An Automated Tool for Analysis and Tuning of GPU-accelerated Code in HPC Applications

Keren Zhou, *Member, IEEE,* Xiaozhu Meng, Ryuichi Sai, Dejan Grubisic, and John Mellor-Crummey

Abstract—The US Department of Energy's fastest supercomputers and forthcoming exascale systems employ Graphics Processing Units (GPUs) to increase the computational performance of compute nodes. However, the complexity of GPU architectures makes tailoring sophisticated applications to achieve high performance on GPU-accelerated systems a major challenge. At best, prior performance tools for GPU code only provide coarse-grained tuning advice at the kernel level. In this paper, we describe GPA, a performance advisor that suggests potential code optimizations at a hierarchy of levels, including individual lines, loops, and functions. To gather the fine-grained measurements needed to produce such insights, GPA uses instruction sampling and binary instrumentation to monitor execution of GPU code. At the time of this writing, GPU instruction sampling is only available on NVIDIA GPUs. To understand performance losses, GPA uses data flow analysis to approximately attribute measured instruction stalls back to their causes. GPA then analyzes patterns of stalls using information about a program's structure and the GPU architecture to identify optimization strategies that address inefficiencies observed. GPA then employs detailed performance models to estimate the potential speedup that each optimization might provide. Experiments with benchmarks and applications show that GPA provides useful advice for tuning GPU code. We applied GPA to analyze and tune a collection of codes on NVIDIA V100 and A100 GPUs. GPA suggested optimizations that it estimates will accelerate performance across the set of codes by a geometric mean of 1.21×. Applying these optimizations suggested by GPA accelerated these codes by a geometric mean of 1.19×.

Index Terms—High performance computing, Performance analysis, Parallel programming, Parallel architectures

1 Introduction

Over the last decade, supercomputers with compute nodes accelerated by Graphics Processing Units (GPUs) have become commonplace. GPU-accelerated designs are popular because GPUs can compute much faster and more efficiently than general purpose processors. For instance, GPUs provide more than 95% of the compute power on each node of OLCF's Summit [1]. Forthcoming exascale systems being developed by the US Department of Energy (DOE) will all employ GPU-accelerated compute nodes as well. For that reason, tools for tuning code performance on GPUs are an urgent need.

Tuning applications for GPU-accelerated compute nodes is a major challenge. GPU architectures are complex and hardware mechanisms for performance monitoring provide only limited information. In this paper, we focus on the challenge of automatically identifying inefficiencies in GPU-accelerated applications that are associated with data movement and computation on GPUs and advising application developers how to ameliorate such inefficiencies.

Prior research on GPU performance tools focused on identifying and analyzing hot GPU code. GPU profilers [2], [3], [4], [5], [6], [7], [8] trace activities and monitor hardware counters to provide performance feedback about GPU code but do not analyze bottlenecks and suggest optimizations. Alternatively, GPU performance tools based on instrumentation [9], [10], [11], [12], [13], [14] collect detailed information to provide low-level performance insights. Shortcomings of these tools are that they can only identify problems amenable to measurement with instrumentation. Furthermore, instrumentation-based tools don't suggest optimizations to ameliorate the problems, and they don't provide an estimate of the improvement one can expect by addressing a problem.

Instruction-based sampling [15], [16], [17] is a useful technique for fine-grained performance measurement and analysis on modern processors. In 2015, NVIDIA added support for instruction-based sampling to their GPUs [18], which they call *PC sampling*. Each instruction sample has a stall reason associated with it if the sampled instruction is stalled. At the time of this writing, hardware support for instruction-level measurement of GPU performance is only available on NVIDIA GPUs. This will change. Intel is developing hardware support for instruction-level performance measurement that will become available in a future generation of its GPUs.[1] In addition, recent commits in AMD's ROCm GitHub repository[2,3] indicate emerging support for PC sampling in their GPU software stack.

While some GPU performance tools [2], [4], [5], [19], [20] employ PC sampling to associate instruction costs with source code, they do not analyze stall reasons to identify why they occur or how to avoid them. A tool that analyzes a program's inefficiencies and suggests optimizations to avoid them would relieve developers of the burden of figuring out how to tune a code. Listing 1 shows a code snippet distilled from a hot loop in *BerkeleyGW* [21]—an exascale GW approximation code that uses OpenMP to offload com-

• *K. Zhou, X. Meng, R. Sai, D. Grubisic, and J. Mellor-Crummey are with the Computer Science Department, Rice University, Houston, TX, 77054. E-mail: {keren.zhou, xiaozhu.meng, ryuichi, dejan.grubisic, johnmc}@rice.edu*

1. Intel has approved public release of this information.
2. https://github.com/ROCm-Developer-Tools/roctracer/blob/amd-master/inc/ext/prof_protocol.h#L75
3. https://github.com/ROCm-Developer-Tools/roctracer/blob/amd-master/test/tool/tracer_tool.cpp#L655

TABLE 1: Fields in the `@P0 LDG.32 R0, [R2]` instruction.

Wait Cycles	Wait Mask	Write Barrier	Read Barrier	Predicate	Opcode	Modifier	Destination Operands	Source Operands
4	B0	B1		P0	LDG	32	R0	R2, R3

```
1 do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
2   wdiff = wx_array_t(n1_loc, iw) - wtilde
3   delw = wtilde / wdiff
4   ...
5   matngmatmgp = conjg(aqsmtemp_local(n1_loc,my_igp)) *
6     aqsntemp(ig, n1_loc)
7 end do
```

Listing 1: A hot loop in the *BerkeleyGW* code.

putation to GPUs. GPU profilers such as Nsight-Compute report high execution dependency stalls in this loop but provide neither information about the cause of the stalls nor candidate optimizations to reduce the stalls.

To address the challenge of understanding performance problems in GPU code and how to ameliorate them, we built GPA [22]—a GPU performance analysis advisor that analyzes instruction samples to guide performance optimization on NVIDIA GPUs. GPA attributes stalls to their causes, matches patterns of inefficiency with optimization strategies, and estimates the potential speedup for each applicable optimization. For the code in Listing 1, GPA attributes stalls to a device function that performs slow complex number division, and associates the device function with its call site on Line 3. To avoid using the costly device function, GPA suggests replacing the division with a multiplication by its reciprocal[4] and estimates a 1.18× speedup, which achieves close to the 1.24× speedup the optimization delivers.

While instruction sampling provides stall information with low overhead, it has several limitations. First, a sampled instruction is issued but not executed when its predicate is false. Moreover, instruction samples lack information about memory access efficiency and branch divergence. To overcome the above limitations, we also equipped GPA to collect performance metrics using binary instrumentation. GPA combines metrics from instruction sampling and instrumentation to yield a comprehensive performance report.

This paper describes the design and implementation of GPA—a tool designed to help analyze and optimize the performance of GPU kernels—as well as our experiences applying GPA to analyze and optimize GPU-accelerated applications. Tools with GPA's capabilities will be essential to simplify the task of tuning applications for GPU-accelerated supercomputers, including the DOE's forthcoming exascale platforms. GPA employs the following novel components to analyze, attribute, and understand the reasons behind stalls observed using instruction-based performance measurement:

- an instruction blamer which attributes stalls to their root causes by analyzing each instruction's dependency, stall reasons, and executed count,
- a hybrid-mode that considers observed control flow paths to improve the accuracy of instruction blaming,
- performance optimizers, which correlate inefficiency patterns with optimization suggestions based on control flow, program structure, and architecture features, and
- performance estimators, which estimate speedups for each optimizer by modeling GPU execution using instruction samples.

The next section provides background for understanding the design of GPA. Section 3 describes GPA's workflow and its implementation. Section 4 evaluates the utility of GPA's optimization suggestions for codes on both NVIDIA V100 and A100 GPUs. Section 5 discusses our experiences using GPA to study three large GPU codes. Section 6 reviews related work and distinguishes GPA. Section 7 discusses limitations of GPA and opportunities for future work.

2 Background

To provide background for understanding the design of GPA, this section describes GPU instructions and GPU performance metrics.

2.1 GPU Instructions

To understand and attribute performance losses in GPU kernels, GPA analyzes a program's GPU machine instructions. GPA currently targets NVIDIA Volta, Turing, and Ampere GPUs. Table 1 shows the fields of an `LDG` (load from global memory) instruction. Each GPU instruction has an opcode and several operands. GPU instructions also contain three special fields—wait cycles, predicate, and barriers. Wait cycles indicate how many cycles the GPU should wait before issuing an instruction. A predicate determines if an issued instruction will execute. If its predicate is false, an instruction issues but does not occupy any execution unit. Barriers enforce instruction ordering. For variable latency instructions, such as memory instructions and long latency arithmetic instructions, a read/write barrier is set at the source instruction, and a wait mask is set in the destination instruction. In the recent GPU architectures, barrier instructions (e.g., `LDGDEPBAR`) can be used to decouple barrier dependency from register dependency [23], yielding more flexible instruction scheduling and better latency hiding. GPA analyzes instruction operands [24] including regular registers, uniform data path registers, predicate registers, barriers, and immediate values.

2.2 Performance Metrics

Since Maxwell, NVIDIA GPUs support instruction sampling. Tools can collect instruction samples using NVIDIA's CUPTI API [18], [25]. Each streaming multiprocessor (SM) in a GPU records instruction samples for its warp schedulers in a round-robin fashion. Each instruction sample is either a *latency* sample or an *active* sample. A latency sample is recorded if all warps on a scheduler are stalled; an active sample is recorded if at least one warp is not stalled. A

4. One can simplify division by a complex number by multiplying the numerator and denominator by the complex conjugate of the denominator.

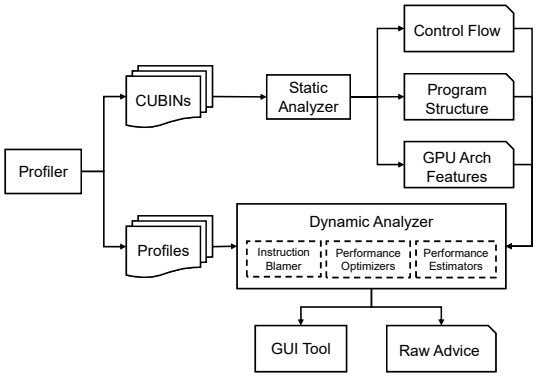


Fig. 1: GPA's Components.

stall reason, such as memory dependency, is associated with each latency sample. We call samples with a stall reason *stall samples* or *stalls* in the rest of the paper.

In addition to instruction sampling, CUPTI also provides interfaces to instrument GPU binaries to collect performance metrics that can't be measured using instruction sampling, such as executed instructions and memory accesses, As mentioned in Section 1, predicated instructions are not always executed. Using instrumentation, we collect the number of executed instructions and issued instructions for every program counter (PC), and define *instruction efficiency* as the ratio of issued instructions to executed instructions. If threads within a warp do not access aligned and coalesced global memory addresses, multiple memory transactions are generated, causing memory throttling. Similarly, if threads within a warp access the same bank on the shared memory, multiple shared memory transactions are generated. We collect the actual number of memory transactions and the theoretical number of memory transactions as if addresses are coalesced and don't conflict to check if memory access patterns should be adjusted. For each memory instruction, we define *memory efficiency* as the ratio of theoretical memory transactions to actual memory transactions.

3 GPA's Design

Section 3.1 introduces GPA's workflow. Section 3.2 describes how GPA attributes stalls to their causes. Section 3.3 describes GPA's performance optimizers. Section 3.4 presents models for estimating the benefit of optimizations.

3.1 Overview

Figure 1 shows a high-level overview of GPA's components. GPA uses a profiler to collect GPU instruction profiles and record GPU binaries for postmortem analysis. GPA's static analyzer analyzes GPU binaries to dump the control flow and program structure of each GPU function and gather information about the architecture (e.g., instruction latency). Then, GPA's dynamic analyzer takes in the instruction profiles and static analysis results to generate optimization suggestions using the following steps. First, the *instruction blamer* attributes instruction stalls to their causes. It also calculates instruction efficiency and memory efficiency for each instruction. Second, each *performance optimizer* employs rules to match stalls based on program structure, instruction efficiencies, and memory efficiencies of hot program regions. *Performance estimators* estimate each optimizer's speedup based the matched stalls. Finally, GPA outputs top recommended optimization suggestions to a raw report. A *Graphical User Interface (GUI)* associates stalls with individual lines and loops in GPU kernels.

GPA supports both *sampling* and *hybrid* modes. In *sampling* mode, GPA's profiler only collects instruction samples in instruction profiles. In sampling mode, the instruction blamer considers every issued instruction as executed when attributing stalls and does not calculate efficiency metrics. As a result, optimizations that rely on efficiency metrics are not available in the *sampling* mode. In *hybrid* mode, GPA's profiler uses one pass to collect instruction samples and a few more passes to collect instrumentation-based performance metrics. In hybrid mode, the instruction blamer assesses which instructions are executed to improve stall attribution, and all GPU optimizers are used to match stalls with optimization strategies. Section 4.3 compares GPA's end-to-end overhead for sampling and hybrid modes.

GPA is an advisor tool that automates performance profiling and analysis. To prepare an application for study by GPA, a developer needs to specify appropriate flags so that the compiler will include line mapping information in the generated executable. GPA needs line mapping information for meaningful performance reports. GPA's performance advice report for an application contains optimization suggestions ranked by their estimated speedups in text format. One can start with GPA's sampling mode to profile and analyze an application with a representative input. If the output advice does not contain insightful suggestions to guide performance optimization, one can switch to hybrid mode, which enables more optimizers but with higher profiling and analysis overhead. Currently, GPA's GUI associates instruction stalls on each GPU kernel's source lines, loops, and functions. In the future, we plan to integrate optimization suggestions into the GUI.

3.2 Blaming Stalls on Instructions

GPA analyzes instruction dependencies to blame stalls on their source instructions. To do so, GPA first uses backward slicing to analyze dependencies among instructions in the control flow of each GPU function. Next, GPA builds an instruction dependency graph in which each node represents an instruction, annotated with its performance metrics, and each edge indicates a def-use relation. Then, GPA prunes edges that are unlikely to cause stalls using several heuristic rules. Finally, GPA apportions each instruction's stalls to its sources based on each source's issued/executed instructions and the length its dependency edge. In the following sections, we separately describe the instruction blamer's sampling mode and hybrid mode.

3.2.1 Sampling Mode

Backward slicing: GPA uses intra-function backward slicing [26] on GPU instructions to find each instruction's dependency sources. Typically, GPA stops when it encounters the first dependency source along a path because transitive dependency sources are unlikely to cause stalls. However, when an instruction dependency source along a control flow

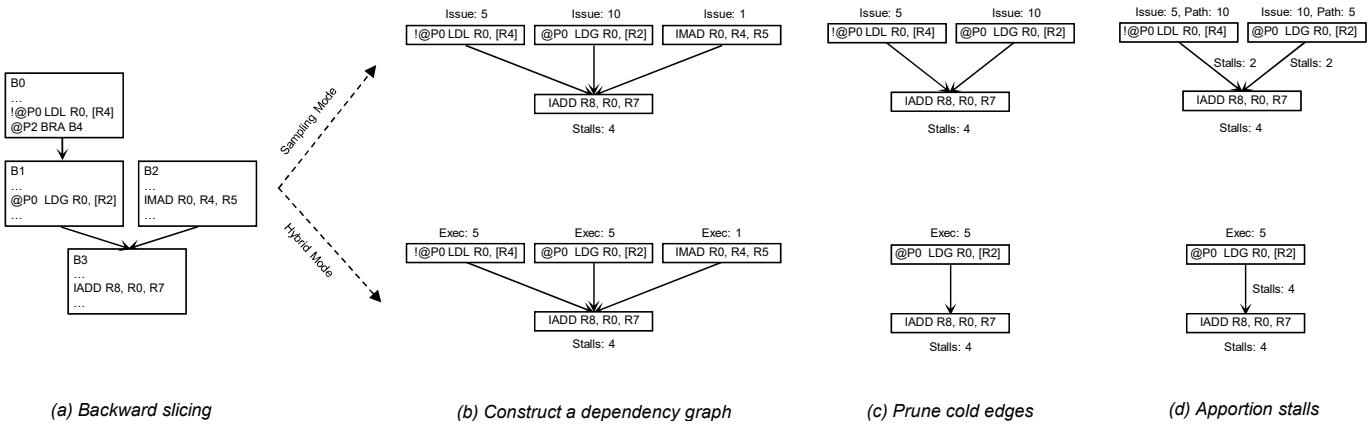


Fig. 2: An example of blaming stalls on the `IADD` instruction.

path is predicated, backward slicing continues along the path until the union of predicates along the path covers all conditions. Consider Figure 2 (a) as an example. To find all possible dependency sources for the stalls of the `IADD` instruction in `B3`, slicing doesn't stop at the `LDG` instruction in `B1` because it is not executed if predicate `P0` is false. Slicing continues to find the `LDL` instruction in `B0` along the path to cover all conditions.

In practice, GPU binaries for some applications contain a large number of functions and blocks, which can make backward slicing costly. For example, NAMD3 [27] contains 68 GPU binaries, of which the largest is 89MB with 4492 functions. Using a single CPU thread, GPA takes 12 hours to analyze these binaries. To accelerate backward slicing, we enhanced GPA to perform slicing for each basic block in parallel to reduce the slicing time for NAMD3 to 30 minutes using 32 CPU threads, achieving a 24× speedup.

Dependency graph creation: After computing static def-use chains for a GPU function's instructions, GPA creates a dynamic instruction dependency graph. GPA creates edges between a stalled instruction and its dependency sources that have any issued instruction sample. GPA assigns every dependency source one sample if none of them has been sampled as an issued instruction.

Cold edge pruning: The instruction dependency graph GPA computes has many "cold edges" that are unlikely to cause stalls. GPA uses the following heuristics rules to prune cold edges from the dependency graph.

1) **Opcode based pruning**. Attribute memory dependency stalls only to memory instructions and memory barrier instructions. Attribute synchronization dependency stalls only to synchronization instructions.
2) **Dominator based pruning**. For an edge e from node i to node j, remove the edge from the dependency graph if there is a non-predicated instruction k that uses the same operands that i defines and j uses, and k appears in every path from i to j in the control flow graph, because we would have observed stalls at k rather than j if i caused any stalls.
3) **Instruction latency based pruning**. For an edge e from node i to node j, prune it from the dependency graph if the number of instructions in every path from i to j in the control flow graph is greater than the $latency$ of i.

Opcode based pruning removes the edge from `IMAD` to the `IADD` in the sampling mode of Figure 2(c) because `IMAD` is an arithmetic instruction that does not cause memory dependency stalls.

GPA uses micro-benchmarks [28] to measure the latency of fixed latency instructions. For variable latency instructions, such as memory load/store, GPA uses their upper bound to perform conservative pruning.

Stall attribution: After pruning cold edges, some nodes in the dependency graph may still have multiple incoming edges. GPA blames stalls on the source of incoming edges based on the following two rules:

1) With more issued instruction samples, blame more stalls on the dependency source.
2) With a longer path, blame less stalls on the dependency source. If an instruction i has multiple paths to instruction j in the control flow graph, GPA calculates the average length of all paths to represent the length of edge e between node i and node j in the dependency graph.

Equation 1 describes how GPA apportions stalls of an observed instruction (S_j) on each dependency source (S_i), where $\mathcal{R}_i^{issue}$ and $\mathcal{R}_i^{path}$ are the ratios of each incoming node i calculated by heuristics (1) and (2) accordingly.

$$S_i = \frac{\mathcal{R}_i^{path} \times \mathcal{R}_i^{issue}}{\sum_{k \in incoming(j)} \mathcal{R}_k^{path} \times \mathcal{R}_k^{issue}} \times S_j \qquad (1)$$

3.2.2 Hybrid Mode

GPA's hybrid mode considers information about executed instructions to blame stalls on their dependency sources more precisely. In hybrid mode, GPA employs the same *backward slicing* strategy used for sampling mode. However, *dependency graph creation* builds an edge from node i to node j only if executed instructions have been observed for i. In hybrid mode, GPA employs an additional rule for *cold edge pruning*: it removes a path if any branch in the path has 1.0 *instruction efficiency* (always jump) and its next block is a fall through block, or 0.0 *instruction efficiency* (always fall

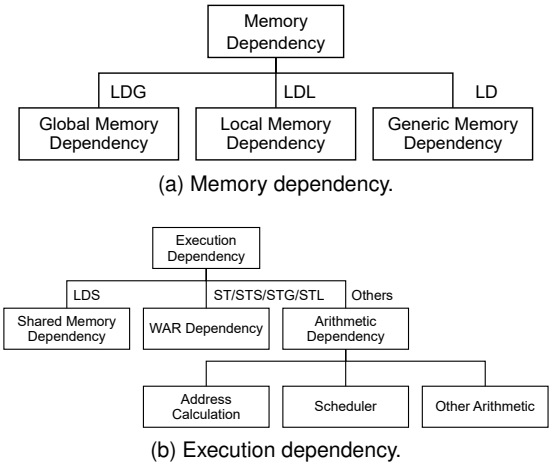


Fig. 3: Classification of detailed dependency stall reasons.

through) and its next block is a jump block. If there is no path from i to j after pruning, GPA removes this edge from the dependency graph. Consider Figure 2 (c) as an example, GPA removes the edge from `LDL` to `IADD` because the `BRA` after `LDL` has 1.0 instruction efficiency, indicating that no path contains `B0` to `B1` as a sub-path.

Finally, in the *stall attribution* phase, GPA calculates the ratios using executed instructions instead of issued instructions to apportion stalls (Equation 2).

$$S_i = \frac{\mathcal{R}_i^{path} \times \mathcal{R}_i^{exec}}{\sum_{k \in incoming(j)} \mathcal{R}_k^{path} \times \mathcal{R}_k^{exec}} \times S_j \tag{2}$$

3.2.3 Stall classification

Using information about dependency sources, GPA further classifies memory dependency and execution dependency stalls as shown in Figure 3. GPA categorizes memory dependencies as global memory, local memory, or generic memory according to the source instruction's opcode. Execution dependency is more sophisticated. GPA first classifies an execution dependency as shared memory, arithmetic, or write-after-read (WAR) based on opcode. WAR dependency stalls occur when a *use* instruction writes a register that is read by a variable latency *def* instruction. It is worth noting that some execution dependency stalls may not have any dependency source. If such stalls occur on a memory instruction with indirect memory access, GPA classifies the stalls as address calculation stalls. For GPU architectures after Volta, modifiers such as `X4` and `X8` are extensively used to calculate address at memory instructions to reduce register usage. In other cases, GPA attributes stalls to the observed instruction and classifies them as scheduler stalls. As shown in Section 4.1, scheduler stalls are small after instruction blaming.

3.3 Suggesting Optimizations

Each performance optimizer in GPA encodes rules to match stalls that could be reduced by its optimization. Table 2 presents all performance optimizers available in GPA. Optimizers annotated with $\mathcal{H}$ are only available in hybrid mode as they rely on instruction efficiency and/or memory efficiency. At a high level, optimizers either improve parallelism or improve code efficiency. Parallelism optimizers check if a GPU kernel lacks sufficient parallelism and recommends adjusting the number of blocks and threads. For example, if occupancy is bounded by the number of threads, the *Thread Increase* optimizer suggests increasing the number of threads per block to hide latency. Code optimizers check if certain patterns of stalls exist in particular program regions. GPA further divides code optimizers into stall elimination and latency hiding optimizers. Stall elimination optimizers suggest eliminating or replacing costly instructions; while latency hiding optimizers recommend rearrange instructions to hide latency.

We elaborate the workflow of *Branch Elimination* and *Asynchronous Memory Copy* optimizers. The branch elimination optimizer checks branches instructions that are determined (always executed/not executed) and aggregates all stalls from the source block to the destination block because it assumes instructions in the two blocks can be rearranged to reduce stalls. Asynchronous memory copy instructions (`LDGSTS`, `LDGMEMBAR`), which load value from global memory directly to shared memory, are new to NVIDIA's Ampere GPU. In earlier GPUs, one must employ a global memory read instruction followed by a shared memory store instruction to load values to shared memory; such a strategy causes large register overhead and exposes global memory latency. The asynchronous memory copy optimizer finds instruction dependency pairs from global memory reads to shared memory stores and sums their stalls. The optimizer suggests using asynchronous memory copy instructions to replace synchronous memory copy instructions to increase the distance between load and use. Moreover, the optimizer also checks the stalls attributed to asynchronous memory copy instructions if any and suggests increasing the distance between memory transaction commit (`LDGMEMBAR`) and its barrier (`DEPBAR`) if latency is not hidden as expected.

3.4 Estimating Speedups

Performance optimizers match optimization suggestions with stalls, but do not provide information about which methods have a better effect considering the given measurement data, program structure, and the GPU architecture. GPA addresses this issue with performance estimators that estimate the speedup of each optimizer based on their matched stalls. GPA employs both code optimization estimators and parallelism optimization estimators. Suggestions from optimizers with top estimated speedups are output to a performance advice report.

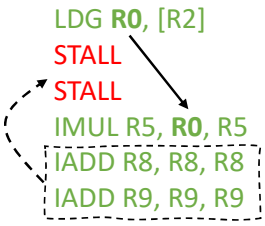
3.4.1 Code Optimization Estimators

We first estimate the effect of stall elimination optimizers. Suppose a GPU kernel has a total of T instruction samples and the matched samples for an optimizer is M. Stall elimination optimizers assume that all matched stalls, including matched active samples and latency samples, can be eliminated by optimizing the code. We derive Equation 3 to estimate the speedup of stall elimination optimizers $\mathcal{S}^e$.

$$\mathcal{S}^e = \frac{T}{T - M} \tag{3}$$

TABLE 2: A description of GPU optimizers in GPA.

Code Optimizers		
	Stall Elimination	
	Register Increase	Match memory dependency stalls of local memory read/write instructions
	Strength Reduction	Match execution dependency stalls of long latency arithmetic instructions
	Function Split	Match instruction fetch stalls
	Fast Math	Match stalls in CUDA math functions
	Warp Balance	Match warp synchronization stalls
	Global Memory Transaction Reduction	Match global memory throttling stalls
	Shared Memory Transaction Reduction	Match shared memory stalls
	Indirect Memory Access Reduction	Match execution dependency stalls on indirect memory access instructions
$\mathcal{H}$	Global Memory Access Adjustment	Match low memory efficiency caused global memory throttling stalls
$\mathcal{H}$	Shared Memory Access Adjustment	Match low memory efficiency caused shared memory stalls
	Latency Hiding	
	Loop Unrolling	Match global memory and execution dependency stalls in loops
	Code Reordering	Match global memory and execution dependency stalls
	Function Inlining	Match stalls in device functions and their call sites
	Asynchronous Memory Copy	Match stalls on copies from global memory to shared memory
$\mathcal{H}$	Branch Elimination	Match stalls in a determined branch
Parallelism Optimizers		
	Block Increase	Match if the number of blocks is less than the number of SMs
	Thread Increase	Match if occupancy is limited by the number of threads per block



Green code indicates active samples, and red stalls represent latency samples; latency hiding optimizers consider the effect of moving the code in the dashed box to fill the stall slots.

Fig. 4: A mental model for latency hiding optimizers.

Unlike stall elimination optimizers, latency hiding optimizers reduce latency samples. We use Equation 4 to estimate the speedup of latency hiding optimizers $\mathcal{S}^h$, where M^L is the number of matched latency samples.

$$\mathcal{S}^h = \frac{T}{T - M^L} \tag{4}$$

Equation 4 supposes all latency samples can be eliminated by rearranging instructions. However, in practice, not all M^L can be eliminated. Figure 4 explains the mental model of latency hiding optimizations, where the upper bound of the optimizations is bounded by both the matched latency samples and active samples. We derive Equation 5 to refine the estimate of $\mathcal{S}^h$, where A denotes the total number of active samples.

$$\mathcal{S}^h = \frac{T}{T - Min(A, M^L)} \tag{5}$$

Some optimizations such as loop unrolling only rearrange code for a specific scope. Therefore, only active samples within the scope can be used to reduce the scope's latency samples. Based on this limitation, we refine the upper bound of latency hiding optimization with Equation 6 to consider optimization scopes representing loops and functions. $\mathcal{S}_l^h$ indicates the speedup for a specific scope l, and M_l^L is the matched latency samples for a scope l.

$$\mathcal{S}_l^h = \frac{T}{T - Min(\sum_{l' \in nested(l)} A_{l'}, M_l^L)} \tag{6}$$

Suppose we have two loops *loop1* and *loop2*, where *loop1* is nested in *loop2*, the speedup of *loop2* is bounded by the total number of active samples of *loop2* and *loop1* according to Equation 6.

3.4.2 *Parallelism Optimization Estimator*

Unlike code optimizers, parallelism optimizers adjust the number of blocks and threads to improve performance. To estimate the speedup of parallelism optimizers, we consider each warp scheduler's change of active warps–$\mathcal{C}_W$ (Equation 7) and change of issue rate—$\mathcal{C}_\mathcal{I}$ (Equation 8) .

For example, if the number of blocks is increased, each warp scheduler is assigned less active warps so that $\mathcal{C}_W$ is less than one. At the same time, as the number of threads is reduced, the rate that a warp scheduler is issuing is reduced, and $\mathcal{C}_\mathcal{I}$ is less than one.

$$\mathcal{C}_W = \frac{W_{new}}{W} \tag{7}$$

$$\mathcal{C}_\mathcal{I} = \frac{\mathcal{I}_{new}}{\mathcal{I}} \tag{8}$$

By assuming each warp scheduler has the same issue rate, we derive Equation 9 and Equation 10 to calculate issue rate $\mathcal{I}$ and $\mathcal{I}_{new}$ respectively, where R_I is the ratio of issued samples among all samples. A warp scheduler is issuing if at least one warp assigned to the scheduler is ready to issue an instruction.

$$\mathcal{I} = 1 - (1 - R_I)^W \tag{9}$$

$$\mathcal{I}_{new} = 1 - (1 - R_I)^{W_{new}} \tag{10}$$

$$\mathcal{S}^p = \frac{1}{\mathcal{C}_W} \times \mathcal{C}_\mathcal{I} \times f \tag{11}$$

We estimate the speedup of parallelism optimizations ($\mathcal{S}^p$) based on $\mathcal{C}_W$ and $\mathcal{C}_\mathcal{I}$ using Equation 11, where f is a factor that varies between optimizers. Some optimizers assume there are no pipeline, memory throttle, and select stalls if we reduce the number of active warps per block to a certain number (e.g., less than the number of schedulers per SM).

TABLE 3: Experimental platforms

Node	Hardware	Software
Local Intel	2×E5-2695v4 CPUs 1×NVDIA V100 GPUs	GCC-8.3.0 CUDA-11.1.105
Local AMD	2×AMD EPYC 7402 CPUs 1×NVIDIA A100 GPU	GCC-8.3.1 CUDA-11.1.105
NERSC cori-gpu	2×Intel E5-2698 v3 CPUs 8×NVIDIA V100 GPUs	GCC-8.3.0 CUDA-11.1.105

TABLE 4: Achieved speedups averaged among five runs. We improved each code according to the suggestion provided by GPA. Estimate error is computed by $\frac{|Estimated\ Speedup - Achieved\ Speedup|}{Achieved\ Speedup} \times 100\%$. The difficulty column shows the complexity of applying the corresponding optimization to the code. The bottom part of the table is the first evaluation of GPA's new optimizations. $\mathcal{S}$ and $\mathcal{H}$ denote metrics measured by the sampling mode and the hybrid mode accordingly. Zeros are treated as 0.01 in geomean calculation.

Application	Kernel	Optimization	Difficulty	Original	Achieved Speedup	Estimated Speedup ($\mathcal{S}/\mathcal{H}$)	Error ($\mathcal{S}/\mathcal{H}$)
rodinia/backprop	bpnn_layerforward_CUDA	Warp Balance	Easy	25.23±0.14us	1.15±0.00×	1.15×/1.18×	0%/3%
rodinia/backprop	bpnn_layerforward_CUDA	Strength Reduction	Easy	22.03±0.25us	1.13±0.01×	1.12×/1.08×	1%/4%
rodinia/bfs	Kernel	Loop Unrolling	Easy	498.55±1.21us	1.06±0.00×	1.26×/1.26×	19%/19%
rodinia/b+tree	findRangeK	Code Reorder	Medium	55.98±0.24us	1.14±0.01×	1.24×/1.19×	**9%/4%**
rodinia/cfd	cuda_compute_flux	Code Reorder	Medium	147.87±2.26ms	1.40±0.02×	1.28×/1.28×	9%/9%
rodinia/gaussian	Fan2	Thread Increase	Easy	93.87±0.00ms	3.69±0.00×	3.85×/3.86×	4%/5%
rodinia/heartwall	kernel	Loop Unrolling	Easy	161.22±0.00ms	1.21±0.00×	1.26×/1.25×	4%/3%
rodinia/hotspot	calculate_temp	Strength Reduction	Easy	19.59±0.01us	1.09±0.01×	1.09×/1.11×	0%/2%
rodinia/huffman	vlc_encode_kernel_sm64huff	Warp Balance	Medium	169.85±0.28us	1.05±0.00×	1.21×/1.09×	15%/13%
rodinia/kmeans	kmeansPoint	Loop Unrolling	Easy	602.94±0.01us	1.02±0.01×	1.03×/1.03×	1%/1%
rodinia/lavaMD	kernel_gpu_cuda	Loop Unrolling	Easy	4.41±0.05ms	1.01±0.01×	1.16×/1.15×	15%/14%
rodinia/lud	lud_diagonal	Code Reorder	Medium	218.33±0.11us	1.36±0.00×	1.15×/1.23×	**15%/10%**
rodinia/myocyte	solver_2	Fast Math	Easy	386.36±1.41ms	1.13±0.00×	1.13×/1.13×	0%/0%
rodinia/myocyte	solver_2	Function Splitting	Medium	342.60±0.99ms	1.00±0.01×	1.02×/1.02×	2%/2%
rodinia/nw	needle_cuda_shared_1	Warp Balance	Medium	1.33±0.01us	1.11±0.01×	1.12×/1.12×	1%/1%
rodinia/particlefilter	likelihood_kernel	Block Increase	Easy	3.62±0.03ms	1.95±0.01×	1.80×/1.81×	8%/7%
rodinia/streamcluster	kernel_compute_cost	Block Increase	Easy	29.97±0.33ms	1.30±0.01×	1.35×/1.33×	4%/2%
rodinia/sradv1	reduce	Warp Balance	Medium	3.08±0.00ms	1.05±0.00×	1.05×/1.04×	0%/1%
rodinia/pathfinder	dynproc_kernel	Code Reorder	Easy	151.84±0.57us	1.06±0.01×	1.35×/1.30×	**27%/23%**
Quicksilver	CycleTracking_Kernel	Function Inlining	Medium	38.87±0.45s	1.13±0.01×	1.1×/1.07×	3%/5%
Quicksilver	CycleTracking_Kernel	Register Increase	Hard	34.52±0.63s	1.07±0.00×	1.08×/1.07×	1%/0%
ExaTENSOR	tensor_transpose	Strength Reduction	Easy	3.24±0.02ms	1.25±0.01×	1.08×/1.10×	14%/12%
ExaTENSOR	tensor_transpose	Global Memory Transaction Reduction	Easy	2.60±0.01ms	1.02±0.00×	1.01×/1.04×	1%/2%
PeleC	react_state	Block Increase	Easy	117.76±0.94s	1.21±0.01×	1.27×/1.29×	5%/7%
Minimod	target_pml_3d	Fast Math	Easy	100.10±1.10ms	1.06±0.00×	1.12×/1.13×	7%/6%
Minimod	target_pml_3d	Code Reorder	Medium	95.00±1.01ms	1.00±0.01×	1.00×/1.00×	0%/0%
rodinia/myocyte	solver_2	Global Memory Access Adjustment	Hard	340.97±0.17ms	1.22±0.09×	*NA*/1.31×	*NA*/7%
ExaTENSOR	tensor_transpose	Asynchronous Memory Copy	Hard	2.54±0.01ms	1.24±0.01×	1.36×/1.35×	9%/10%
ExaTENSOR	tensor_transpose	Indirect Memory Access Reduction	Easy	2.05±0.02ms	1.03±0.01×	1.05×/1.06×	2%/3%
Minimod	target_pml_3d	Branch Elimination	Easy	94.98±1.03ms	1.07±0.01×	*NA*/1.06×	*NA*/1%
Minimod	target_pml_25d	Warp Balance	Hard	0.27±0.01s	1.26±0.01×	1.25×/1.26×	1%/0%
NAMD3	nonbondedForceKernel	Register Increase	Medium	19.22±0.05s	1.09±0.00×	1.05×/1.05×	4%/4%
BerkeleyGW	mtxel	Fast Math	Easy	100.31±0.42s	1.24±0.01×	1.18×/1.18×	5%/5%
geomean					1.19×	1.21×/1.19×	4.0%/4.0%

4 Evaluation

We evaluated GPA's utility on three x86_64 systems. The hardware and software specification of the three systems are shown in Table 3. We used GPA to analyze a broad range of benchmarks and applications, including the Rodinia GPU benchmarks [29], QuickSilver [30], ExaTensor [31], PeleC [32], Minimod [33], NAMD3 [27], and BerkeleyGW [21]. Rodinia is a well-known benchmark suite that consists of many representative GPU computation patterns. PeleC, NAMD, and BerkeleyGW are codes being refined for simulations on exascale platforms.

Table 4 shows speedups we achieved on an A100 GPU by applying optimizations suggested by GPA. One exception is the BerkeleyGW example, which we only profiled on cori-gpu using a V100 GPU becaue of its complicated package dependencies. For each benchmark, we focused on the most costly kernels and implemented some of the top five optimizations recommended by GPA. We achieved a geometric mean of 1.19× speedup averaged across optimizations with individual speedups ranging from 1.00× to 3.69×. The geometric mean of the gap between achieved speedups and estimated speedups is as low as 4.0%. We used the same optimizations in the top part of Table 4 as our previous study [22] on a V100 GPU and evaluated their effects on an A100 GPU to demonstrate the utility of GPA's suggested optimizations on a different GPU architecture. The bottom section in Table 4 is the first evaluation of GPA's new optimizations. In the rest of this section, we discuss estimated and achieved speedups by applying one of GPA's suggested optimizations, GPA's stall attribution, overhead comparison between GPA's sampling and hybrid modes, and GPA's optimization workflow.

4.1 Stall Attribution

We evaluated GPA's instruction blamer using *single dependency coverage* and *scheduler stalls ratio*. A node is a *single dependency* node if each of its incoming edges represents a different stall dependency. *single path coverage* is computed as the ratio of *single path node* over the total number of nodes in a dependency graph. Figure 5 shows that most benchmarks have high single dependency coverage, demonstrating the effectiveness of GPA's pruning rules. We also observed that GPA's hybrid mode improves the single path coverage of some benchmarks such as *lud* and *PeleC* by pruning with additional rules.

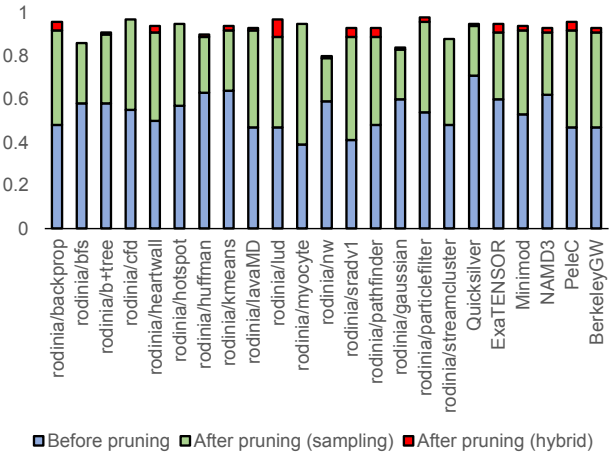


Fig. 5: Single dependency coverage (The higher the better).

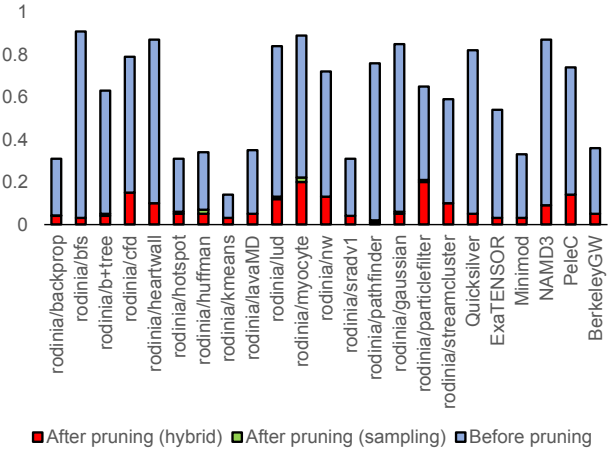


Fig. 6: Scheduler stall ratio (The lower the better).

Figure 6 shows the *scheduler stalls ratio* of each benchmark. As mentioned in Section 3, some arithmetic stalls are attributed to the observed instruction and categorized as scheduler stalls if the instruction has no dependent edge after pruning. There are many stall causes for nodes without any dependent edge. For example, we observed that the compiler may annotate some instructions with wait cycles larger than needed to satisfy the dependencies for the instructions themselves to satisfy latency requirements for later instructions. Nevertheless, scheduler stalls only account for 6% stalls on average. We observe similar scheduler stall ratios using the hybrid mode and the sampling mode.

4.2 Speedups

Table 4 compares the estimated speedups reported by GPA's hybrid mode and sampling mode. Most benchmarks show similar estimated speedups in two modes. We note that the hybrid mode outperforms the sampling mode from the following aspects. First, the hybrid mode offers optimization suggestions that are not available (*NA*) in the sampling mode. For example, based on the global memory access adjustment optimizer, we achieved a 1.22× additional speedup for *myocyte* by coalescing the global memory accesses to

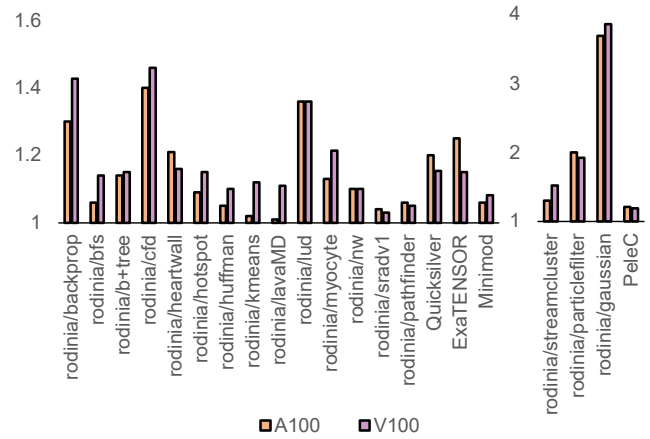


Fig. 7: Achieved speedups comparison between A100 and V100

an array. Second, the hybrid mode employs extra pruning and attribution rules to attribute and apportion stalls, which increase the estimate accuracy of code reordering related optimizations, including *b+tree*, *lud*, and *pathfinder*.

Figure 7 compares the achieved speedups in the top section of Table 4 on A100 and V100 GPUs. Most optimizations achieve similar speedup on A100 and V100, demonstrating the generality of GPA's performance insights. Several optimizations behave differently on the two GPUs. For the *kmeans* benchmark, GPA estimates a lower speedup on A100 than V100. For the *lavaMD* benchmark, GPA's estimated speedup is 14% higher than we achieved. We observed that the execution on A100 incurs more execution dependency stalls than V100. Using GPA's GUI tool, we checked stalls on each source line and found that the major stall difference is concentrated on two lines where the compiler schedules instructions differently on the two architectures.

4.3 Overhead

We measured GPA's runtime, static analysis, and dynamic analysis overhead. Figure 8 shows the slowdown of GPA's hybrid mode comparing to its sampling mode at runtime. NVIDIA's PC sampling mechanism currently serializes concurrent kernels, increasing measurement time. For most applications, the sampling mode introduces overhead less than 10×. However, it introduces 54× overhead to the BerkeleyGW benchmark on cori-gpu. GPA's hybrid mode has 2–14× slowdown comparing to its sampling mode because it instruments GPU binaries to collect other performance metrics. For two ExaScale applications, PeleC and BerkeleyGW, the hybrid mode introduces 92× and 296× overhead correspondingly. While GPA's dynamic analysis is fast, GPA's static analysis could introduce significant overhead for applications with many large GPU binaries in the assembly instruction decoding and backward slicing processes.

4.4 Optimization Workflow

Guided by GPA's performance advice report, one can gain performance insight, modify a program's source code, and quickly assess the benefits of an optimization. The time

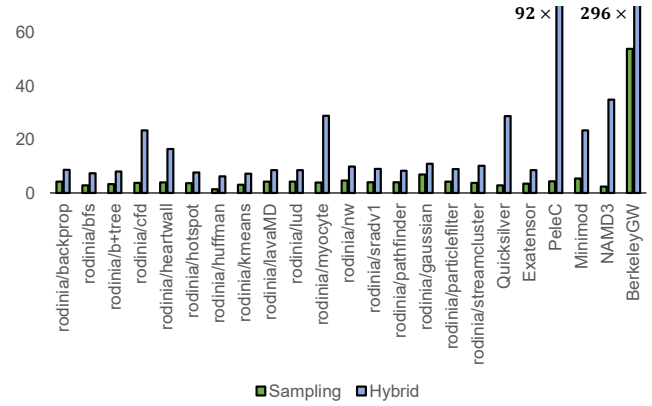


Fig. 8: Slowdown with GPA's hybrid mode and sampling mode.

needed for the whole optimization process typically varies between a few minutes to an hour, depending on the difficulty of optimizations, as noted in Table 4. GPA may overestimate speedups for many reasons, including the limited scope for reordering code associated with a data dependency, unawareness of loop transformations performed by the compiler, and work imbalance. For instance, in the *bfs* benchmark, GPA suggests unrolling a loop to improve its performance. We observed the estimated speedup is 19% higher than the achieved speedup because the workload is highly unbalanced across threads so that loop unrolling only benefits a few threads. There are also cases in which overestimation is caused by data dependency. For example, in the *huffman* benchmark, much of the stalls are synchronization dependency stalls. We replaced some `__syncthreads` by `__syncwarp` but not all of them. The `pathfinder` benchmark also has a data dependency related problem. We can only reorder instructions within a limited scope before synchronizations take place.

5 Case Studies

The codes studied in the bottom part of Table 4 are described below:

- ExaTensor [31] is a tensor algebra library implemented on NVIDIA GPUs. We studied its tensor transpose kernel with a six-dimensional tensor.
- Minimod [33] is a stencil benchmark for seismic modeling. We analyzed its performance with a grid size of 100^3.
- NAMD3 [27] is a high performance parallel molecular dynamics code for simulating large biomolecular systems. We used its *alanin* input configuration running 9×10^5 steps.
- BerkeleyGW [21] is a GPU-accelerated code that uses the GW method to calculate the quasiparticle properties and the optical responses of a large variety of materials. We studied it's matrix elements calculation (*mtxel*) kernel.

The case studies were performed on a system with an A100 GPU. When compiling these codes for the GPU, we used the options `-lineinfo -O3` to generate line mapping information helpful for GPA's performance reports.

5.1 ExaTensor

Figure 9 shows the top performance suggestion from GPA's performance report for the ExaTensor benchmark. In GPA's performance reports, GPU kernels are ordered by their execution time. For each kernel, the report lists several performance optimization suggestions ranked by their estimated speedups. For each suggestion, GPA offers *hints* about code changes to improve performance and lists hotspots where those hints could be applied. For each hotspot, GPA supplies *program context*, *importance*, and *speedup* information. Using the hybrid mode, GPA also provides the memory efficiency and instruction efficiency about the stall source. Program context provides information about the locations of the stalls and their dependency sources. The importance metric indicates the percentage of stalls this optimizer matches, and the speedup metric indicates the estimated speedup after applying suggested code changes.

In Figure 9, GPA estimates that applying asynchronous memory copy optimization may improve code performance by 1.35×. GPA also provides the locations of the matched hotspot for this optimization. We show this hotspot's assembly instructions in Figure 10, in which a load from global memory (`LDG`) instruction is immediately followed by a store to shared memory (`STS`) instruction. Following GPA's hints, we used a `LDGSTS` instruction to replace this `LDG` and `STS` pair and overlap the calculation of transposed index for each element with data transfer. This single optimization achieves a 1.24× speedup, which is 10% lower than the estimated 1.35× speedup because the scope for code rearrangement is limited by synchronization. On NVIDIA GPU architectures prior to the A100, which lack support for asynchronous memory copy, it is difficult to optimize this code. As illustrated in Figure 10, if we increase the distance between `LDG` and `STS`, we have to keep register `R2` alive. Since a large tile is loaded into shared memory, separating each pair of dependent instructions would require many live registers and significantly increase register pressure.

We used GPA again to analyze optimized code. This time GPA suggests eliminating high execution dependency stalls caused by indirect memory access at constant memory load instructions that read the dimensions of the input tensor. We can assign the number of dimensions as a template parameter for this kernel so that the compiler can treat it as a constant and eliminate indirect memory access. This optimization achieves another 1.03× speedup, which is close to the estimated 1.06× speedup.

5.2 Minimod

Minimod's `target_pml_3d` kernel loads all values from a three dimensional tensor at once and performs a high-order stencil computation with a halo size of 4. Every thread with an index within a halo size of a tile boundary reads values from a halo area. GPA's branch elimination optimizer suggests some branches are always true. Because each GPU block has $32 \times 4 \times 4$ threads, each thread's Y and Z dimensions are always less than the halo size away from a tile boundary. To eliminate stalls caused by these branch conditions, we used the block size as a template parameter and let the compiler elide the branch conditions

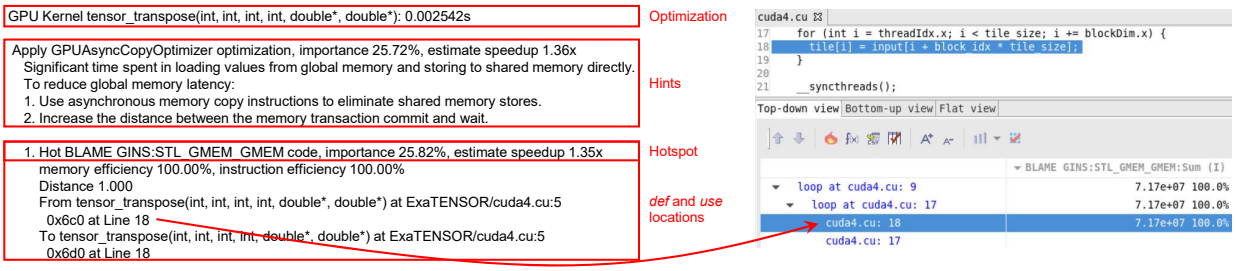


Fig. 9: The highest ranked optimization suggestion in GPA's performance report for the ExaTensor benchmark and the corresponding hotspot in GPA's GUI.

```
/*06c0*/   LDG.E.64 R2, [R2.64]
/*06d0*/   STS.64 [R5.X8], R2
```

Fig. 10: Code pattern that matches asynchronous memory copy optimization.

at compilation time. This optimization achieves a 1.07× speedup, which matches GPA's 1.06× estimated speedup.

Next, we analyzed Minimod's `target_pml_25d` kernel which iteratively loads a piece of data from global memory to shared memory and performs a high-order stencil computation. GPA's top warp balance optimization highlights significant synchronization dependency stalls at a `__syncthreads` invocation in a loop and suggests eliminating unnecessary synchronizations. Two synchronizations are needed to prevent data race if a single tile is used. If the amount of shared memory use is small, we can use two tiles to read and update shared memory separately to eliminate one synchronization in each iteration. We achieved a 1.26× speedup by employing this optimization, which is the same as GPA's estimated speedup.

5.3 NAMD3

We used GPA to analyze the most costly `nonbondedForceKernel` function in NAMD3. Based on GPA's report, we observed that NAMD3 is a highly optimized application. GPA's report only suggests two significant optimizations—code reorder and register increase. We first followed the hints of the code reorder optimizer but failed to obtain a non-trivial speedup with GPA since the data dependency is highly intricate. Then we checked the register increase optimizer which identifies the locations of local memory loads and stores caused by register spills, and suggests a 1.05× speedup by eliminating these stalls. This kernel uses `__launch_bounds__` to imply the minimum number of concurrent blocks on each SM of a GPU and thus enforces a very low register limit per thread. However, for the input we studied, while the number of kernel invocations is large, the number of blocks used by each kernel is small. By using a special `__launch_bounds__` constraint to allow more registers when the number of blocks is small, we achieved a 1.09× speedup.

5.4 BerkeleyGW

In the Introduction, we mentioned that GPA suggests optimzing a costly device function. Following the guidance, BerkeleyGW developers changed the code and achieved a 1.24× speedup. GPA's loop unrolling optimizer also suggests unroll the core loop on Line 1 to improve performance. We confirmed that this loop is not automatically unrolled because of dependency among iterations and notified the developers such an potential optimization. Next, we profiled BerkeleyGW again using GPA's hybrid mode. This time GPA's global memory access adjustment optimizer indicates that the `aqsmtemp_local` array on Line 5 is accessed with low efficiency and estimates a 1.09× speedup by coalescing memory accesses. The estimated speedup matches the performance difference between this *mxtel* kernel's CUDA version and OpenMP version, and the CUDA code loads `aqsmtemp_local` to shared memory using colaseced memory read to ameliorate this problem. Unfortunately, shared memory is declared implicitly OpenMP Target and is not widely supported in existing compilers.

6 Related Work

Some prior GPU performance tools [2], [3], [4], [5], [6], [7], [19], [20] collect instruction samples for performance analysis. However, these tools only characterize performance bottlenecks at the kernel level but do not provide optimization suggestions for specific code regions. In contrast, GPA analyzes raw instruction samples along with metrics from instrumentation, matches stalls with inefficiency patterns to suggest optimizations, and estimates their benefits. Due to the lack of hardware support in publically-available GPUs, Intel's VTune [34] and AMD's ROCProfiler [35] don't use instruction sampling to analyze the performance of GPU kernels.

GPU vendors also provide instrumentation tools [10], [11], [12], [13] to facilitate fine-grained performance measurement and analysis. Based on these tools, researchers have used instrumentation to detect certain kinds of inefficiencies. Arafa et al. [36] collect a memory trace and build a performance prediction model for a GPU's cache hierarchy. GVProf [9] analyzes both temporal and spatial redundant value patterns in GPU-accelerated applications. Each of these tools analyze only a narrow class of problems and can't accurately assess their impact on performance due to the high overhead of instrumentation. In comparison, GPA employs instruction sampling to measure code hotness with minimal overhead and uses instrumentation in a separate pass to collect instruction execution information for comprehensive analysis.

Performance advisor tools that examine code quality and provide optimization suggestions have been extensively

studied on CPUs. PerfExpert [37] employs HPCToolkit [38] to collect CPU performance metrics with sampling. Like GPA, it combines the analysis of measurement data and system parameters to estimate performance upper-bounds. Built upon PerfExpert, AutoScope [39] ranks the optimization suggestions and outputs the most effective ones. Unlike the above tools that analyze profiles to derive performance insights, there also exist tools that analyze only static code patterns. MAQAO [40] performs static analysis of assembly code to offer optimization suggestions with much lower overhead than dynamic profiling based tools. CQA [41] is a loop-centric tool that employs a static model to analyze code quality and models instruction execution by emulating processor pipelines. Egeria [42] adopts natural language processing (NLP) to associate code transformation rules from vendor optimization guides with inefficiencies reported by profilers. Unlike Egeria, GPA attributes stalls to their root causes and outputs optimizations at a hierarchy of scopes at lines, loops, and functions.

7 Conclusions and Future Work

The rise of GPU-accelerated supercomputers, including forthcoming exascale systems, has created an urgent need to improve programming models, compilers, performance tools, and runtime systems to better support application development and tuning for such platforms. Our GPA tool bridges the gap between the discovery of GPU performance bottlenecks and the identification of effective optimizations. We show that GPA can identify a range of performance problems and then identify effective optimizations to ameliorate these problems. At present, the lack of hardware support for instruction-level performance measurement precludes retargeting GPA to Intel and AMD GPUs.

Currently, GPA has several limitations. First, GPA has high measurement overhead for some applications. CUDA 11.3 introduced a new PC sampling mode with much lower overhead. A prototype using this new sampling mode reduced measurement overhead for Laghos [43] from 21× to 7×. We plan to reduce the overhead of GPA's hybrid mode by implementing more efficient instrumentation callbacks. Second, while adding parallelism to backward slicing improves its speed, it also increases its memory footprint. At present, Dyninst [44] performs backward slicing by analyzing effects of individual instructions; it would be more space and time efficient to use basic-block summaries, where appropriate. Third, GPA does not support analysis of branch divergence and warp synchronization. We can leverage methods developed by others [45] to enhance GPA so that it can diagnose such issues.

In our experiments, we found that compilers are sometimes myopic in optimizing instruction schedules. NVIDIA's `nvcc` compiler only reorders instructions within a limited scope based on heuristics. In some cases, reordering a few program statements improves performance. Furthermore, high level programming models, such as OpenMP, though portable, lack critical features to match the performance of native programming models, including explicit support for using shared memory, asynchronous memory copies, and fast math instructions. We believe that GPA provides a solid foundation for feedback-based performance advisor/optimizers on GPU-accelerated platforms that can be adapted as new capabilites for instruction-based performance measurement emerge. As future work, we are interested in exploring how GPA's insights can be used to drive profile-guided optimization of GPU code.

Acknowledgments

This research was supported in part by the Exascale Computing Project (17-SC-20-SC)—a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, Lawrence Livermore National Laboratory (Subcontract B639429), and an ExxonMobil Graduate Fellowship. We thank Total E&P Research & Technology USA, LLC for allowing us to use Minimod as a case study. We thank Helen He (NERSC), Mauro Del Ben (LBNL), and William Huhn (ANL) for their help analyzing BerkeleyGW. We also thank the anonymous reviewers of this paper for their feedback, which helped us significantly improve the paper.

References

[1] S. S. Vazhkudai *et al.*, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *SC18: Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 661–672.

[2] NVIDIA Corporation, *Profiler User's Guide DU-05982-001_v11.2*, December 2020. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf

[3] ——. (2020) NVIDIA Nsight Systems. [Accessed Jan. 1, 2021]. [Online]. Available: https://developer.nvidia.com/nsight-systems

[4] ——. (2020) NVIDIA Nsight Compute. [Accessed Jan. 1, 2021]. [Online]. Available: https://developer.nvidia.com/nsight-compute

[5] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of GPU-accelerated applications," in *Proc. of the 34th ACM Intl. Conf. on Supercomputing*, ser. ICS '20. New York, NY, USA: ACM, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392752

[6] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The Intl. Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[7] D. Mey *et al.*, "Score-P: A unified performance measurement system for petascale applications," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Springer Berlin Heidelberg, 2012, pp. 85–97.

[8] C. January *et al.*, "Allinea MAP: Adding energy and OpenMP profiling without increasing overhead," in *Tools for High Performance Computing 2014*. Springer, 2015, pp. 25–35.

[9] K. Zhou *et al.*, "GVProf: A value profiler for GPU-based clusters," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[10] M. Kambadur *et al.*, "Fast computational GPU design with GT-Pin," in *2015 IEEE Intl. Symp. on Workload Characterization*. IEEE, 2015, pp. 76–86.

[11] M. Stephenson *et al.*, "Flexible software profiling of GPU architectures," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 185–197.

[12] O. Villa *et al.*, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. of the 52nd Annual IEEE/ACM Intl. Symp. on Microarchitecture*. ACM, 2019, pp. 372–383.

[13] NVIDIA Corporation, *NVIDIA Compute Sanitizer DA-05679-001_v11.2*, December 2020. [Online]. Available: https://docs.nvidia.com/cuda/pdf/Compute_Sanitizer.pdf

[14] D. Shen *et al.*, "CUDAAdvisor: LLVM-based runtime profiling for modern GPUs," in *Proc. of the 2018 Intl. Symp. on Code Generation and Optimization*, 2018, pp. 214–227.

[15] J. Dean *et al.*, "ProfileMe: Hardware support for instruction-level profiling on out-of-order processors," in *Proc. of 30th Annual Intl. Symp. on Microarchitecture*. IEEE, 1997, pp. 292–302.

[16] P. J. Drongowski, "Instruction-based sampling: A new performance analysis technique for AMD family 10h processors," *Advanced Micro Devices*, 2007.

[17] IBM Corporation, *POWER9 Performance Monitor Unit User's Guide, version 1.2*, November 2018. [Online]. Available: https://ibm.ent.box.com/s/8kh0orsr8sg32zb6zmq1d7zz6hud3f8j

[18] NVIDIA Corporation. (2019) PC sampling. [Accessed Jan. 1, 2021]. [Online]. Available: https://docs.nvidia.com/cupti/Cupti/r_main.html#r_pc_sampling

[19] H. Zhang and J. Hollingsworth, "Understanding the performance of GPGPU applications from a data-centric view," in *2019 IEEE/ACM Intl. Workshop on Programming and Performance Visualization Tools (ProTools)*, Nov 2019, pp. 1–8.

[20] H. Zhang, "Data-centric performance measurement and mapping for highly parallel programming models," Ph.D. dissertation, University of Maryland—College Park, 2018.

[21] J. Deslippe *et al.*, "BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures," *Computer Physics Communications*, vol. 183, no. 6, pp. 1269–1289, 2012.

[22] K. Zhou *et al.*, "GPA: A GPU Performance Advisor Based on Instruction Sampling," in *Intl. Symp. on Code Generation and Optimization (CGO'21)*. IEEE, 2021.

[23] R. Ohannessian Jr *et al.*, "System, method, and computer program product for implementing software-based scoreboarding," Apr. 4 2017, US Patent 9,612,836.

[24] NVIDIA Corporation. (2021) CUDA binary utilities. [Accessed Feb 1, 2021]. [Online]. Available: {https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref}

[25] ——, *CUPTI User's Guide DA-05679-001_v11.2*, 2020, https://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf.

[26] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *1997 Proc. Intl. Conf. on Software Maintenance*. IEEE, 1997, pp. 188–195.

[27] J. C. Phillips *et al.*, "NAMD: Biomolecular simulation on thousands of processors," in *SC'02: Proc. of the 2002 ACM/IEEE Conf. on Supercomputing*. IEEE, 2002, pp. 36–36.

[28] Z. Jia *et al.*, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[29] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[30] Lawrence Livermore National Laboratory. (2020) Quicksilver. [Accessed Jan. 1, 2021]. [Online]. Available: https://github.com/LLNL/Quicksilver

[31] D. I. Lyakh. (2020) ExaTENSOR. [Accessed Jan. 1, 2021]. [Online]. Available: https://iadac.github.io/projects/

[32] National Renewable Energy Laboratory. (2020) PeleC. [Accessed Jan. 1, 2021]. [Online]. Available: https://github.com/AMReX-Combustion/PeleC

[33] J. Meng *et al.*, "Minimod: A finite difference solver for seismic modeling," *arXiv preprint arXiv:2007.06048v1*, 2020.

[34] J. Reinders, "VTune performance analyzer essentials," *Intel Press*, 2005.

[35] Advanced Micro Devices, Inc. AMD ROCm ROCProfiler. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html

[36] Y. Arafa *et al.*, "Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles," in *Proc. of the 34th ACM Intl. Conf. on Supercomputing*, ser. ICS '20. New York, NY, USA: ACM, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392761

[37] M. Burtscher *et al.*, "PerfExpert: An easy-to-use performance diagnosis tool for HPC applications," in *SC'10: Proc. of the 2010 ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–11.

[38] L. Adhianto *et al.*, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[39] O. A. Sopeju *et al.*, "AutoScope: Automatic suggestions for code optimizations using PerfExpert," *Evaluation*, 2011.

[40] L. Djoudi *et al.*, "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, vol. 200, no. 5. Citeseer, 2005.

[41] A. S. Charif-Rubial *et al.*, "CQA: A code quality analyzer tool at binary level," in *2014 21st Intl. Conf. on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–10.

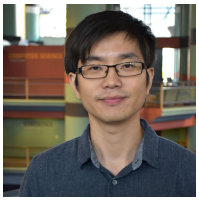
[42] H. Guan *et al.*, "Egeria: a framework for automatic synthesis of HPC advising tools through multi-layered natural language processing," in *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[43] V. A. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM Journal on Scientific Computing*, vol. 34, no. 5, pp. B606–B641, 2012.

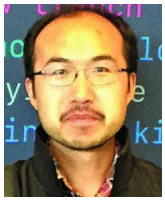
[44] U. of Wisconsin-Madison. Dyninst. [Accessed Jan. 1, 2021]. [Online]. Available: https://github.com/dyninst/dyninst

[45] S. Damani *et al.*, "Speculative reconvergence for improved SIMT efficiency," in *Proc. of the 18th ACM/IEEE Intl. Symp. on Code Generation and Optimization*, 2020, pp. 121–132.

Keren Zhou received the MS degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2017. He is currently a PhD student student at Rice University, Houston, TX. His research focuses on performance tools for HPC applications. Zhou is a recipient of a 2020 ACM-IEEE CS George Michael Memorial HPC Fellowship.



Xiaozhu Meng received the PhD in Computer Science in 2018 at University of Wisconsin-Madison, Madison, WI. He is a Research Scientist in the Department of Computer Science at Rice University, Houston TX. His research focuses on binary code analysis and instrumentation, and its applications in high performance computing and computer security.



Ryuichi Sai received the MS degree in computer science from University of Houston in 2012. He is currently a PhD student at Rice University, Houston, TX. His research focuses on programming languages, compiler technologies, and their applications in high performance computing.

Dejan Grubisic received the MS degree in computer science from the University of Novi Sad, Serbia in 2019. He is currently a PhD student at Rice University, Houston, TX. His research focuses on measuring and analysis of the performance of HPC applications. Grubisic is a recipient of the Pollard fellowship in 2019.

John Mellor-Crummey received the PhD degree in computer science from the University of Rochester, Rochester, NY, in 1989. He is a Professor of Computer Science at Rice University, Houston, TX. His research focuses on software technology for high performance parallel computing, including compilers, runtime systems, tools, and synchronization. Mellor-Crummey is a co-recipient of the 2006 Dijkstra Prize in Distributed Computing and a Fellow of the ACM.