DrGPU: A Top-Down Profiler for GPU

Yueming Hao
yhao24@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Nikhil Jain
nikhijain@nvidia.com
Nvidia Corporation
Santa Clara, California, USA

Rob Van Der Wijngaart
robv@nvidia.com
Nvidia Corporation
Santa Clara, California, USA

Nirmal Saxena
NSAXENA@nvidia.com
Nvidia Corporation
Santa Clara, California, USA

Yuanbo Fan
Yfan@tenstorrent.com
Tenstorrent Incorporated
San Francisco, California, USA

Xu Liu
xliu88@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

ABSTRACT

GPUs have become common in HPC systems to accelerate scientific computing and machine learning applications. Efficiently mapping these applications to rapid evolutions of GPU architectures for high performance is a well-known challenge. Various performance inefficiencies exist in GPU kernels that impede applications from obtaining bare-metal performance. While existing tools are able to measure these inefficiencies, they mostly focus on data collection and presentation, requiring significant manual efforts to understand the root causes for actionable optimization. Thus, we develop DrGPU, a novel profiler that performs top-down analysis to guide GPU code optimization. As its salient feature, DrGPU leverages hardware performance counters available in commodity GPUs to quantify stall cycles, decompose them into various stall reasons, pinpoint root causes, and provide intuitive optimization guidance. With the help of DrGPU, we are able to analyze important GPU benchmarks and applications and obtain nontrivial speedups — up to 1.77× on V100 and 2.03× on GTX 1650.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **General programming languages**; • **General and reference** → **Measurement**; **Metrics**.

KEYWORDS

CUDA, profiler, GPU, performance optimization, measurement

ACM Reference Format:
Yueming Hao, Nikhil Jain, Rob Van Der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A Top-Down Profiler for GPU. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578244.3583736

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04…$15.00
https://doi.org/10.1145/3578244.3583736

1 INTRODUCTION

General-purpose graphics processing units (GPGPU) have become common in HPC systems to accelerate various parallel applications. Among the latest Top 500 list [7], more than 100 supercomputers employ CPU+GPU heterogeneous architectures. At present, two of the supercomputers—Summit and Sierra—employ GPUs to provide most of the FLOPS on their compute nodes. Moreover, emerging U.S. exascale systems will be all powered by GPUs.

Unlike traditional CPUs, GPUs typically offer a unique programming and architectural scenario. For instance, they employ thousands of threads, which are divided into warps on NVIDIA GPUs or wavefronts on AMD GPUs. With the Single-Instruction Multiple-Threads (SIMT) programming model, all the threads in one warp share the same program counter and execute in lockstep. Moreover, a warp is able to coalesce multiple memory requests to adjacent memory words into one single request, so threads can benefit from spatial locality. Caches on GPUs are often limited in capacity and shared across threads.

A variety of programming models, such as CUDA [21], HIP [8], OpenACC [35], and OpenCL [32] are developed to help programmers offload computation kernels to GPUs (aka GPU kernels). In modern HPC applications, GPU kernels take a big portion of the whole program execution. However, without careful programming, it is easy to introduce inefficiencies that impede GPU kernels from obtaining bare-metal performance. For example, control flow and memory divergence can significantly hurt GPU parallelism; heavy type, which is that the array's data type is overused based on the values of this array, can result in wasted memory bandwidth since values will be copied with useless bits; over-synchronization across threads can result in execution serialization. It is important to optimize GPU kernels for optimal performance. Typically, these inefficiencies can be buried deep in complex HPC applications, and manual analysis is tedious and error-prone.

Optimizing compilers, such as nvcc [20], LLVM [1, 3, 36], DPC++ [15], are adept at improving GPU code performance, such as removing redundant computation, and efficiently utilizing registers. However, they have a narrower view of the program, which limits their analysis to a small scope—individual functions or files. Moreover, layers of abstractions, dynamically loaded libraries, multilingual components, aggregate types, aliasing, sophisticated flows of control, and the combinatorial explosion of execution paths make it practically impossible for compilers to obtain a holistic view of an application to apply its optimizations. Moreover, without dynamic

information, static analysis has no insights into the execution behaviors in GPU memory subsystems and pipelines, resulting in suboptimal optimization. Advanced compiler optimizations such as link-time optimization (LTO) and profile-guided optimization (PGO) enlarge the optimization scope and add some dynamic information, but they are still conservative. Thus, compilers often fail to eliminate many kinds of inefficiencies.

Orthogonal to the static analysis is evaluating GPU execution with simulators (e.g., GPGPUSim [9]) or emulators (e.g., Ocelot [14]). While simulators and emulators provide rich execution insights, they incur unaffordable overhead to simulate every detail of a state-of-the-art GPU architecture, which is impractical to be deployed to supercomputers to measure HPC applications. Instead, selective instrumentation with tools such as NVBit [34], GTPin [2], SASSI [6], and GVProf [37] can reduce the overhead. However, they identify pure software inefficiencies, with no insights into inefficiencies due to software-hardware interactions.

Another popular approach is profiling that identifies program inefficiencies at runtime with performance monitoring units (PMU) available in GPU architectures. Profiling tools such as Nsight Compute (NCU) [24], NVProf [28], HPCToolkit [38], TAU [31], Score-P [16], and ROCProfiler [5] leverage PMUs to collect a variety of performance events, such as cache misses, memory divergences, stall cycles, and many others. Based on these events, these tools can derive many metrics, such as instructions per cycle and cache miss ratios to measure the performance of GPU kernels.

However, existing profiling tools require substantial expertise and manual efforts to obtain actionable optimization insights. On one hand, GPU can support tens of thousands of performance events [25], monitoring all the events is impractical, so users need to manually filter out unnecessary events for monitoring. On the other hand, existing tools mostly focus on data collection and attribution with little emphasis on data analysis for actionable optimization guidance.

To address the limitations of existing tools, we develop DrGPU, an end-to-end profiler that automatically measures GPU kernel execution and provides intuitive optimization guidance. We implement DrGPU for NVIDIA GPUs but the technique is generally applicable to AMD and Intel GPUs. In summary, we make the following contributions in DrGPU:

- DrGPU is a novel top-down profiler for GPU kernels. DrGPU quantifies stall cycles and decomposes them according to various hardware events for root causes.
- DrGPU provides focused, hierarchical performance deficit attribution with minimum manual interference. Moreover, DrGPU provides detailed actionable optimization guidance that can be interpreted by non-experts.
- DrGPU has been deployed to Summit and used to optimize a variety of well-known GPU benchmarks and applications, yielding significant speedups.

2 BACKGROUND AND RELATED WORK

2.1 NVIDIA GPU and CUDA

To efficiently utilize the GPU hardware, NVIDIA introduced CUDA [21], a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs. With the help of CUDA, NVIDIA GPUs can solve complex computational problems. CUDA C++ extends C++ to support user defined functions, aka *GPU kernels*, to run on GPUs with multiple CUDA threads. These threads are organized into multiple equally-shaped *thread blocks or CTAs*, and blocks could be formed to a *grid* whose dimension is up to three. Thread blocks are further divided into *warps*, which consist of 32 consecutive threads each.

In CUDA programming, there are multiple memory types. Each thread has its own private local memory. Each thread block has shared memory visible to all threads in this block. All threads have accesses to the global, constant, and texture memory, which are optimized for different usages. Local, global, texture, constant, and surface memories all reside on *device memory*, which is the main and slowest memory layer on GPU. In contrast, shared memory is able to be configured to various sizes and is expected to be as fast as the L1 cache.

2.2 PMUs on NVIDIA GPUs

NVIDIA GPUs support tens of thousands of PMU events for program monitoring, such as `sm__warps_active`, `lts__t_requests`, and many others. One can count these event occurrences for any given GPU kernel.

Moreover, NVIDIA GPUs support an advanced monitoring facility: PC sampling [17], which performs device-wide sampling of the program counter (PC) in execution. In a fixed interval of cycles, the PMU in each GPU streaming multiprocessor collects program counters; the minimum interval can be 32 or 2048 cycles given different GPU generations. In the meanwhile, the PMU reports stall cycles with no warp scheduled for execution as well as the stall reasons, such as memory access, barriers, instruction dependencies, and others.

To program the PMUs, NVIDIA provides CUDA Profiling Tools Interface (CUPTI) [23], which can also capture GPU kernel launch and return. CUPTI has become the de-facto interface for performance analysis tools on NVIDIA GPUs.

2.3 Related Work

Existing GPU profilers mainly fall into two categories: exhaustive and lightweight analyses.

Exhaustive profilers. GPU code instrumentation engines, such as NVIDIA NVBit [34], SASSI [6], Sanitizer API [18], Intel GTPin [2], and LLVM infrastructure [3] support powerful exhaustive analysis by monitoring each executed instruction instance. For example, GVProf [37], atop Sanitizer API, is able to identify redundant values used in each memory access; CUDAAdvisor [30], based on LLVM, tracks control and data flow in GPU kernels; CUDA Flux [11] instruments GPU code via LLVM to identify redundant instructions. While exhaustive analysis tools provide unique insights with microscopic views, they incur large overhead (100-1000×) and lack architecture-specific insights. In contrast, DrGPU is a lightweight profiler.

Lightweight profilers. Vendor-provided tools, such as NVIDIA NVProf [28], Nsight Compute [24], and Nsight Systems [26] leverage CUPTI [23] and NVTX [27] to monitor hardware or user-defined events. ROCProfiler [5] on AMD GPUs and HPCToolkit [38],

TAU [31], and Score-P [16] in the research community provide similar functionality. These tools rely on GPU PMUs for data collection, usually incurring low measurement overhead. However, these tools usually need significant expert knowledge. On one hand, users need to identify necessary events to monitor; on the other hand, users need to interpret the results and identify appropriate optimization strategies. While Nsight Compute provides some optimization guidance on unit level, it is difficult to understand how inefficiency in one unit impacts other units and users, especially non-experts, to optimize the GPU kernels.

HPCToolkit [39] recently added CUPTI API-based support for performance bottleneck and root cause analysis. It constructs the CPU-GPU call paths to understand performance inefficiencies across CPU and GPU. It also leverages PC sampling and a small set of hardware events to pinpoint GPU kernel inefficiencies. It mostly pinpoints the hot spots in GPU kernels. Unlike DrGPU, it does not exhaustively analyze inefficiencies in different GPU units and provides actionable optimization guidance for non-experts. Perhaps, GPA [40, 41] is the most related tool to DrGPU. GPA uses PC sampling to quantify GPU stall cycles and performs on-the-fly program slicing to identify instructions that cause the stalls. However, unlike DrGPU, GPA does not collect hardware events to understand inefficient software-hardware interactions.

We elaborate on the comparison of DrGPU with Nsight Compute and GPA in Section 5.2.

3 METHODOLOGY

DrGPU leverages NVIDIA Nsight Compute (NCU) to collect necessary hardware events and outputs a top-down analysis tree with rich performance insights in any GPU kernel. Figure 1 shows the top level of the tree. DrGPU builds these nodes by analyzing the PC sampling results. For the root node, DrGPU derives the total stall cycles of the given GPU kernel as the GPU cycles not fully utilizing the issue slots. Starting from the Volta architecture, each streaming multiprocessor can issue four warp instructions per cycle (IPC). DrGPU calculates the percentage of stall cycles as

$$stall\ cycle\% = \frac{ideal\ IPC - achieved\ IPC}{ideal\ IPC} \quad (1)$$

It includes both horizontal and vertical stalls[1] [33] for instruction issuing. Moreover, DrGPU shows the IPC and quantifies the memory and compute intensity for the entire GPU kernel in the tree root.

Under the root, DrGPU categorizes stall cycles according to their causes. In recent NVIDIA GPU generations (including Volta, Turing, and Ampere), NCU provides 18 warp scheduler states. DrGPU identifies 13 of them that are related to warp issuing stalls and have a clear association with particular GPU units. DrGPU groups these 13 stall reasons in five categories, and decomposes the percentage of stall cycles to each stall reason (child node in Figure 1). When the future GPU generation supports more stall states, DrGPU can be easily extended.

In the remaining section, we detail the construction of each sub-tree by identifying the root cause of various stalls, generating optimization guidance, and using examples to show the effectiveness of the top-down analysis.

3.1 Device Memory Stalls

Memory subsystem in GPU devices is a major source of performance bottlenecks. The stall reason related to device memory is mainly device memory accesses. If a device memory access instruction has been issued but cannot be executed because of cache misses, or throughput limitation, the warp will be stalled. Moreover, the warp could also be stalled when memory accesses cannot be issued due to the congestion in the memory bandwidth.

Root Cause Analysis. When a warp accesses global memory, it coalesces accesses of the threads within the warp into one or more memory transactions. It depends on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. These memory transactions are then transferred to L1 cache[2]. If L1 misses occur, L2 accesses happen. If L2 misses occur, accesses to device memory are necessary.

Unit	Cycles (C_i)
L1	$L1_{latency} + cc * (L1\_per\_inst - 1)$
L2	$L1_{miss\_ratio} * L2_{latency}$
Device Memory	$L1_{miss\_ratio} * L2_{miss\_ratio} * DM_{latency}$

Table 1: Average stall cycles on each memory unit. The *cc* is the conflict cost per extra L1 access, and it equals to 2 cycles. We compute $L1\_per\_inst$ as the ratio of all L1 cache accesses over the total memory requests in a warp.

We build the sub-tree of device memory access stalls based on the average stall cycles of different layers in the GPU memory hierarchy. Table 1 shows the formula DrGPU derives for the stall cycle calculation. We decompose the average stall cycles of all warp memory instructions into L1 cache, L2 cache, and device memory. For a warp memory instruction, if all memory accesses in this warp are coalesced into one memory transaction for one L1 cache access, the latency of this instruction is $L1_{latency}$ whose value is 28 cycles on V100 GPU according to NVIDIA data sheet. But if these memory accesses are coalesced into multiple memory transactions, there is a penalty for extra memory transactions: each extra one costs 2 cycles known as conflict cost (*cc*) in Table 1. We compute $L1\_per\_inst$ as the ratio of all L1 cache accesses over the total memory requests in a warp; both events can be monitored by the NVIDIA PMU counters. Thus, the average stall cycles due to L1 cache accesses per memory instruction in a warp are calculated as follows.

$$C_{L1} = L1_{latency} + cc * (L1\_per\_inst - 1) \quad (2)$$

Since only memory transactions missed in L1 cache are transferred to L2 cache, the average stall cycle per warp memory instruction on L2 cache is calculated as follows.

$$C_{L2} = L1_{miss\_ratio} * L2_{latency} \quad (3)$$

Memory transactions missed in both L1 and L2 caches access device memory. Thus, the average stall cycle per warp memory instruction is calculated as follows.

$$C_{DM} = L1_{miss\_ratio} * L2_{miss\_ratio} * DM_{latency} \quad (4)$$

[1] Vertical stall is introduced when the GPU issues no instructions in a cycle, horizontal stall when not all issue slots can be filled in a cycle.

[2] Global memory accesses for devices of compute capability 3.5 or 3.7 are normally not cached in L1. Moreover, 1d instructions with modifier cg will bypass L1 cache.

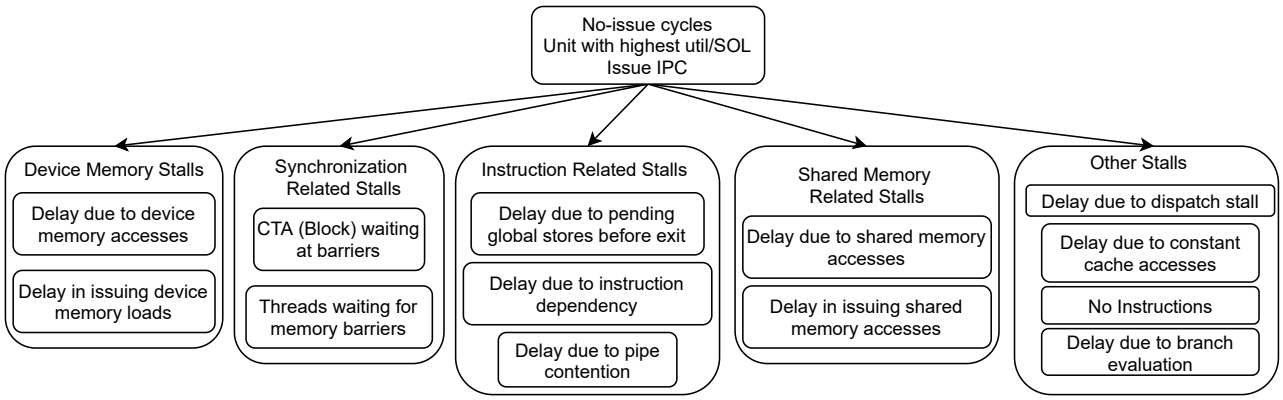

Figure 1: DrGPU derives the total stall cycles ratio as the root node and identifies 13 stall reasons in five categories.

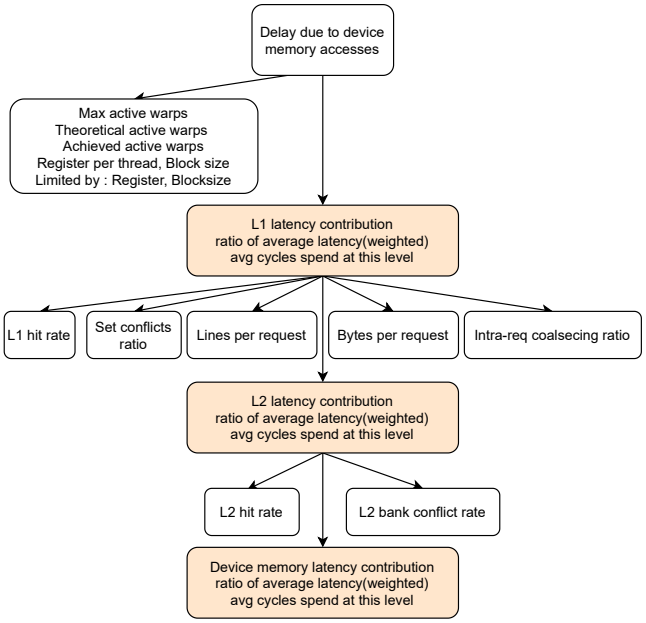

Figure 2: The sub-tree of stalls due to device memory accesses.

$L2_{latency}$ is about 200 cycles on V100 GPU, and $DM_{latency}$ is 250 cycles on V100 GPU according to NVIDIA data sheet. PMUs events $L1_{miss\_ratio}$ and $L2_{miss\_ratio}$ are available for monitoring.

The percentage of stall cycle contribution on each memory layer is normalized to the sum of average stalls in all the layers. For example, the stall cycle contribution of L1 is $\frac{C_{L1}}{C_{L1}+C_{L2}+C_{DM}}$.

DrGPU can further derive various metrics, such as hit rate, conflicts, bytes per request in each memory layer as shown in the sub-tree to give deep insights about cycle stalls.

Suggestions. DrGPU provides rich suggestions for this sub-tree. If L1 cache latency is high, DrGPU suggests to use temporary variables in registers to reduce L1 cache utilization. If the cache conflict rate or miss rate of L1 or L2 caches are high, DrGPU suggests to change memory access patterns, such as reordering code, unrolling loops, or moving frequently used data to shared memory. For Ampere and successors, DrGPU further suggests to change L2 cache persistence policy for frequently used data blocks.

```
1  __global__ void findRangeK(...){
2  for(i = 0; i < height; i++){
3    if((knodesD[currKnodeD[bid]].keys[thid] <= startD[bid])
4    && (knodesD[currKnodeD[bid]].keys[thid+1] > startD[bid])){
5      if(knodesD[currKnodeD[bid]].indices[thid] < knodes_elem)
6        offsetD[bid] = knodesD[currKnodeD[bid]].indices[thid];
7    }
8    if((knodesD[lastKnodeD[bid]].keys[thid] <= endD[bid]) &&
     ↪  (knodesD[lastKnodeD[bid]].keys[thid+1] > endD[bid])){
9      if(knodesD[lastKnodeD[bid]].indices[thid] < knodes_elem){
10       offset_2D[bid]=knodesD[lastKnodeD[bid]].indices[thid];
11   ... }
```

Listing 1: In kernel `findRangeK`, loop invariant `endD[bid]` in Line 4-12 still incurs memory requests that can be optimized by moving out of the for loop.

Example. In kernel `findRangeK` of Rodinia `b+tree`, device memory access stalls account for 63.34% of no-issue cycles. The L1 miss rate is 26.68%. DrGPU reports the bottlenecks residing in L1 cache and suggests to reduce the L1 cache transactions. DrGPU further shows the problematic code in Listing 1. With further investigation, we find that memory accesses to `startD[bid]`, `endD[bid]`, `knodesD[currKnodeD[bid]`, and `knodesD[lastKnodeD[bid]]` are all loop invariant; the compiler fails to hoist them outside of the loop. Therefore, we manually hoist these loop invariants out of the loop, which reduces total stall cycles by 35.48% and yields a 5.15× speedup on NVIDA GTX 1650 and a 1.15× speedup on NVIDA V100, V100 receives less speedup because V100's memory bandwidth is 7× of GTX 1650.

3.2 Synchronization-Related Stalls

There are two stall reasons related to synchronization: block barriers and memory barriers. Stalled warps can wait at block barriers for their sibling warps in the same CTA/block under execution. The API `__syncthreads()` provided by CUDA can perform explicit block synchronization. All threads in the same block must wait at this API, which are treated as stalls. In addition, warps can be stalled due to waiting at memory barriers. The `membar/fence`

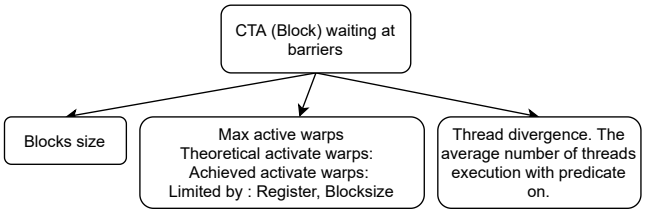

Figure 3: The sub-tree of stall reason that CTA (Block) waiting at barriers.

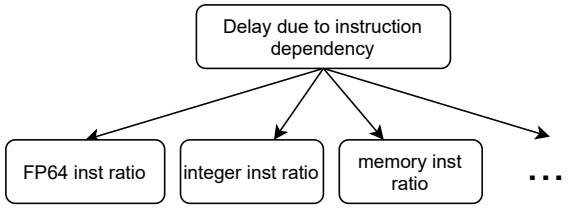

Figure 4: The sub-tree of stall due to instruction dependency.

instructions are such memory barriers that guarantee the order of memory operations.

Root Cause Analysis. When there are too many block barriers and significant workload imbalance across threads, warps may be stalled waiting for threads synchronization. Figure 3 illustrates different root causes for the stalls due to the block barrier. Blocks are executed independently on stream multiprocessors and `__syncthreads()` only works for threads belonging to one block. Thus, if we reduce the block size, fewer threads need to wait for the block barrier. Moreover, an insufficient number of active warps may also lead to stalls due to the failure of scheduling active warps to overlap the stalled ones.

For GPU architectures prior to Volta, warps execute instructions in a lock-step mode, which means that all threads in the same warp share the same program counter. Branches can significantly hurt performance resulting in thread divergence. Although Volta generation independently schedules threads in a warp, high thread divergence can still introduce workload imbalance across threads, causing block barrier stalls.

The memory barriers can synchronize threads beyond a single block. Over-synchronizing a large scope of threads can trigger unnecessary stalls.

Suggestions. For block barrier stalls, if the block size is larger than the warp size, we suggest to reduce the block size. If the block size could be reduced to 32 which is the fixed number of threads in a warp, we suggest to replace `__syncthreads()` with `__syncwarp()`, which has less overhead when the data dependency could be guaranteed by access pattern. If the amount of theoretical active warps is low, and register usage pressure is easy to reduce without significant performance loss, we suggest to add more concurrent warps. For memory barrier stalls, we suggest to reduce the scope of the memory barrier to warp or thread block by restructuring code.

3.3 Instruction-Related Stalls

There are three stall reasons related to instruction dependencies or distributions: instruction dependency, pipeline contention, and pending global stores before exit. Figure 4 and Figure 5 show the sub-trees of the first two reasons; the last one does not have a sub-tree.

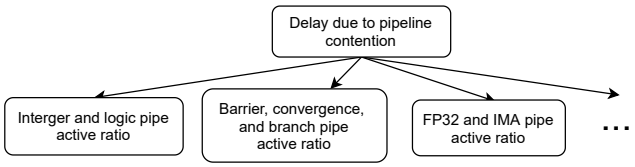

Figure 5: The sub-tree of stall due to pipeline contention.

```
__global__ void calculate_temp(...)
  ...
  temp_t[ty][tx] = temp_on_cuda[ty][tx] + step_div_Cap * (power_on_cuda[ty][tx] +
 ▶ (temp_on_cuda[S][tx] + temp_on_cuda[N][tx] - 2.0 * temp_on_cuda[ty][tx]) *
 ▶    Ry_1 + ...  - 2.0 * temp_on_cuda[ty][tx]) * Rx_1 +
```

Listing 2: Inappropriate constant floating point number usage in hotspot. The constant floating point number 2.0 occurs unnecessary type conversion.

Root Cause Analysis. If two instructions depend on each other and there are not enough other intervening instructions for overlapping, GPU stalls occur. Figure 4 analyzes the instruction mix and understands the type of instructions that cause the stalls.

Furthermore, NVIDIA GPUs have multiple pipelines, which could execute different kinds of instructions concurrently. If a kernel could utilize these pipelines efficiently, instruction latency can be hidden nicely. If a kernel oversubscribes some pipelines, e.g., FP32 pipelines, warps need to be stalled until the pipelines become available again. DrGPU computes the active ratios of various pipelines as shown in Figure 5 to locate pipeline contentions.

Finally, pending global stores before exit are triggered when a large amount of data are written back to memory at the end of the kernel.

Suggestions. For stalls due to instruction dependency, DrGPU suggests to restructure or unroll the problematic codes to have enough independent instructions hiding the stalls. Moreover, DrGPU suggests to reduce the instruction latency on the critical path. For example, DrGPU recommends using compiler option `use_fast_math` or changing functions to their fast versions to minimize the stalls, though this should always be attempted with extreme caution and with the concurrence of domain scientists.

For stalls due to pipeline contention, DrGPU suggests to balance the instruction mix to avoid the contention in a subset of pipelines.

Example. DrGPU reports that the kernel `calculate_temp` in Rodinia `hotspot` accounts for 5.37% stall cycles due to instruction dependency; FP32 operations account for 12.46% of all instructions and type conversion related instructions account for 5.8% of all instructions. DrGPU suggests to check the `F2F` conversion instructions. List 2 highlights the source codes identified by DrGPU, which shows the constant floating point values (i.e., 2.0) used in the computation. As the compiler automatically assigns the FP64 type to the constant FP value, the computation needs to convert all the other FP32 values to FP64 for computation. To remove these unnecessary type conversion instructions, we explicitly declare these constant

```
1 __global__ void dynproc_kernel(...){
2   __shared__ int result[BLOCK_SIZE];
3   result[tx] = shortest + gpuWall[index];
4   if (computed) // Assign the computation range
5 ▶     prev[tx] = result[tx];
6   gpuResults[xidx] = result[tx];
```

Listing 3: Overused shared memory in pathfinder.

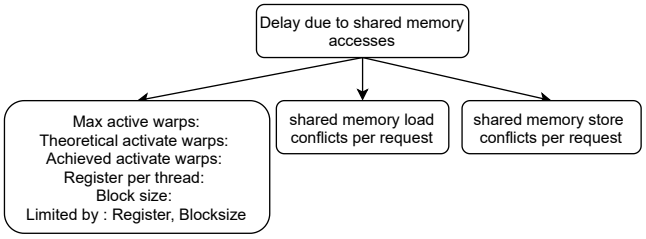

Figure 6: The sub-tree of stalls due to shared memory accesses.

values as FP32, which yields 5.15× and 1.15× speedups on NVIDIA GTX 1650 and NVIDIA V100, respectively.

3.4 Shared Memory Stalls

Figure 6 shows the sub-tree for the stalls due to shared memory accesses. DrGPU shows active warps, shared memory load, and store conflicts per requests in this sub-tree to further understand the root causes of stalls.

Root Causes Analysis. To achieve high bandwidth, shared memory is divided into equally-sized memory banks, which can be accessed concurrently. However, bank conflict can happen if multiple operations access to the same back together [19], which serializes these memory accesses. The stalls due to shared memory can be caused by large amounts of bank conflicts and the saturation of shared memory bandwidth.

Suggestions. If the number of bank conflicts is high, DrGPU suggests to restructure code or use loop unrolling to hide shared memory latency. If applicable, DrGPU further suggests to hoist values in registers to reduce share memory accesses. For stall due to shared memory bandwidth saturation, DrGPU suggests to reduce the number of concurrent shared memory loads via restructuring code, issuing wider loads, spreading the loads, or reducing the loop unrolling factor. For Ampere and newer generations, DrGPU also suggests to use asynchronous shared memory copy to reduce the stalls.

Example. In the kernel `dynproc_kernel` of Rodinia `pathfinder`, accessing to the shared memory accounts for 6.17% of no-issue cycles. Line 7 in Listing 3 accounts for 60.32% of total shared memory access stalls. With the guidance of DrGPU, we further study the source code and find that each thread only accesses one element in array `result`. Thus, we replace `result` allocated in the shared memory with scalar variables, yielding 1.14× and 1.07× speedups on NVIDIA GTX 1650 and NVIDIA V100, respectively.

3.5 Other Stalls

There are other four stall reasons DrGPU is able to identify: dispatch, constant cache accesses, no instructions, and branch evaluation. When a kernel has excessive branches, warps are stalled waiting for branch targets to be resolved. The dispatch stalls could be due to the limited register read bandwidth. "No instructions" stall occurs when warps are stalled at instruction cache misses.

For branch evaluation stalls, DrGPU suggests to reduce control divergence. For constant cache access stalls, DrGPU suggests to replace constant memory usage with device memory or change memory access patterns.

4 IMPLEMENTATION

Figure 7 shows the workflow of DrGPU. DrGPU accepts fully optimized binary code and leverages NVIDIA Nsight Compute (NCU) to collect necessary hardware events. DrGPU then processes the performance data offline and outputs the top-down analysis tree with rich performance insights. We implement DrGPU to monitor applications running on single and multiple GPUs, but with a focus on GPU, not network or CPU activity. DrGPU works on NVIDIA GPUs of Maxwell architectures and its successors whose compute capability is 5.2 and beyond.

4.1 Online Data Collection

DrGPU configures NCU to perform PC sampling as well as collect 69 necessary hardware events out of over thousands of events supported in NVIDIA GPUs. We select these events according to experiences from NVIDIA performance engineers as well as our experiments. The details of these hardware events are available via this paper's artifacts. To collect all these events, DrGPU may need to run a program multiple times. By default, DrGPU monitors every GPU kernel in a program. To reduce the overhead, DrGPU is able to profile selected GPU kernels that are hotspots.

With the help of NCU, DrGPU records the performance details of any GPU kernel under investigation in a profile, including kernel names, statistics of the hardware events, and detailed line mapping. In a multi-GPU execution, DrGPU outputs a profile per GPU.

4.2 Offline Data Processing

The offline component of DrGPU accepts all the profiles produced by the online component. There are three main tasks done offline: merging profiles from multiple GPUs, creating a top-down analysis tree, and generating optimization guidance.

Profile Coalesce. DrGPU coalesces the profiles from different GPUs to give an aggregate analysis. The coalescing process employs a hierarchical strategy: DrGPU first merges kernels of the same names and then the program counters of the same value; the statistics of cycle stalls and hardware events are summed up during the merging process. All the follow-up analyses are based on the aggregate profile.

Tree Construction. DrGPU creates the top-down analysis tree based on the technique described in Section 3. Figure 11 shows part of the tree in dot format produced by DrGPU. The root denotes the percentage of no-issue cycles among all execution cycles and the unit with the highest utilization. From the root node, we could

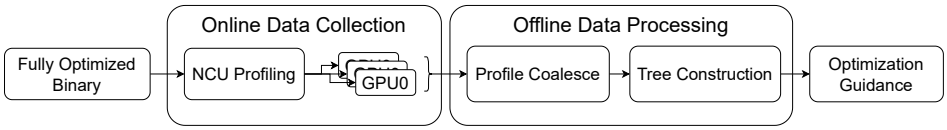

Figure 7: Workflow of DrGPU.

GPU	GPU Memory	L1 Cache	L2 Cache	Linux Kernel	GPU Driver	CUDA Toolkit	Nsight Compute	GCC
GTX 1650	4GB	up to 64KB/SM	1MB	Linux 5.8.0	460.27	11.2.0	2020.3.1	9.3.0
V100	16GB	up to 128KB/SM	6MB	Linux 4.14.0	418.116	11.2.0	2020.3.0	7.4.0

Table 2: The configurations of two GPU systems to evaluate DrGPU: GTX 1650 and V100 GPUs.

immediately know the kernel's optimization state. We observe that usually when no-issue cycles are less than 50% of all cycles in an HPC application, one of the relevant hardware units is fully utilized and the application has been fairly well optimized. The node in the first level denotes the ratio of stall due to instruction dependency of no-issue cycles. And the leaf nodes (in gray) in this branch are the decomposition of this stall reason. The leaf nodes in pink show corresponding kernel code contributions, which is the percentage following each line, to the current stall. The leaf nodes in green are suggestions provided by DrGPU.

Guidance Generation. DrGPU produces optimization guidance according to the experiences from NVIDIA performance engineers as well as our experiments. DrGPU has 39 optimization guidance. The thresholds are defined in the tree nodes to enable DrGPU highlights the top-down path reaching to the optimization guidance that users can immediately take actions. For example, Figure 9 shows the performance analysis tree for PeleC, we can easily change the functions used in the source code block to the fast version by following the suggestion. DrGPU defines these thresholds empirically. The details are in DrGPU's artifact. DrGPU is highly extensible with providing additional optimization guidance and configurable thresholds to make accurate optimization decisions for applications running on NVIDIA GPUs of different generations. For example, besides the suggestions for Volta (V100) and Turing (GTX 1650), DrGPU supports additional optimization suggestions for A100 (Ampere) GPUs, such as adding asynchronous shared memory copy.

4.3 Tool Usage

DrGPU can launch an application and generate the profiling report, with no need to modify or recompile the application code. Users can then revise the code according to DrGPU's optimization suggestions in the report. It is worth noting that the difficulty of applying DrGPU's optimization suggestions varies. Some suggestions only need to change a compilation flag, while some others suggest to overhaul the GPU algorithms if possible. Users need to make the decision whether to take the suggestions or not.

5 EVALUATION

We evaluate DrGPU on two NVIDIA GPU platforms: a desktop with GTX 1650 GPU and a Summit node with V100 GPUs. Table 2 elaborates the configurations.

We utilize DrGPU to analyze all programs of Rodinia benchmarks [13] and three real CUDA applications in important HPC and deep learning domains. We describe these applications as follows:

- YOLOv4 [10], which is a state-of-the-art real-time object detector. YOLOv4 is based on Darknet [29], a popular deep learning framework. We run YOLOv4 with a built-in neural network and pre-trained `yolov4.weights` to detect dogs.
- PeleC [4], a DoE Exascale Computing Project (ECP) application, performs adaptive-mesh compressible hydrodynamics for reacting flows. We run PMF test in PeleC with its default input.
- Castro [12], an adaptive mesh, radiation hydrodynmics application to model astrophysical reacting flows. We study the example `Sedov` of Castro with input `inputs.2d.cyl_in_cartcoords`. We run Castro on multi-GPUs to evaluate DrGPU.

All the programs are compiled with `-O3` and run 10 times on a single GPU device. Monitoring and optimizing applications running on multi-GPUs and multi-nodes follow the same way. DrGPU relies on NCU to collect all the necessary PMU events and replay a GPU kernel for 35 passes on GTX 1650 and 73 passes on V100, given different NCU versions and different GPU architectures. Table 3 summarizes the optimizations done on these applications guided by DrGPU, including 13 Rodinia benchmarks and all four real applications. With the guidance of DrGPU, a Ph.D. student with no prior knowledge about the programs spends about 1.5 hours on each program for optimization. On average, we receive a 1.58× kernel-level speedup on GTX 1650 and a 1.36× kernel-level speedup on V100.

5.1 Discussions

Optimization Summary. Besides the nontrivial speedups guided by DrGPU, we obtain several insights from our optimization. First, loop unrolling is one of the most actionable optimization methods; CUDA compilers do not unroll loops in an efficient way, even with the highest optimization. Second, the speedups guided by DrGPU vary in different GPU architectures with the same optimization. For example, the optimization for `hotspot` on GTX 1650 is much larger than V100, because GTX 1650 has a weaker FP64 capability compared to V100. Furthermore, DrGPU identifies bottlenecks in V100, but not in GTX 1650 for `heartwall`. The loop unrolling does not work for GTX 1650 because it increases the burden on the relatively weaker memory units compared to V100.

Application	Kernel	State	Optimization	Newly Found by DrGPU	GTX 1650		V100	
					Original	Speedup	Original	Speedup
bfs	Kernel	Long Scoreboard	Loop unrolling		6829us	1.02x	570us	1.09x
heartwall	kernel	Wait	Loop unrolling		194ms	0.95x	85ms	1.36x
huffman	vlc_encode_kernel_sm64huff	Barriers	Restruct code		774us	1.04x	123us	1.06x
kmeans	kmeansPoint	Wait	Loop unrolling		7134us	1.12x	806us	1.16x
lud	lud_diagonal	Wait/ Short Scoreboard/ No instruction	Restruct code		212us	1.39x	223us	1.44x
myocyte	solver_2	Short Scoreboard	Function spliting		472ms	1.07x	281ms	1.22x
		Math Pipe Throttle	Add use_fast_math					
backprop	bpnn_layerforward_CUDA	Barrier	Remove unnecessary barriers		144us	1.35x	17.4us	1.36x
		Wait	Restruct code	✓				
b+tree	findRangeK	Long Scoreboard	Restruct code		425us	2.03x	50us	1.55x
		Barrier	Reduce blocksize	✓				
hotspot	calculate_temp	Wait	Remove inappropriate FP convertion		336us	5.84x	13us	1.33x
			Add use_fast_math	✓				
lavaMD	kernel_gpu_cuda	Long Scoreboard/Wait	Loop unrolling		241ms	1.50x	3492us	1.77x
		Wait	Replace speical FP functions	✓				
nw	needle_cuda_shared_1	Barriers	Remove unnecessary barriers		12ms	1.11x	834us	1.25x
			Replace syncthreads with sync warp safely	✓				
sradv1	reduce	Short Scoreboard	Loop unrolling		63us	1.57x	19us	1.38x
		Barrier	Reduce blocksize	✓				
pathfinder	dynproc_kernel	Short Scoreboard	Replace shared memory with variables		560us	1.22x	94us	1.15x
		Wait	Remove unnecessary iterations	✓				
Darknet	im2col_gpu_kernel_ext	Wait	Loop unrolling	✓	1543us	1.06x	214us	1.00x
LULESH2	ApplyMaterialProperties AndUpdateVolume_kernel	Wait	Add use_fast_math	✓	9586us	2.26x	8712us	2.56x
Pelec	react_state	Long Scoreboard	Increase occupancy		757ms	1.34x	32ms	1.36x
		Wait	Replace speical FP functions	✓				
Castro	trace_ppm	long scoreboard	Increase occupancy	✓	1431ms	1.04x	152ms	1.11x
Average						1.58x		1.36x

Table 3: Optimization summary of all the programs investigated by DrGPU. We show the problematic GPU kernels, root causes of inefficiencies, optimization techniques, and speedups. Adding `use_fast_math` or replacing special FP functions may cause accuracy loss. Many optimizations are firstly reported by DrGPU.

Application	GTX 1650	V100
	Runtime Overhead	Runtime Overhead
LULESH	3.10×	11.07×
Darknet	5.16×	4.78×
PeleC	8.65×	118.47×
Castro	3.02×	83.68×
Average	4.98×	54.50×

Table 4: We measure the overhead of DrGPU with four real applications on GTX 1650 and V100.

Overhead Analysis. Table 4 shows the overhead of DrGPU. We evaluate four real applications on GTX 1650 and V100. Since offline data processing always finishes in several milliseconds, we only measure the overhead of online data collection when using NCU to collect needed events. On average, DrGPU occurs 4.98× overhead on GTX 1650 and 54.50× overhead on V100. Summit has a relatively old version of GPU driver and NCU, which are the potential reason for the fact that DrGPU incurs a much larger overhead on V100 than on GTX 1650. The latest GPU driver and NCU, which are installed in GTX 1650 GPU, largely reduces the kernel reply times and improves the data collection overhead.

5.2 Comparison with Past Work

GPA and Nsight Compute are two state-of-the-art tools that are most related to DrGPU. We give a detailed comparison with them to distinguish our approaches.

Comparison with GPA. First, unlike DrGPU, GPA does not monitor hardware events via performance counters on GPU. Second, GPA analyzes the instruction dependencies for the root causes, limiting the insights to individual threads; instead, DrGPU monitors the entire GPU execution, avoiding myopic insights. Third, GPA works on Volta generation only, but DrGPU is applicable to wide generations such as Volta, Turing, and Ampere. Thus, DrGPU outperforms GPA in identifying more bottlenecks and guiding higher optimization speedups. We compare the insights obtained from the two tools in Rodinia benchmarks and PeleC because they are common programs investigated by both tools. First, we find that all the optimization opportunities identified by GPA are also identified by DrGPU. Moreover, DrGPU pinpoints unique bottlenecks that GPA does not identify. Table 3 highlights the performance issues newly found by DrGPU compared with GPA. Figure 8 highlights that DrGPU provides additional guidance to optimize inefficiencies in six Rodinia benchmarks and PeleC, and yields more speedups over GPA. Furthermore, we elaborate more insights obtained by DrGPU in PeleC in Section 6.1. The overhead of GPA depends on the size of GPU kernels. GPA typically incurs 100× overhead for the applications studied in this paper, while DrGPU incurs 5× overhead.

Comparison with Nsight Compute. DrGPU employs Nsight Compute for collecting hardware events for its analysis. As described in Section 2, while Nsight Compute presents and analyzes

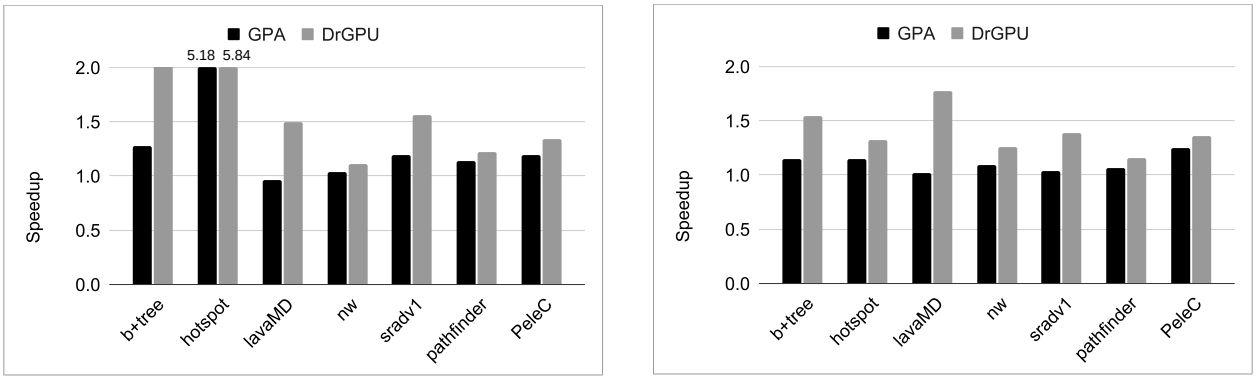

Figure 8: Comparing the optimization speedups under the guidance of GPA and DrGPU on common benchmarks and applications. DrGPU is able to guide higher speedups on both GTX 1650 and V100 GPUs. The left figure is the result on GTX 1650 GPU and the right figure is on V100 GPU.

the gathered data at unit level, e.g. SM, L2 Caches, etc., DrGPU uses hardware events to conduct cross-unit analysis as well. This relieves the end users from requiring to understand how different units interact, how the inefficiencies reported at unit level impact other units, and how these relate to the root cause of performance degradation. Given the top-down analysis, DrGPU provides more targeted and actionable suggestions, such as reducing the block size to reduce barrier stalls, unrolling loops for better memory accesses, trying the fast math option, and many more.

6 CASE STUDIES

With the help of DrGPU, we are able to optimize PeleC, Castro, LULESH 2.0, and YOLOv4. Many of the bottlenecks in these applications DrGPU identifies are firstly reported.

6.1 PeleC

DrGPU identifies the hot GPU kernel `react_state`, which accounts for 57.0% of total execution time. Figure 9 shows the analysis tree produced by DrGPU for this kernel running on GTX 1650. DrGPU produces a similar tree on V100. DrGPU reports 97.81% stall cycles due to device memory accesses and instruction dependency. SMs has the highest utilization 48.54% among units in GPU. The max active warps limited by GTX 1650 is 32. According to this kernel's resource usage and block size, the number of theoretical active warps is 8 and its achieved active warps is 7.98. DrGPU also reports that kernel occupancy is limited by register and block size. DrGPU first suggests to increase active warps for higher kernel occupancy. Since kernel occupancy is limited by register and block size, reducing thread register usage and block size are two ways to increase occupancy. DrGPU reports the per-thread register usage is as high as 254, so reducing register usage with `maxregcount` compilation flag can significantly hurt the performance. Instead, we explore the technique to reduce the block size. By reducing block size from 256 to 128, we get 1.19× speedup on GTX 1650 and 1.24× speedup on V100.

DrGPU further reports 3.92% stall cycles due to instruction dependency. 62% of all instructions are FP64(64-bit floating point)

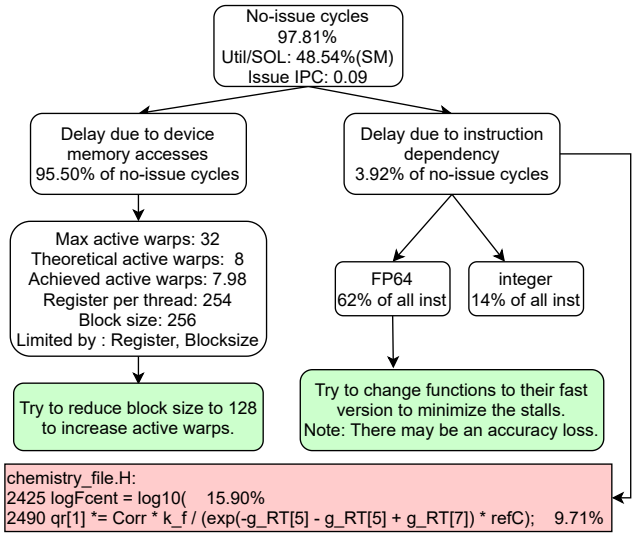

Figure 9: A portion of the performance analysis tree for the hot GPU kernel in PeleC, which shows the performance bottlenecks due to memory accesses and FP64 instruction dependencies. DrGPU gives detailed guidance for these two bottlenecks. The bottom box shows corresponding source code lines with their line indexes and contributions to this stall.

related, thus DrGPU suggests to balance the floating point instructions by changing specific functions to the fast versions with lower precision. We manually changed double precision functions highlighted in the source code block (in pink) in Figure 9 to single precision functions. Specifically, we change `log10` to `log10f`, and `exp` to `expf`. We use the `fcompare` tool provided by AMRex to check the accuracy loss of the program output, and observe less than 0.1% relative error among all metrics.

With these two optimizations, we obtained 1.34× speedup on GTX 1650 and 1.36× speedup on V100 for kernel `react_state`. It

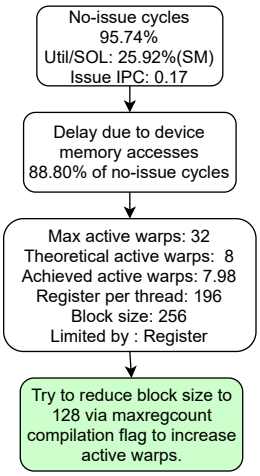

Figure 10: A portion of the performance analysis tree for the hot GPU kernel in Castro, which shows the performance bottlenecks due to memory accesses. DrGPU suggests to reduce register usage to increase occupancy.

```
__global__ void im2col_gpu_kernel_ext(const int n, const float* data_im, const
↪  int kernel_h, const int kernel_w, ...){
+ #pragma unroll 4
+ for (int index = blockIdx.x * blockDim.x + threadIdx.x; index < n;
+       index += blockDim.x * gridDim.x){
-   CUDA_KERNEL_LOOP(index, n) {
    for (int i = 0; i < kernel_h; ++i) {
      for (int j = 0; j < kernel_w; ++j) {
```

Listing 4: Unrolling the important loop in YOLOv4.

is worth noting that GPA reports the first performance issue but not the second one.

6.2 Castro

We study the hottest kernel `trace_ppm`, which accounts for 19.65% of the entire program execution on GTX 1650. Figure 10 shows the analysis tree produced by DrGPU for this kernel running on GTX 1650. DrGPU produces a similar tree on V100. DrGPU reports 88.80% stall cycles caused by device memory accesses. With further drilling down of the tree, we find that the number of theoretical active warps is 8, which is much smaller than the max allowable active warps, i.e., 32. DrGPU reports the occupancy is limited by the register usage, which is 192 per thread. To increase occupancy, DrGPU suggests to reduce register usage to 128 with `maxregcount` compilation flag. By applying this optimization, we obtain a 1.04× speedup on GTX 1650 and 1.11× speedup on V100 for this kernel.

We also run Castro with 4 GPUs across 4 Summit nodes: each node has one MPI process and each MPI process uses a GPU. DrGPU produces a similar tree and gives the same suggestion for this kernel optimization. Our optimization yields a 1.04× speedup for this kernel across all GPUs.

6.3 YOLOv4

YOLOv4 is built atop Darknet, which utilizes cuBLAS [22], a closed-source library from NVIDIA to perform basic linear algebra operations. Figure 11 shows the analysis tree produced by DrGPU

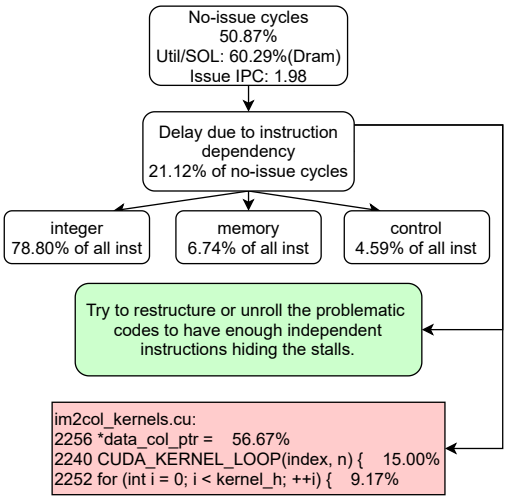

Figure 11: The portion of performance analysis tree for `im2col_gpu_kernel_ext` in YOLOv4, which shows a performance bottleneck due to instruction dependencies. Optimization with loop unrolling is suggested.

on GTX 1650, which highlights the two hot GPU kernels. The cuBLAS kernel (sgemm) accounts for half of the entire application execution time. Unfortunately, we cannot directly optimize the kernel even with the optimization guidance. However, this insight is useful for cuBLAS developers to improve their libraries.

We further study the second hottest kernel `im2col_gpu_kernel_ext`, which accounts for 18.79% of total execution time. This kernel transforms a row-major image array into a column-major one. DrGPU reports 76.34% stall cycles in this kernel, and the top two stall reasons—device memory accesses and instruction dependency—account for 95% of total idle cycles in this kernel. DrGPU suggests to restructure the kernel or unroll the loops to reduce the control instructions and increase the length of instructions sequences, which can help overlap long latency instructions.

With the code investigation, we find that this GPU kernel has three `for` loops and the compiler does not unroll them for some unknown reasons even with the highest optimization option. Figure 11 represents related codes. The first loop is in Line 2240. Based on DrGPU's suggestion, we unroll these three loops with a factor of 4 as List 4 shows. This optimization reduces the stall cycles by 15.45%, yielding a 1.06× speedup to this kernel, and a 1.04× speedup to the entire application on GTX 1650.

It is worth noting that DrGPU does not report such a bottleneck on V100 GPU. DrGPU shows a significantly smaller amount of stall cycles for this GPU kernel running on V100. It is because the memory bandwidth of V100 is 7 times to GTX 1650's, also it has 80 SMs while GTX 1650 only has 14 SMs. To verify DrGPU produces the correct guidance, we apply the same loop unrolling optimization anyway but receive no noticeable speedup.

7 CONCLUSIONS AND FUTURE WORK

This paper introduces DrGPU, a novel top-down profiler for GPU kernels. DrGPU quantifies stall cycles and decomposes them according to various hardware events for root causes. To provide intuitive optimization guidance, DrGPU automatically collects all the necessary hardware events and generates a performance analysis tree for each investigated GPU kernel. The performance analysis tree consists of rich information on the inefficiencies, including source code location, root causes, and actionable guidance. With the insights provided by DrGPU, we are able to optimize Rodinia benchmarks, HPC applications, and a machine learning framework with nontrivial speedups on both desktop and Summit NVIDIA GPUs. On average, we receive a 1.58× kernel-level speedup on GTX 1650 and a 1.36× kernel-level speedup on V100. DrGPU is ready for deployment.

We plan to extend DrGPU in three ways. First, we will further extend DrGPU to identify more performance inefficiencies and provide more intuitive optimization guidance. Second, we will work with the NVIDIA team to integrate DrGPU with their Nsight Compute performance tool. Third, we will port DrGPU to AMD and Intel GPUs by exploring their performance monitoring units.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for the constructive feedback. This work is partially supported by NSF grants of No. 2125813 and 2050007.

REFERENCES

[1] 2021. *Compiling CUDA with clang.* https://llvm.org/docs/CompileCudaWithLLVM.html [Accessed May 29, 2021].
[2] 2021. *GTPin - A Dynamic Binary Instrumentation Framework.* https://software.intel.com/content/www/us/en/develop/articles/gtpin.html [Accessed April 6, 2021].
[3] 2021. *The LLVM Compiler Infrastructure.* https://llvm.org/ [Accessed April 6, 2021].
[4] 2021. *PeleC.* https://github.com/AMReX-Combustion/PeleC [Accessed April 6, 2021].
[5] 2021. *ROC-profiler.* https://github.com/ROCm-Developer-Tools/rocprofiler [Accessed April 6, 2021].
[6] 2021. *SASSI Instrumentation Tool for NVIDIA GPUs.* https://github.com/NVlabs/SASSI [Accessed April 6, 2021].
[7] 2021. *Top 500 List.* https://www.top500.org [Accessed August 25, 2021].
[8] AMD Corporation. 2021. *HIP Programming Guide.* https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html [Accessed April 6, 2021].
[9] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE Int'l Symp. on Performance Analysis of Systems and Software*. IEEE, 163–174.
[10] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv:2004.10934 [cs.CV]
[11] 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)*. IEEE, 73–81.
[12] Almgren A CASTRO. 2010. A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity/A. Almgren, et al. *The Astrophysical Journal* 715 (2010), 1221–1238.
[13] 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 Int'l Symp. on workload characterization (IISWC)*. Ieee, 44–54.
[14] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *19th Int'l Conference on Parallel Architecture and Compilation Techniques (PACT-19)*, Vol. 10.
[15] Intel Inc. 2021. *Data Parallel C++.* https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/oneapi-programming-model/data-parallel-c-dpc.html [Accessed April 6, 2021].
[16] Dieteran Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knapfer, Daniel Lorenz, Allen Malony, WolfgangE. Nagel, Yury Oleynik, Christian Rassel, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Unified Performance Measurement System for Petascale Applications. In *Competence in High Performance Computing 2010*, Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum (Eds.). Springer Berlin Heidelberg, 85–97.
[17] NVIDIA Corporation. [n.d.]. NVIDIA PC sampling view. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#pc-sampling.
[18] NVIDIA Corporation. 2020. *NVIDIA Compute Sanitizer.* https://docs.nvidia.com/cuda/compute-sanitizer/index.html [Accessed March 26, 2021].
[19] NVIDIA Corporation. 2021. *CUDA C++ Programming Guide.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html [Accessed March 8, 2021].
[20] NVIDIA Corporation. 2021. *CUDA compiler driver, NVCC.* https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html [Accessed April 6, 2021].
[21] NVIDIA Corporation. 2021. *CUDA Toolkit.* https://developer.nvidia.com/cuda-toolkit [Accessed April 6, 2021].
[22] NVIDIA Corporation. 2021. *NVIDIA cuBLAS.* https://developer.nvidia.com/cublas [Accessed March 26, 2021].
[23] NVIDIA Corporation. 2021. *NVIDIA CUPTI.* https://docs.nvidia.com/cupti/Cupti/index.html [Accessed May 9, 2021].
[24] NVIDIA Corporation. 2021. *NVIDIA Nsight Compute.* https://developer.nvidia.com/nsight-compute [Accessed Aug 9, 2021].
[25] NVIDIA Corporation. 2021. *NVIDIA Nsight Compute Kernel Profiling Guide.* https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html [Accessed March 8, 2021].
[26] NVIDIA Corporation. 2021. *NVIDIA Nsight Systems.* https://developer.nvidia.com/nsight-systems [Accessed March 9, 2021].
[27] NVIDIA Corporation. 2021. *NVIDIA Tools Extension (NVTX).* https://docs.nvidia.com/gameworks/content/gameworkslibrary/nvtx/nvidia_tools_extension_library_nvtx.htm [Accessed May 9, 2021].
[28] NVIDIA Corporation. 2021. *The user manual for NVIDIA profiling tools for optimizing performance of CUDA applications.* https://docs.nvidia.com/cuda/profiler-users-guide [Accessed March 9, 2021].
[29] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[30] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 Int'l Symp. on Code Generation and Optimization*. 214–227.
[31] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The Int'l Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.
[32] The Khronos Group Inc. 2021. *OPENCL.* https://www.khronos.org/opencl/ [Accessed April 6, 2021].
[33] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual Int'l Symp. on Computer architecture*. 392–403.
[34] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. 2019. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM Int'l Symp. on Microarchitecture*. 372–383.
[35] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—first experiences with real-world applications. In *European Conference on Parallel Processing*. Springer, 859–870.
[36] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. gpucc: an open-source GPGPU compiler. In *Proceedings of the 2016 Int'l Symp. on Code Generation and Optimization*. 105–116.
[37] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2020. GVProf: a value profiler for GPU-based clusters. In *2020 SC20: Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1263–1278.
[38] Keren Zhou, Mark Krentel, and John Mellor-Crummey. 2020. A tool for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 25th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. 415–416.
[39] Keren Zhou, Mark W Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *Proceedings of the 34th ACM Int'l Conference on Supercomputing*. 1–12.
[40] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, Dejan Grubisic, and John Mellor-Crummey. 2021. An automated tool for analysis and tuning of gpu-accelerated code in hpc applications. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 854–865.
[41] Keren Zhou, Xiaozhu Meng, Ryuichi Sai, and John Mellor-Crummey. 2021. GPA: A GPU Performance Advisor Based on Instruction Sampling. In *2021 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO)*. 115–125. https://doi.org/10.1109/CGO51591.2021.9370339