.developer.nvidia.com/t/208418
[14]https://stackoverflow.com/questions/60269623

the smallest data size. The program could only run normally after memory allocations were merged, achieving a speedup of 3.12.

Fixing Shared Memory Bank Conflicts (1, 1.4%). The fix is to change the shared memory access pattern to avoid multiple threads accessing different data in the same bank. The fix applied to one performance problem[15] changed the distribution of data across the banks such that threads accessed different banks,

Fixing Improper Kernel Launch Parameters (4, 5.8%). To fully utilize GPU devices, it is important to avoid excessively small grid sizes and block sizes (e.g., 1). At the same time, an overly large block size may reduce parallelism of blocks on the SM. Adjusting to an appropriate block size can increase warp occupancy. Since the impact of launch parameters on the device's utilization varies across performance problems, the speedup between different performance problems can differ significantly. For example, for the performance problem[16] discussed in Sec. 4.2, after fixing the grid size to a reasonable value, the speedup reached as high as 29.62.

Fixing Imbalanced Thread Workload (6, 8.7%). A common fix pattern is to balance the thread workload by ensuring that all threads within a warp have similar execution times and resource demands. The average speedup for this type of performance problems is 1.62, and the speedup does not vary significantly with data size. Due to the extremely imbalanced workload in one performance problem[17], the speedup after fixing reached 1906.37.

Fixing Inefficient Loop Unrolling (3, 4.3%). The common fix is to unroll loops with high time overhead. For example, adding `#pragma unroll 32` to a for loop to unroll it resulted in a speedup of 1.23[18]. Differently, one performance problem was caused by excessive unrolling[19]. Loop unrolling can provide better performance for small data sizes, but when the data size exceeds 2^{26} bytes, it degrades performance. Thus, not unrolling the loop achieved a speedup of 1.22.

Fixing Branch Divergence (2, 2.9%). A common fix is to adjust the conditional statements to ensure all threads in a warp follow the same execution path. The speedups ranged from 1.98 to 9.57.

Fixing Redundant Warp Synchronization (1, 1.4%). The fix is to remove the redundant `__syncthreads()` statement. The performance loss caused by redundant synchronization is relatively small, and therefore the speedup after the fix is low, with an observed speedup of only 1.01.

Fixing Efficient APIs Not Used (5, 7.2%). Developers need to fully understand the APIs provided by commonly used CUDA high-performance libraries rather than implementing their own simple versions. The simple implementation in one performance problem[20] showed a significant performance gap compared to the Thrust implementation. After switching to Thrust, the speedup reached 190.68 at data scale of 2^{20} bytes and 2959.97 at data scale of 2^{29} bytes.

Fixing Inefficient API Usage (5, 7.2%). Developers should understand how to use library APIs correctly, eliminating incorrect or improper usage scenarios. There was a mix usage of Managed Memory API and Traditional Memory API[21], which were incompatible and introduced latency. After switching to consistent APIs, the speedup reached 31.65 at data scale of 2^{20} bytes and 6214.44 at data scale of 2^{29} bytes.

Fixing Incompatible Data Types (7, 10.1%). GPUs have different compute throughput for various data types. In scenarios where high precision is not required, replacing data types with higher compute throughput can lead to performance improvements. For example, GPU's compute throughput

[15]https://stackoverflow.com/questions/75414006
[16]https://stackoverflow.com/questions/76745515
[17]https://stackoverflow.com/questions/76745515
[18]https://forums.developer.nvidia.com/t/111215
[19]https://stackoverflow.com/questions/62445861
[20]https://stackoverflow.com/questions/76247513
[21]https://stackoverflow.com/questions/75831866

for floats is higher than for doubles, and thus using float instead of double brought a performance speedup of 31.34 when double precision was not necessary[22].

Fixing Inefficient Data Transfer (1, 1.4%). Kernel execution and data transfer were not overlapped in a performance problem[23]. As kernel execution and data transfer used different hardware parts, they could be performed in parallel. The non-overlapping execution made data transfer and computation tasks execute serially, slowing down the overall progress of the program. overlapping kernel execution and data transfer led to a speedup of up to 1.13.

Fixing Inefficient Algorithms Selection (4, 5.8%). The fix is to directly replace or optimize the inefficient algorithm. The speedup varies based on the difference between the original algorithm and the optimized algorithm. For example, after fixing the reduction algorithm[24], the speedup reached 107.91 at data scale of 2^{20} and 988.71 at data scale of 2^{29}.

Fixing Hardware-Unaware Algorithm Implementation (4, 5.8%). Directly porting a serial algorithm to GPU execution often fails to fully utilize the GPU device. The GPU characteristics and the CUDA programming model should be considered when designing algorithms. For example, replacing the matrix operations with a parallel-friendly algorithm resulted in a speedup of 1.38[25].

Fixing Debugging Mode Used (3, 4.3%). Although debugging mode is commonly used to check other CUDA bugs, it is essential to ensure that the program is not compiled with debugging mode for production. The speedup for three performance problems were 3.72, 173.64, and 3.43, respectively.

Summary. Fixing these performance problems results in varying levels of speedup, achieving an average speedup of 2.14 (after excluding extreme cases). Therefore, the benefit of fixing performance problems can be huge, which motivates the necessity of performance optimization.

4.4 Approach Assessment (RQ4)

We selected two typical performance analysis methods, which can be used by CUDA developers to optimize the performance, and documentation related to CUDA performance optimization that developers can use to analyze the performance of CUDA programs.

- Nsight Compute [34] is a kernel-level performance analysis tool provided by NVIDIA, specifically designed for optimizing kernel performance in CUDA programs. It provides rich metrics and detailed statistics, helping developers identify bottlenecks during GPU kernel execution process, such as computational throughput, memory access patterns, and instruction execution.
- Nsight Systems [35] is a system-level performance analysis tool, aimed at helping developers gain a comprehensive understanding of the interactions between the application, CPU, GPU, and other system resources. Unlike Nsight Compute, which focuses on kernel performance, Nsight Systems provides multi-dimensional timeline analysis that displays thread scheduling, memory access, I/O operations, etc. It helps developers identify problems such as multi-threaded parallelism, system bottlenecks, and data transfer latency, allowing for optimizations across the entire application.
- Documentation often provides suggestions for performance optimization. Thus, we selected two official NVIDIA documents related to CUDA performance optimization, the CUDA C++ Programming Guide [28], and the CUDA C++ Best Practices Guide [27], as well as a well-known book about CUDA programming that provides developer's guide to parallel computing with GPUs [8].

For these three techniques, we evaluated whether a technique was applicable to a performance problem (or whether the performance problem fell within the capability scope of the technique).

[22]https://forums.developer.nvidia.com/t/111963
[23]https://stackoverflow.com/questions/73155788
[24]https://stackoverflow.com/questions/69313189
[25]https://forums.developer.nvidia.com/t/263823

Moreover, for Nsight Compute, we evaluated whether it could identify performance problems and suggest fixes. Our assessment results on our dataset are presented in the last four columns of Table 1.

According to the eighth and ninth columns of Table 1, Nsight Compute is applicable to 37 (53.6%) performance problems; and it suggests reasonable fixes for 31 (44.9%) performance problems. For example, a CUDA program had two performance problems[26], branch divergence and uncoalesced memory accesses. Nsight Compute provided indicators on the code lines causing performance problems and offered fix suggestions, *"this line is responsible for a high number of warp stalls. See markers on SASS lines for details"* and *"96.88% of this line's global accesses are excessive"*. Nsight Compute is not applicable to other performance problems for two main reasons. First, Nsight Compute can only analyze performance by capturing performance metrics, but cannot analyze code semantics. However, code semantics understanding is often required to identify performance problems caused by APIs and algorithms. Second, Nsight Compute works at the kernel level [31], while performance problems such as "small memory allocation granularity" and "inefficient data transfer" are more system-level. These results suggest that Nsight Compute works well for analyzing performance problems in the warp category and most of the memory category, but has very limited capability in analyzing problems in the API and algorithms category.

According to the tenth column of Table 1, Nsight Systems is only applicable to 9 (13.0%) performance problems. For example, in the Timeline View of Nsight Systems, a massive number of memory allocations were observed for one performance problem[27], from which the performance problem of small memory allocation granularity was inferred. Nsight Systems is not applicable to other performance problems for two main reasons. The first reason is the same to the first one for Nsight Compute. The second is that Nsight Systems is a system-level performance analysis tool [32], and performance problems that require in-depth kernel-level analysis fall outside its capability. These results suggest that Nsight Systems complements Nsight Compute well when analyzing performance problems in the warp, memory and data categories, but its capability to analyze problems in other categories is also very limited.

According to the last column of Table 1, documentation is applicable to 50 (72.5%) performance problems. Notice that as long as the documents mention hints to certain performance problems, we consider them as applicable. In that sense, the documents provide more coverage on performance problems related to warp, memory, and data categories, while there is less discussion on performance problems in the API, algorithms, and compilation categories.

Notice that we also evaluated other performance analysis tools, such as Linaro Forge [20], TAU [37], Score-P [41], and HPCToolkit [44]. However, these tools offer very limited support for CUDA, and can only provide some high-level statistical metrics, making them not applicable to CUDA performance problems.

Summary. Existing performance analysis tools Nsight Compute and Nsight Systems are respectively applicable to 53.6% and 13.0% of the performance problems, with an especially limited capability in identifying API- and algorithms-related performance problems. In addition, although documentation provides hints for a high coverage (i.e., 72.5%) of performance problems, automated tools are still needed to ease the problem identification.

5 Implications and Threats

We provide the implications for developers and researchers, and discuss the threats to validity.

[26] https://stackoverflow.com/questions/61873539

[27] https://stackoverflow.com/questions/60269623

5.1 Implications

For Developers. Our study uncovers common symptoms of performance problems, which CUDA developers can observe when testing and running CUDA programs. Notably, diagnosing performance problems related to memory, warp and data requires fine-grained performance metrics for accurate localization. This highlights the need for developers to focus on more diverse performance metrics when identifying and localizing performance problems. We also summarize common root causes of performance problems, which can serve as a reference for CUDA developers when diagnosing, debugging, or fixing performance problems. Root causes related to API, algorithms, CUDA Toolkits, and compilation are rarely covered in NVIDIA documentation [8, 27, 28] and tools [34, 35]. Therefore, developers need to learn how to diagnose and resolve these performance problems independently. For example, developers should be familiar with common high-performance CUDA libraries and their APIs to reduce performance problems in the API category by using optimized and well-supported libraries for common functionalities. While constructing our dataset, we found that some performance problems caused by loop unrolling[28] and algorithms[29] only manifest at larger data scales. Therefore, when testing program performance or reproducing performance problems, developers should consider multiple data scales.

For Researchers. Our findings suggest three directions for future research. The first is to improve performance analysis methods. Although current performance analysis tools, such as Nsight Compute and Nsight Systems, provide useful insights into CUDA program performance problems, these tools have certain limitations when analyzing more complex problems. For example, current tools primarily rely on collecting performance metrics to analyze program execution, but they lack sufficient depth in analyzing performance problems related to categories such as API and algorithms. Future research could explore how to enhance the semantic analysis capabilities of these tools, enabling them to better understand the specific implementations within the code, analyze potential performance bottlenecks, and offer more targeted optimization suggestions. In addition, incorporating developer studies can help to better understand how practitioners identify and diagnose performance problems in practice, thereby informing the design of more effective analysis techniques.

The second is intelligent API recommendation. According to our empirical results, many CUDA programs fail to efficiently utilize existing CUDA APIs. Particularly, when handling common computational tasks, developers often prefer writing custom implementations rather than utilizing optimized, high-performance CUDA libraries. These manual implementations typically do not achieve the same performance levels as existing APIs, leading to performance bottlenecks. Therefore, there is an urgent need to develop intelligent API recommendation techniques to recommend the best CUDA APIs based on the specific requirements of the code. By automating the selection process, developers can choose the most suitable APIs, thereby improving the overall performance.

The third is to automate performance problem fixing. Current performance analysis tools assist developers in identifying performance problems and suggesting fixes, but they do not automate the repair process. Developers still need to manually adjust their code based on the suggestions. Future research should focus on automating the performance problem fixing process, particularly by integrating problem detection with automated implementation of suggested fixes to minimize developer intervention and manual debugging. Possible techniques include pattern-based transformation, where known performance problems are matched to predefined optimization templates, and machine learning-driven approaches that leverage trained models to suggest code modifications. Compiler-assisted optimization could also be enhanced to detect inefficiencies and apply fixes

[28]https://stackoverflow.com/questions/62445861
[29]https://stackoverflow.com/questions/59895961

automatically. However, challenges such as ensuring correctness and adapting to diverse CUDA workloads remain significant barriers to fully automated performance optimization.

5.2 Threats

First, our assessment across the four RQs may have the threat of subjective assessment. The data collection process involved filtering. While this process ensures high-quality data, it is not completely exhaustive, and some performance problems may have been missed. However, the adopted keywords set achieved a recall of 100% on the 300 sampled posts. For RQ1 and RQ2, our study involved manual analysis of a large number of posts, taking six person-months to complete. However, we follow the open coding procedure, and our description of performance problems are mainly based on the discussion in their original posts. For RQ3 and RQ4, they are answered by experimental results. Therefore, we believe our assessment is fairly objective.

Second, while GitHub could have been a potential data source, several factors led us to exclude it from this study. The quality and consistency of the data on GitHub can be inconsistent. Some submitted code and problem descriptions may lack detailed context or complete reproduction steps, which increases the difficulty of collecting data from GitHub and ensuring its quality. Therefore, we chose StackOverflow and NVIDIA forums, as they provide relatively high-quality and more targeted performance problem data, particularly in the context of CUDA and GPU programming.

Third, while our dataset size is relatively large, it may not cover all the CUDA performance problems encountered in real-world applications, and thus the distribution of performance problems may be influenced by the characteristics of those reported in public discussions. Specifically, some non-reproducible problems (due to hardware issues, insufficient context or reproducible code) were excluded, which may affected RQ3 and RQ4. However, we believe the impact is limited. This is because our dataset has a high coverage of symptom and root cause categories, and is derived from real-world developer discussions across two independent sources, which helps capture common patterns of CUDA performance problems. To further validate and strengthen the generalizability of our findings, future research should be conducted on larger and more diverse datasets, especially across different domains and application scenarios.

6 Related Work

We review the closely related work in three aspects, i.e., high-performance computing performance problems, CUDA program performance analysis, and CUDA program performance optimization.

High-Performance Computing Performance Problems. We first discuss studies on performance problems in general HPC programs and GPU programs. Azad et al. [2] collected 1,729 HPC performance commits from 23 real-world HPC projects. They analyzed root causes, fix patterns, fix effort, and the required HPC skills to fix performance problems. However, they did not differentiate GPU-related performance problems from CPU-related ones, resulting in a taxonomy that lacks some GPU-specific root causes. Yang et al. [48] examined 10 open-source GPU projects on GitHub and identified root causes, fix patterns, and energy effects of performance problems. However, they did not isolate CUDA-specific performance problems (on NVIDIA GPUs) with OpenCL-specific ones (on AMD GPUs); and it targeted old versions of CUDA.

Next, we summarize studies on CUDA performance problems. Landaverde et al. [18] found that the performance overhead of Unified Memory Access (UMA) introduced in CUDA 6+ varied significantly depending on the memory access patterns. Czarnul [9] analyzed the impact of CUDA streaming parameters and memory access patterns on host-device data transfer time. Cao et al. [4] discovered that a mismatch between the CUDA and TensorFlow versions could lead to performance problems. Yoshida et al. [50] analyzed how CUDA version changes affected GPU performance. They found that in some cases, newer CUDA versions exhibited performance degradation compared to

older ones due to excessive loop unrolling, inefficient instruction scheduling, and compiler effects. These studies focus on specific types of CUDA performance problems, whereas our work aims to systematically characterize CUDA performance problems.

Finally, we summarize existing datasets for CUDA programs. Several benchmarks [1, 5, 43] have been constructed. Rodinia [5] is a benchmark suite for heterogeneous computing, providing OpenMP, OpenCL, and CUDA implementations. It includes benchmarks that exhibit various types of parallelism and data access patterns across diverse application domains. Parboil [43] provided a set of scientific computing applications to evaluate the performance of heterogeneous computing platforms. Araujo et al. [1] developed a CUDA version of the NAS Parallel Benchmarks [3], a standard suite for assessing parallel hardware and software. However, they are designed for performance evaluation rather than for reproducing or analyzing performance problems. Yi et al. [49] constructed the CUDAMicroBench benchmark, which contained 14 microbenchmarks collected from previous works [26, 46] to analyze CUDA performance problems and fix patterns. However, it only covers a subset of performance problems related to warp and memory.

CUDA Program Performance Analysis. Knobloch et al. [16] provided an overview of commercial performance analysis tools for heterogeneous HPC applications [20, 34, 35, 37, 41, 44]. Among them, Nsight Compute [34] and Nsight Systems [35], both developed by NVIDIA, offer more detailed performance analysis (e.g., per-thread and per-warp analysis, memory access efficiency, and asynchronous data transfer monitoring), making them the selected baselines in Sec. 4.4. However, these tools primarily provide detailed performance metrics but rarely localize performance bottlenecks. Developers must interpret the data to identify root causes and determine appropriate fixes. Muller et al. [23] proposed RACUDA, an end-to-end resource analysis tool for CUDA kernels. Unlike previous tools, RACUDA can directly detect and quantify the impact of three types of performance problems, i.e., divergent warps, uncoalesced memory accesses, and shared memory bank conflicts. However, it supports only a simple subset of CUDA (miniCUDA) and thus cannot be evaluated on our dataset, which is constructed from real-world examples with advanced CUDA features.

CUDA Program Performance Optimization. CUDA applications can be optimized at two levels, i.e., infrastructure-level and program-level [14]. Infrastructure-level optimizations enhance performance without modifying CUDA programs by optimizing the infrastructure of CUDA applications [15, 19, 21, 53]. Examples include flexible kernel fusion [53], offloading parts of GEMM operations from CUDA cores to Tensor cores [15], optimizing data transfer between kernels by leveraging the L2 cache of GPUs [21], and reducing collective communication overhead across multiple GPU nodes [19]. Fixing CUDA program performance problems requires modifying CUDA programs, making it complementary to infrastructure-level optimizations. Program-level optimizations improve performance by modifying CUDA programs to better use heterogeneous computing resources [6, 10, 40]. Examples include efficient usage of CUDA Unified Memory [6], loop unrolling [10], and overlapping data transfer with GPU kernel execution [40]. We did not select program-level performance optimization techniques as baselines in Sec. 4.4 for two reasons. First, they can only fix programs with known performance problems and cannot detect performance problems. Second, each technique targets specific root causes, but our dataset covers a wide range.

7 Conclusions

This paper has comprehensively investigated CUDA program performance problems. We have collected performance problems, identified their symptoms and root causes, measured the speedup of fixing performance problems, and assessed the capability of existing performance analysis tools. Our work offers practical insights for CUDA developers to enhance CUDA program performance, and pinpoints directions for future research.

8 Data Availability

The data supporting the findings of this study is available at https://sites.google.com/view/cudaperf.

Acknowledge

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] Gabriell Araujo, Dalvan Griebler, Dinei A Rockenbach, Marco Danelutto, and Luiz G Fernandes. 2023. NAS Parallel Benchmarks with CUDA and beyond. *Software: Practice and Experience* 53, 1 (2023), 53–80. doi:10.1002/spe.3056

[2] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. 2023. An empirical study of high performance computing (HPC) performance bugs. In *Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories*. 194–206. doi:10.1109/msr59073.2023.00037

[3] David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Ramesh A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Robert S. Schreiber, Horst D. Simon, Venkat Venkatakrishnan, and S. Weeratunga. 1991. The NAS Parallel Benchmarks. *The International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73. doi:10.2172/983318

[4] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369. doi:10.1145/3540250.3549123

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization*. Ieee, 44–54. doi:10.1109/iiswc.2009.5306797

[6] Hyeonseong Choi and Jaehwan Lee. 2021. Efficient use of GPU memory for large-scale deep learning model training. *Applied Sciences* 11, 21 (2021), 10377. doi:10.3390/app112110377

[7] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46. doi:10.1177/001316446002000104

[8] Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[9] Paweł Czarnul. 2020. Investigation of parallel data processing using hybrid high performance CPU+ GPU systems and CUDA streams. *Computing and informatics* 39, 3 (2020), 510–536. doi:10.31577/CAI_2020_3_510

[10] Jacek Gambrych. 2021. Influence of optimization techniques on software performance for subsequent generations of CUDA architecture. In *Proceedings of the 2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking*. 1002–1009. doi:10.1109/ispa-bdcloud-socialcom-sustaincom52081.2021.00140

[11] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. 134–144. doi:10.1109/ispass.2011.5762730

[12] Stijn Heldens, Pieter Hijma, Ben Van Werkhoven, Jason Maassen, Adam SZ Belloum, and Rob V Van Nieuwpoort. 2020. The landscape of exascale research: A data-driven literature analysis. *Comput. Surveys* 53, 2 (2020), 1–43. doi:10.1145/3372390

[13] John L Hennessy and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier. doi:10.1016/0026-2692(93)90111-q

[14] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben Van Werkhoven, and Henri E Bal. 2023. Optimization techniques for GPU programming. *Comput. Surveys* 55, 11 (2023), 1–81. doi:10.1145/3570638

[15] Khoa Ho, Hui Zhao, Adwait Jog, and Saraju Mohanty. 2022. Improving gpu throughput through parallel execution using tensor cores and cuda cores. In *2022 IEEE Computer Society Annual Symposium on VLSI*. 223–228. doi:10.1109/isvlsi54635.2022.00051

[16] Michael Knobloch and Bernd Mohr. 2020. Tools for gpu computing–debugging and performance analysis of heterogenous hpc applications. *Supercomputing Frontiers and Innovations* 7, 1 (2020), 91–111. doi:10.14529/jsfi200105

[17] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (1974), 83–93. doi:10.1145/360827.360844

[18] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. 2014. An investigation of unified memory access performance in cuda. In *Proceedings of the 2014 IEEE High Performance Extreme Computing Conference*. 1–6. doi:10.1109/hpec.2014.7040988

[19] Baojia Li, Xiaoliang Wang, Jingzhu Wang, Yifan Liu, Yuanyuan Gong, Hao Lu, Weizhen Dang, Weifeng Zhang, Xiaojie Huang, Mingzhuo Chen, et al. 2024. TCCL: Co-optimizing Collective Communication and Traffic Routing

for GPU-centric Clusters. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*. 48–53. doi:10.1145/3672198.3673799

[20] Linaro. 2025. Linaro Forge. https://www.linaroforge.com/. Accessed: February 25, 2025.

[21] Arian Maghazeh, Sudipta Chattopadhyay, Petru Eles, and Zebo Peng. 2019. Cache-Aware Kernel Tiling: An Approach for System-Level Performance Optimization of GPU-Based Applications. In *2019 Design, Automation and Test in Europe Conference and Exhibition*. 570–575. doi:10.23919/date.2019.8714861

[22] Michael McCool, James Reinders, and Arch D Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier.

[23] Stefan K Muller and Jan Hoffmann. 2021. Modeling and analyzing evaluation cost of CUDA kernels. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–31. doi:10.1145/3639403

[24] Felix Nahrstedt, Mehdi Karmouche, Karolina Bargieł, Pouyeh Banijamali, Apoorva Nalini Pradeep Kumar, and Ivano Malavolta. 2024. An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 58–68. doi:10.1145/3661167.3661203

[25] NVIDIA. 2020. GeForce RTX 3090. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/. Accessed: February 25, 2025.

[26] NVIDIA. 2023. CUDA Samples. https://docs.nvidia.com/cuda/cuda-samples/index.html. Accessed: February 25, 2025.

[27] NVIDIA. 2025. CUDA C Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. Accessed: February 25, 2025.

[28] NVIDIA. 2025. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide. Accessed: February 25, 2025.

[29] NVIDIA. 2025. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed: March 8, 2025.

[30] NVIDIA. 2025. CUDA Zone. https://developer.nvidia.com/cuda-zone. Accessed: March 8, 2025.

[31] NVIDIA. 2025. Nsight Compute Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide. Accessed: February 25, 2025.

[32] NVIDIA. 2025. Nsight Systems User Guide. https://docs.nvidia.com/nsight-systems/UserGuide. Accessed: February 25, 2025.

[33] NVIDIA. 2025. NVIDIA cuBLAS Library. https://developer.nvidia.com/cublas. Accessed: February 25, 2025.

[34] NVIDIA. 2025. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. Accessed: February 25, 2025.

[35] NVIDIA. 2025. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems. Accessed: February 25, 2025.

[36] NVIDIA. 2025. Thrust: The C++ Parallel Algorithms Library. https://nvidia.github.io/cccl/thrust/. Accessed: February 25, 2025.

[37] University of Oregon. 2025. TAU Performance System. https://www.tau.uoregon.edu/. Accessed: February 25, 2025.

[38] David A Patterson and John L Hennessy. 2016. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

[39] P Victer Paul, N Moganarangan, S Sampath Kumar, R Raju, T Vengattaraman, and P Dhavachelvan. 2015. Performance analyses over population seeding techniques of the permutation-coded genetic algorithm: An empirical study based on traveling salesman problems. *Applied Soft Computing* 32 (2015), 383–402. doi:10.1016/j.asoc.2015.03.038

[40] K Raju and Niranjan N Chiplunkar. 2021. Performance enhancement of CUDA applications by overlapping data transfer and Kernel execution. *Applied Computer Science* 17, 3 (2021). doi:10.35784/acs-2021-17

[41] Score-P. 2025. Score-P: Scalable Performance Measurement Infrastructure for Parallel Codes. https://gitlab.com/score-p/scorep. Accessed: February 25, 2025.

[42] Linhai Song and Shan Lu. 2014. Statistical Debugging for Real-World Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 561–578. doi:10.1145/2660193.2660234

[43] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127, 7.2 (2012).

[44] HPC Toolkit. 2025. HPC Toolkit: Performance Tools for High-Performance Computing. https://hpctoolkit.org/. Accessed: February 25, 2025.

[45] Lars B. van den Haak, Anton Wijs, Mark van den Brand, and Marieke Huisman. 2020. Formal Methods for GPGPU Programming: Is the Demand Met?. In *Proceedings of the International Conference on Integrated Formal Methods*. 160–177. doi:10.1007/978-3-030-63461-2_9

[46] Anjia Wang, Xinyao Yi, and Yonghong Yan. 2020. Supporting data shuffle between threads in openmp. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*. Springer, 98–112. doi:10.1007/978-3-030-58144-2_7

[47] Wenhua Yang, Chong Zhang, and Minxue Pan. 2023. Understanding the Topics and Challenges of GPU Programming by Classifying and Analyzing Stack Overflow Posts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1444–1456. doi:10.1145/3611643.3616365

[48] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. 2012. Fixing performance bugs: An empirical study of open-source GPGPU programs. In *Proceedings of the 2012 41st International Conference on Parallel Processing*. 329–339. doi:10.1109/icpp.2012.30

[49] Xinyao Yi, David Stokes, Yonghong Yan, and Chunhua Liao. 2021. CUDAMicroBench: Microbenchmarks to assist CUDA performance programming. In *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops*. 397–406. doi:10.1109/ipdpsw52791.2021.00068

[50] Kohei Yoshida, Shinobu Miwa, Hayato Yamaki, and Hiroki Honda. 2024. Analyzing the impact of CUDA versions on GPU applications. *Parallel Comput.* 120 (2024), 103081. doi:10.2139/ssrn.4565014

[51] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140. doi:10.1145/3213846.3213866

[52] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*. 382–393. doi:10.1109/hpca.2011.5749745

[53] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. 2022. Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos. In *2022 IEEE International Symposium on High-Performance Computer Architecture*. 800–813. doi:10.1109/hpca53966.2022.00064

Received 2025-09-09; accepted 2025-12-22