Tools for top-down performance analysis of GPU-accelerated applications

Keren Zhou
Department of Computer Science
Rice University
Houston, TX, USA
keren.zhou@rice.edu

Mark W. Krentel
Department of Computer Science
Rice University
Houston, TX, USA
krentel@rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University
Houston, TX, USA
johnmc@rice.edu

ABSTRACT

This paper describes extensions to Rice University's HPCToolkit performance tools to support measurement and analysis of GPU-accelerated applications. To help developers understand the performance of accelerated applications as a whole, HPCToolkit's measurement and analysis tools attribute metrics to calling contexts that span both CPUs and GPUs. To measure GPU-accelerated applications efficiently, HPCToolkit employs a novel wait-free data structure to coordinate monitoring and attribution of GPU performance metrics. To help developers understand the performance of complex GPU code generated from high-level programming models, HPCToolkit's `hpcprof` constructs sophisticated approximations of call path profiles for GPU computations. To support fine-grain analysis and tuning, HPCToolkit attributes GPU performance metrics to source lines and loops. Also, HPCToolkit uses GPU PC samples to derive and attribute a collection of useful GPU performance metrics. We illustrate HPCToolkit's new capabilities for analyzing GPU- accelerated applications with three case studies.

CCS CONCEPTS

• **General and reference** → **Measurement**; **Metrics**; • **Computer systems organization** → **Parallel architectures**; • **Theory of computation** → *Concurrent algorithms.*

KEYWORDS

GPU, Profiler, Wait-free, Calling Context Tree, HPC, Roofline

ACM Reference Format:
Keren Zhou, Mark W. Krentel, and John Mellor-Crummey. 2020. Tools for top-down performance analysis of GPU-accelerated applications. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392752

1 INTRODUCTION

Today, five of the ten most powerful supercomputers employ nodes accelerated with GPUs [36]. Application developers need performance tools to help them harness the full power of such systems. Pinpointing performance problems in GPU-accelerated programs can be difficult as it often requires detailed analysis that combines measurements of a program's execution, information passed to a GPU kernel in the calling context where it was invoked, characteristics of compiler-generated GPU code, and GPU hardware features. Furthermore, while the use of higher-level programming models such as RAJA [18], Kokkos [10], OpenMP [29], and OpenACC [28] can simplify the development of accelerated applications, they can increase the difficulty of tuning kernels for high performance by separating developers from many key details, such as what GPU code is generated and how it will be executed.

Performance tools for GPU-accelerated programs typically employ two kinds of views to identify performance bottlenecks. A *trace view* shows a series of events that happen over time on each process, thread, and GPU stream. By examining a trace view, one can determine if these events interleave and overlap over time as expected; however, from a trace view it can be difficult to determine whether or why any kernel is inefficient. A *profile view* collapses out the time dimension and correlates performance metrics with program contexts. Using a profile view, one can determine how much time, effort, or inefficiency is associated with each context.

Most GPU performance tools [1, 5, 6, 8, 19, 20, 22, 30, 32, 34] only provide a trace view for GPU API calls and/or a flat profile view. With the adoption of template-based models for GPU programming such as RAJA and Kokkos, it has become essential to understand the context where a template for GPU computation is instantiated and invoked. For example, a template-based dot product kernel in the RAJA performance suite [23] yields 25 different GPU functions that implement the computation. To our knowledge, only HPCToolkit [4] and CUDABlamer [44, 45] collect a *call path* profile view that associates costs for GPU kernel invocations with the source-level calling contexts where they are invoked.[1]

Since 2015, NVIDIA GPUs support collection of fine-grain measurement and attribution of GPU performance using PC sampling. While tools such as Nsight-compute [5], nvprof [8], TAU [24], and Allinea Map [19] can use PC sampling to associate metrics with individual source lines for GPU code, they lack the ability to associate costs with loop nests within GPU functions. Loop-level agregation of metrics is often important to understand the performance of scientific programs.

Prior performance tools for GPU-accelerated code employ multiple runs to compute a rich set of performance metrics because GPUs provide only a small number of hardware counters. To assess the performance of a small tensor transpose kernel, Nsight-compute

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392752

[1] Unlike conventional profiles, *call path* profiles associate costs with the full calling context in which they are incurred rather than just the procedure and often source line.

runs multiple passes to collect all of its metrics. Such an approach would yield unacceptably large overhead when applied to a long-running application that employs many kernels.

To address these shortcomings, we extended HPCToolkit [1]—an open-source suite of tools for call path profiling of CPU applications—to support measurement and analysis of GPU-accelerated codes. After our enhancements, HPCToolkit has the following capabilities:

- It constructs heterogeneous calling contexts that span CPU and GPU. To our knowledge, no other tool has this capability. A highlight of this feature is a technique to construct an approximate calling context tree for GPU computations from flat GPU PC samples without instrumentation.
- It employs a novel and fast wait-free data structure for inter-thread communication necessary to correlate performance metrics for each asynchronous GPU kernel invocation with its CPU calling context.
- It derives a rich set of metrics from PC samples gathered during a single execution.

We demonstrate the effectiveness of HPCToolkit's support for analyzing GPU-accelerated codes with three case studies. Our results show that HPCToolkit's profile views provide insights into performance problems in compiler-generated code, resource usage, synchronization, parallelism, and memory access patterns.

This rest of the paper is organized as follows. Section 2 introduces the workflow of HPCToolkit and its extensions for performance analysis of GPU-accelerated code. Section 3 describes the design of HPCToolkit's measurement subsystem for collecting GPU performance metrics. Section 4 discusses how HPCToolkit attributes performance metrics to GPU-accelerated programs. Section 5 describes methods for analyzing GPU code efficiency. Section 6 studies three codes and demonstrates the effectiveness of HPCToolkit for analyzing performance problems in GPU-accelerated applications. Section 7 describes related work. Finally, Section 8 summarizes our conclusions and briefly mentions plans for future work.

2 DESIGN OVERVIEW

Figure 1 shows HPCToolkit's workflow and our changes to analyze code offloaded to NVIDIA GPUs. We extended HPCToolkit's `hpcrun` to collect GPU performance measurements. Our measurement substrate uses callbacks from NVIDIA's CUPTI API [26] for NVIDIA GPU profiling. Our tool can be applied to general GPU programs that employ one or more of OpenMP, OpenACC, and CUDA. To associate GPU APIs with OpenMP offloading operations, HPCToolkit uses our implementation of the OpenMP Tool Interface for NVIDIA GPUs in LLVM OpenMP. For PGI's OpenACC [28], no special support is needed. As a GPU-accelerated program executes, our extensions to `hpcrun` record CUDA binaries (CUBINs) loaded into NVIDIA GPUs for later analysis.

HPCToolkit's `hpcstruct` employs Dyninst [27] to analyze CPU binaries to recover information about procedures, source lines, loop nests, and inlined functions. We augmented `hpcstruct` to analyze GPU binaries. First, `hpcstruct` employs NVIDIA's `nvdiasm` to dump instructions and control flow graphs (CFGs). Second, it parses these CFGs and injects them into Dyninst for loop analysis. Third, it identifies GPU instruction opcodes and modifiers and then

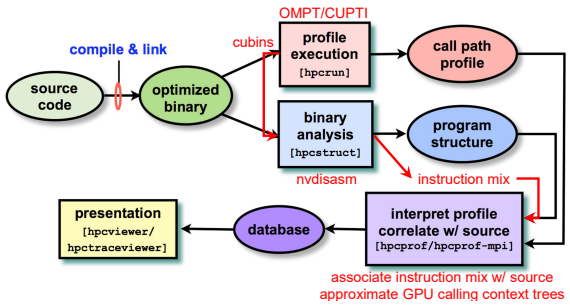

Figure 1: HPCToolkit's workflow [1], with our adaptations for analysis of GPU-accelerated applications shown in red.

maps them to instruction classes. Fourth, it attributes instructions to source code with the help of line maps in GPU binaries.

Finally, we extended HPCToolkit's `hpcprof` to calculate GPU instruction mix for program contexts and approximate GPU calling context trees. Our extensions to HPCToolkit provide a complete workflow to analyze the performance of GPU-accelerated applications.

3 PERFORMANCE MEASUREMENT

When monitoring a CPU-accelerated program, in addition to *application threads*, which offload work onto GPUs, there is a background *GPU monitor thread*, that receives measurement data from the GPUs. When an application thread T calls a GPU API, it uses call stack unwinding to recover its CPU calling context and creates a placeholder node P in the calling context. Then, thread T associates a correlation ID C with P and creates a *correlation* record to indicate that event C belongs to thread T. To avoid data races, only an application thread may attribute metrics to nodes in its calling context tree. Thus, when the GPU monitor thread needs to associate GPU metrics with C, it creates *measurement* records and communicates them to thread T for attribution to P. For an instruction sample, thread T creates a node with the instruction's PC and inserts the node as a child of P.

The next section describes how we enhanced HPCToolkit's measurement subsystem to efficiently unwind call stacks at every GPU API invocation. Section 3.2 describes a wait-free data structure used to communicate correlation records and measurement records between each application thread and the GPU monitor thread.

3.1 Fast CPU Call Stack Unwinding

To monitor program execution on CPUs, HPCToolkit uses asynchronous sampling. Typically, call stack unwinding is performed when an asynchronous profiling event (e.,g a HW counter overflow event or a timer interrupt), triggers a sample. However, for GPU-accelerated programs, HPCToolkit unwinds the call stack at every GPU API call. To monitor frequent GPU invocations and attribute their cost to calling contexts without significant overhead, unwinding must be fast. To address this challenge, we added three new mechanisms to HPCToolkit's `hpcrun` to accelerate unwinding.

First, we avoided unnecessary unwinding of stack frames. In programs that frequently offload operations to GPUs, temporally

Algorithm 1 A wait-free producer/consumer channel.

```
1:  struct Node
2:      Node *next; Record r
3:
4:  struct Channel
5:      Node *consumer_in                    ▷ a list of nodes for the consumer
6:      Node *producer_out                   ▷ a list of nodes from the producer
7:
8:  procedure PRODUCER_PUSH(Channel c, Node n) ▷ races with consumer_STEAL
9:      repeat
10:         n.next ← atomic_load(&c.producer_out)
11:     until atomic_compare_exchange(&c.producer_out, &n.next, n)
12:
13: procedure CONSUMER_STEAL(Channel c)      ▷ races with producer_PUSH
14:     c.consumer_in← atomic_exchange(&c.producer_out, NULL)
15:
16: function CONSUMER_POP(Channel c)         ▷ serial with consumer_STEAL
17:     first ← c.consumer_in
18:     if first is not NULL then c.consumer_in← first.next
19:     return first
```

Algorithm 2 Wait-free producer-consumer protocol

```
1:  struct BidirectionalChannel
2:      Channel produced, consumed
3:
4:  function PRODUCE(BidirectionalChannel b, Record r)
5:      if node ← consumer_POP(b.consumed) is NULL then
6:          consumer_STEAL(b.consumed)
7:          node ← consumer_POP(b.consumed)
8:      if node is NULL then node ← allocate()
9:      node.r ← r
10:     producer_PUSH(b.produced, node)
11:
12: function CONSUME(BidirectionalChannel b, CallBack fn)
13:     consumer_STEAL(b.produced)
14:     while node ← consumer_POP(b.produced) is not NULL do
15:         fn(node.r)
16:         producer_PUSH(b.consumed, node)
```

adjacent GPU operations often share a long common call path prefix. We identify the adjacent common call path prefix unwind by using trampolines [11] on x86_64 and mark bits [2] on Power. Once `hpcrun's` call stack unwinder reaches a stack frame in common with the previous unwind, it unwinds no further.

Second, we accelerated `hpcrun`'s lookup of unwind recipes. As a program executes, `hpcrun` computes and memoizes unwind recipes in a concurrent skip list. Prior to our modifications, a thread acquires a read lock and searches the skip list for each unwind step. To avoid lock contention and the expected $O(\log n)$ lookup cost in a skip list with n entries, each thread now employs a 1000-entry hashtable to memoize pointers to recipes found in the skip list. This reduces the expected cost of finding a recipe to $O(1)$ as we search the concurrent skip list only when a hashtable lookup fails.

Third, we eliminated redundant unwinding associated with CUDA API calls. Any CUDA driver API called by a CUDA runtime API shares its application calling context with the runtime API; thus, unwinding of the driver API can be skipped. Our three enhancements reduced overall measurement overhead by 3.5× for frequent GPU API calls by the Laghos code, described in Section 6.

3.2 Wait-free Communication

When attributing the performance of asynchronous kernel executions back to the application threads that initiated them, communication between a GPU monitor thread and application threads becomes frequent. Delaying any thread with locking or unbounded attempts of lock-free operations of would disrupt execution behavior. Moreover, for long-running applications, memory for data passed between threads must be reclaimed. We designed a data structure that supports wait-free communication of *measurement* and *correlation* records between the GPU monitor thread and each application thread.

3.2.1 A Wait-free Communication Channel. Algorithm 1 shows a point-to-point channel designed for producer/consumer communication. The channel consists of a shared linked list *producer_out* and a private linked list *consumer_in*. A *producer_PUSH* repeatedly tries to add a node at the head of *producer_out*, like a Treiber stack push [40]. A *consumer_STEAL* points *consumer_in* at a list of nodes atomically removed from *producer_out*. A *consumer_POP* extracts and returns an element from the head of *consumer_in*. The communication channel is *wait-free*: *consumer_POP* and *consumer_STEAL* execute only a few instructions; the `atomic_compare_exchange`[2] in *producer_PUSH* may fail at most once when a concurrent *consumer_STEAL* sets *producer_out* to NULL. Once *producer_out* is NULL, any additional *consumer_STEAL*s cannot change that. Thus, the next `atomic_load` of *producer_out* by *producer_PUSH* will yield NULL and the next `atomic_compare_exchange` in *producer_PUSH* will succeed. The linearization point of *producer_PUSH* is its successful `atomic_compare_exchange`. The linearization point of *consumer_STEAL* is its `atomic_exchange`. Note that the channel is not FIFO; one could make it FIFO by reversing the list stolen from *producer_out* prior to setting *consumer_in* in *consumer_STEAL*. This change would not compromise wait-freedom because the cost of reversing the list is bounded and is proportional to the list length.

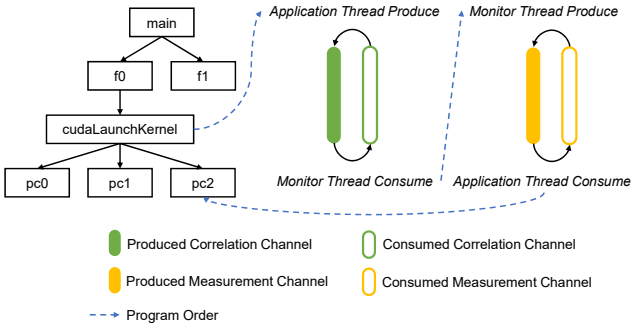

Figure 2: Interactions between the GPU monitor thread and application threads. pc0, pc1, and pc2 denote PC samples

3.2.2 A Wait-free Coordination Protocol. A bidirectional channel for producer-consumer communication consists of a pair of wait-free channels. Records flow from a producer to a consumer. The consumer ingests records transmitted and then returns them to the producer for reuse. Each application thread communicates with

[2]The primitive `atomic_compare_exchange(&mem, &oldval, newval)` will return `true` if the value of `mem` can be atomically changed from `oldval` to `newval`. If `mem` no longer has the value `oldval`, the primitive will return `false` and `oldval` will be updated with the value of `mem`.

the GPU monitor thread using a pair of bidirectional channels, as shown in Figure 2. Next, we describe the four operations in Figure 2.

Application Thread Produce. When a GPU API is invoked, an application thread calls Algorithm 2's *PRODUCE* to push a correlation record into a *produced* correlation channel. Before allocating a correlation record, Line 5 attempts to pop a node from the *consumed* correlation channel. If the *consumer_in* list in the *consumed* correlation channel is empty, Line 6 updates it by stealing nodes (if any) from the channel's *producer_out* list. If the *consumed* channel is empty, a new node is allocated at Line 8 to communicate the correlation record. The record *r* is entered in the empty node at Line 9. Finally, Line 10 pushes the node into the *produced* correlation channel.

Monitor Thread Consume. When the GPU monitor thread receives a buffer of records from GPUs, it calls *CONSUME* to steal all correlation records previously pushed by the application thread (Line 13) and processes them using a callback function (*fn*) at Line 15. Consumed records are pushed into the *consumed* correlation channel (Line 16) for reuse by *Application Thread Produce*.

Monitor Thread Produce. After the GPU monitor thread finishes *Monitor Thread Consume*, it adds measurement records to the *produced* measurement channel by calling *PRODUCE*.

Application Thread Consume. Before the next GPU API call, an application thread calls *CONSUME* to process measurement records pushed by *Monitor Thread Produce*. A callback function attributes metrics to the corresponding calling context at Line 15. As above, consumed records are returned to their producer for reuse.

An invariant at the end of each *CONSUME* call is that *produced.consumer_in* is empty, so the next *CONSUME* call may overwrite it without losing records. Each bidirectional channel is *wait-free* because Line 10 fails at most once if Line 13 executes concurrently, and Line 16 fails at most once when Line 6 executes concurrently.

Using wait-free channels to communicate correlation and measurement records between application threads and the GPU monitoring thread accounts for only 2% of the time of a monitored execution. When measuring GPU accelerated codes, HPCToolkit has comparable measurement overhead to `nvprof`; this is noteworthy because HPCToolkit attributes measurements to GPU kernel invocation calling contexts and `nvprof` does not.

4 PROGRAM STRUCTURE RECOVERY

HPCToolkit attributes performance metrics to individual PCs at runtime and aggregates metrics to lines and loops post-mortem based on program structure recovered with offline binary analysis. In comparison, TAU [34] and Nsight-compute [5] read `CUpti_ActivitySourceLocator` records at runtime to match PCs with lines, which only handle code generated from CUDA but not code generated from OpenMP. Our approach works for any statically-compiled language, including C, C++, and Fortran.

HPCToolkit's `hpcstruct` employs the Dyninst binary analysis framework [27] to analyze a binary's line map, symbol table, and machine code to recover information about procedures, source lines, loop nests, and inlined functions. We extended `hpcstruct` to analyze CUDA binaries (CUBINs).

A CUBIN is an ELF binary. Its form is unlike any CPU binary we have seen: each global function is in its own text segment at address 0. Also, GPU device function are sometimes "embedded" inside a global GPU function. To simplify the measurement and analysis of CUBINs, we (1) relocate each function to a unique address—its offset in the binary's section table, and (2) split overlapping GPU functions into disjoint address ranges.

There is no reliable API for decoding GPU instructions to enable `hpcstruct` to recover loop nests directly from GPU machine code. For that reason, we use NVIDIA's `nvdisasm` binary tool to analyze GPU machine code and dump a control flow graph for each function in a CUBIN. `nvdisasm`'s Control Flow Graphs (CFGs) are not suitable for analysis since basic blocks are merged, and out-going edges for call and branch instructions are not fully identified. To cope with this issue, we split superblocks into basic blocks, and link edges between sources and targets. We extended Dyninst to analyze CFGs that we glean from `nvdisasm` so that we can use Dyninst's ParseAPI to discover and analyze loops and their nesting.

When a CUBIN function contains an indirect control transfer, `nvdisasm` fails to dump a CFG. In such cases, *hpcstruct* skips analysis of the function's loops and produces a simpler program structure using only information from the function's line map.

5 PERFORMANCE ANALYSIS

This section describes a top-down approach to identify performance problems in GPU-accelerated codes. Section 5.1 introduces metrics to pinpoint bottlenecks in heterogeneous applications. Section 5.2 presents metrics to analyze GPU kernel execution in detail.

Currently, all existing GPU performance tools do not automate optimizations and thus are best suited for expert users. At best, Nsight-compute and nvprof provide high-level suggestions on GPU kernels. In contrast, our analysis methods pinpoint hotpots of hybrid GPU-CPU calling context in large-scale GPU-accelerated programs automatically. Our work here provides a solid foundation for a performance advisor that suggests program improvement strategies at every level.

5.1 Importance Metrics

A performance tool should focus user attention on the most expensive program contexts. Here we describe how to compute some useful derived metrics to identify such hotspots. Let C_i and G_i be the inclusive time spent in calling context i on CPUs and GPUs respectively. GPU computations are typically launched asynchronously, so C_i and G_i may overlap. Let P represent the calling context for the whole program. $I_{C,i}$ is the *importance* of CPU time spent in calling context i.

$$I_{C,i} = Max(\frac{C_i - G_i}{C_i}, 0) \times \frac{C_i}{C_P} \quad (1)$$

If a calling context takes more time on GPUs than CPUs, we consider the context unimportant and set $I_{C,i}$ to zero. Otherwise,

$\frac{C_i - G_i}{C_i}$ indicates the importance of CPUs for this calling context and we multiply it by $\frac{C_i}{C_P}$ to normalize it to the whole execution.

If we find the bottleneck of an application is not on CPUs, we compute $I_{G,i}$ as shown in Equation 2 to assess the importance of GPU time in calling context i by computing the fraction of the total GPU computation time spent in context i:

$$I_{G,i} = \frac{G_i}{G_P} \tag{2}$$

Let $G_{i,j}$ represent the GPU time spent in calling context i performing a particular GPU operation type j, e.g. kernel execution. We refine Equation 2 to consider operation types in Equation 3.

$$I_{G,i,j} = \frac{G_{i,j}}{G_P} \tag{3}$$

The metrics described by equations 1–3 are considered in sequence to identify performance bottlenecks. If a program has low $I_{C,P}$, we follow paths from the calling context tree root to GPU API invocations where $I_{G,i}$ is high and then identify the problematic GPU API type j at that spot. For example, if memory copies and synchronizations take a long time, we first inspect the most expensive memory copies and synchronizations instead of optimizing GPU kernels. Otherwise, we use analysis methods introduced in the next sections to analyze kernel performance.

5.2 Analysis Using PC Sampling

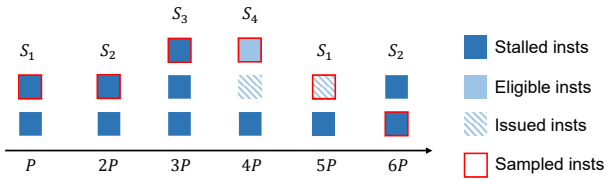

Figure 3: NVIDIA's GPU PC sampling example on an SM. $P - 6P$ represent six sample periods P cycles apart. $S_1 - S_4$ represent four schedulers on an SM.

5.2.1 PC Sampling. NVIDIA's GPUs have supported PC sampling since Maxwell [7]. Instruction samples are collected separately on each active streaming multiprocessor (SM) and merged in a buffer returned by NVIDIA's CUPTI API [26]. In each sampling period, one warp scheduler of each active SM samples the next instruction from one of its active warps. Sampling rotates through an SM's warp schedulers in a round robin fashion. When an instruction is sampled, its stall reason (if any) is recorded. If all warps on a scheduler are stalled when a sample is taken, the sample is marked as a latency sample, meaning no instruction will be issued by the warp scheduler in the next cycle. Figure 3 shows a PC sampling example on an SM with four schedulers. Among the six collected samples, four are latency samples, so the estimated stall ratio is 4/6.

5.2.2 GPU Calling Context Tree Reconstruction. The CUPTI API returns flat PC samples without any information about GPU call stacks. With complex code generated from higher-level GPU programming models, we need calling contexts on GPUs to understand the code and its performance. Currently, no API is available for efficiently unwinding call stacks on NVIDIA's GPUs. To address this issue, we designed a method to reconstruct approximate GPU calling contexts offline.

Algorithm 3 GPU Calling Context Tree Reconstruction

```
1:  function ConstructCallGraph(symbols, samples)
2:      G ← create an empty Graph
3:      for s in symbols do
4:          v ← create a vertex with address range [s.start, s.end]
5:          add v to G.V
6:      for v in G.V do
7:          for instruction i in address range [v.start, v.end] do
8:              i.weight ← 0
9:              for sample p in samples with p.address == i.address do
10:                 i.weight ← p.count
11:             if i is Call then
12:                 e ← create an edge
13:                 e ← (.weight = i.weight, .source = i.v, .target = i.target)
14:                 add e to e.source.outgoing
15:                 add e to e.target.incoming
16:                 add e to G.E
17:             v.weight ← v.weight + i.weight
18:     return G
19:
20: function PropagateCallGraph(G)
21:     q ← create a FIFO queue with {v ∈ G|v.weight > 0}
22:     while q is not Empty do
23:         v ← pop the first node from q
24:         caller ← False
25:         for e in v.incoming with e.weight > 0 do
26:             caller ← True
27:         if caller is False then
28:             for e in v.incoming do
29:                 if e.source.weight == 0 then
30:                     push e.source to the end of q
31:                 e.weight ← 1
32:                 e.source.weight ← e.source.weight + 1
33:     for v in G.V with v.weight == 0 do
34:         remove v from G
35:     return G
36:
37: function SplitCallGraph(G, root, factor)
38:     root.weight ← root.weight × factor
39:     for instruction i in address range [root.start, root.end] do
40:         i.weight ← i.weight × factor
41:     for e in root.outgoing do
42:         v ← clone e.target
43:         create a tree link between root and v
44:         sum ← sum e.target's incoming edges' weight
45:         f ← factor × e.weight /sum
46:         SplitCallGraph(G, v, f)
47:     return root
48:
49: function ConstructCallingContext(symbols, samples)
50:     G ← ConstructCallGraph(symbols, samples)
51:     SCCs ← IdentifySCC(G)
52:     G ← ConstructSCCNodes(G, SCCs)
53:     G ← PropagateCallGraph(G)
54:     root ← find v ∈ G.V with indgree(v) == 0
55:     return SplitCallGraph(G, root, 1.0)
```

Like Gprof [14], we assume that every call to a function takes the same amount of time. While Gprof counts calls exactly, we calculate approximate call counts based on PC samples of call instructions. Algorithm 3 presents a sketch of *ConstructCallingContext* method, which takes two input arguments–function *symbols* in GPU binaries and PC *samples* of each GPU kernel calling context. At Line 50, we call *ConstructCallGraph* to construct a call graph $G = \{V, E\}$,

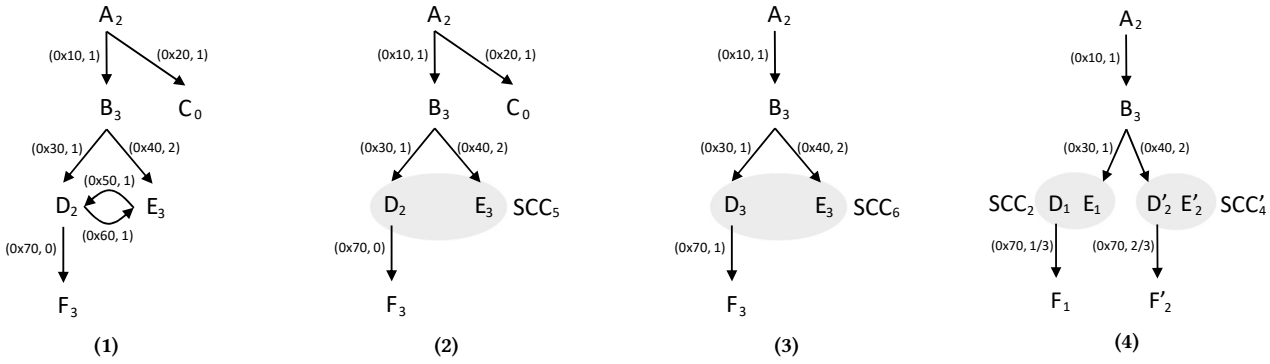

Figure 4: Reconstruct a GPU calling context tree. A-F represent GPU functions. Each subscript denotes the number of samples associated with the function. Each (a, c) pair indicates an edge at address a has c call instruction samples.

where each vertex $v \in V$ is a function, and each edge $e \in E$ is a call relation. For each function symbol (Line 3), we create a vertex with the symbol's start and end addresses and add the vertex to $\mathcal{G}$. Then, for each vertex, we iterate over instructions within its address range (Line 7). If we find a PC sample has the same address as an instruction (Line 9), we assign the sample's count to the instruction and increase the function's total weight (Line 17). If the instruction is a *Call* (Line 11), we add an edge with *source* vertex, *target* vertex, and the instruction's *weight* to $\mathcal{G}$.

Our method handles recursive calls. First, we identify strongly connected components (SCCs) using Tarjan's algorithm [37] at Line 51. Next, we unlink edges between vertices within the SCC, and add an SCC vertex to enclose the set of vertices in each SCC at Line 52. Then, we treat an SCC vertex as a normal "function" in the call graph. In some cases, the leaf function of a long call path has samples, but no call instruction is sampled in the call path that reaches it. In Line 53, we propagate samples to all possible call sites and prune the call graph to include vertices only for functions that appear to have been called at runtime.

PropagateCallGraph employs an iterative pattern to update the call graph. Initially, vertices with positive weight are pushed into a queue (Line 21); one of them is popped at a time (Line 23). If we cannot find any incoming edge of v has positive weight (Line 27), we equally apportion the costs of v among its call sites by assigning each edge to v a weight of one sample (Line 31). We repeat this process until we reach a fixed point where each vertex has at least one incoming edge with positive weight. The propagation ends in $O(|V| + |E|)$ time because a vertex is pushed into the queue at most once (Line 30), and each edge is accessed at most twice (Line 25 and Line 28). After the propagation terminates, we prune vertices with zero samples (Line 34).

In the last step, we build a calling context tree by splitting the call graph at Line 55. Function *splitGraph* uses a depth-first traversal on a call graph to construct a calling context tree. We begin traversing the call graph at its root which does not have any incoming edge (Line 54). In each invocation, we update the current root's weight (Line 38) and its instructions' weight (Line 40) by multiplying by the current apportion *factor*. For each outgoing edge of *root*, we clone a copy of the edge's target vertex (Line 42) to v and create a tree link between *root* and v (Line 43). A new apportion factor f is created based both the current *factor* and the ratio of an edge's weight to the sum of all incoming edges' weight of a vertex (Line 45). In the end, *splitGraph* is applied for v (Line 46).

Figure 4 shows the reconstruction process for a small synthetic example. ① We first construct a draft call graph based on function symbols and call instructions. Each function is annotated with the number of sampled instructions within the its address range; each edge is annotated with the number of times its call instruction is sampled. ② We add an SCC vertex to enclose Function D and Function E. Incoming edges to D and E are linked to SCC. Edges within SCC are unlinked. ③ We propagate sampled functions to all possible call sites. Because F has samples but 0x70 does not have samples, we assign the call site one sample and increase the samples of SCC and D. ④ Since SCC has two potential call sites, we apportion the number of samples of SCC and SCC' using ratios of sampled calls from each call site (1 and 2 for 0x30 and 0x40) to the total number of sampled calls from all call sites (*3*). *F* is also cloned to *F'* and apportioned based on the prior apportion factor.

5.2.3 Derived Metrics. By attributing raw metrics to full calling contexts, we can identify a GPU program's hotspots. To identify opportunities for tuning, we need some additional metrics. Due to the limited number of performance monitoring units, CUPTI cannot collect PC samples in the same pass with other fine-grained metrics. Our tool collects or estimates the essential metrics shown in Table 1. Nsight-compute runs nine passes to collect all of these metrics for a GPU kernel; this approach is infeasible for a large-scale execution that performs millions of kernel launches. In this section, we propose a method to approximate instruction throughput and related metrics based on measurements gathered using PC sampling in a single profiling pass.

In Table 1, *Parallelism* metrics can be estimated using *Launch Statistics* and device properties as described in [46]. Based on the number of total samples (S) and latency samples (S_L), we can estimate the Warp Issue Rate ($\mathcal{W}_\mathcal{R}$) of schedulers, as described in Equation 4. *IPC* (Equation 5) is Warp Issue Rate times the number of threads per warp ($\mathcal{T}$), and times the number of used warp schedulers (W), which is the minimum of the number of warp schedulers and active warps. The total number of eligible instructions observed is the sum of issued samples (S_I) and not selected samples (S_N). So Warp Eligible Rate ($\mathcal{E}$) can be calculated by Equation 6. If any of

Table 1: GPU Kernel Metrics Collected by HPCToolkit. (C) denotes collected metrics. (E) denotes estimated metrics.

Metrics Class	Metrics
Parallelism	Theoretical SM Efficiency (E), Theoretical Occupancy (E), Occupancy Limiters (E)
Compute Workload	IPC (E), Warp Issue Rate (E) Warp Eligible Rate (E), SM Busy Rate (E)
Source Counters	PC Samples (C), Issued Instruction Counts (E)
Launch Statistics	Grid Dimension (C), Block Dimension (C), Shared Memory Usage (C), Register Usage (C)
Instruction Statistics	Instruction Throughput (E)
Device Statistics	Elapsed Cycles (E), Active Ratio (E), SM Frequency (E), Memory Frequency (E)

the warp schedulers in a SM is busy, the SM is considered as busy. Therefore SM Busy Rate ($\mathcal{B}$) is estimated by Equation 7.

$$\mathcal{W}_{\mathcal{R}} = \frac{S - S_L}{S} \tag{4}$$

$$IPC = \mathcal{W}_{\mathcal{R}} \times \mathcal{T} \times W \tag{5}$$

$$\mathcal{E} = \frac{S_I + S_N}{S} \tag{6}$$

$$\mathcal{B} = 1 - (1 - \mathcal{E})^W \tag{7}$$

We collect GPU SM Frequency continuously over time and compute the average frequency $\bar{C}$. Suppose a kernel runs for T seconds on a GPU. With PC sampling period P, we can estimate the total number of samples if the GPU is active all the time (E_S) as shown in Equation 8. A GPU's active rate $\mathcal{A}$—the fraction of cycles when a GPU is actively processing data versus the total elapsed cycles can be estimated by Equation 9. With $\mathcal{A}$ and the estimated used number of *SMs* (N_{SM}), we estimate the Issued Instruction Counts of a kernel ($\mathcal{I}$) with Equation 10.

$$E_S = \bar{C}/P \times T \tag{8}$$

$$\mathcal{A} = \frac{S}{E_S} \tag{9}$$

$$\mathcal{I} = \mathcal{A} \times T \times \bar{C} \times N_{SM} \tag{10}$$

Using $\mathcal{I}$, we can derive throughput metrics for different types of instructions. We map instructions to different classes by splitting opcodes and modifiers to identify operation types, instruction length, tensor operation, and memory hierarchy. For example, a `LDG.64` instruction is categorized as "memory.global.load.64". Then, we count the number of a specific type of issued samples within a kernel as S_{I_t}. In Equation 11, we estimate the throughput of an instruction type by multiplying $\mathcal{I}$ with the ratio of the instruction type among all the issued samples.

$$\mathcal{I}_t = \mathcal{I} \times \frac{S_{I_t}}{S_I} \tag{11}$$

Our estimates may differ from actual values for various reasons, with the most critical factor being predicated instructions. Since PC sampling collects issued instructions instead of executed instructions, our estimated instruction throughput might be higher than actual throughput if predicates are often false. We can fix the error by collecting CUPTI *INSTRUCTION_EXECUTION* records with another pass to get the exact number of executed instructions.

6 CASE STUDIES

We tested HPCToolkit's new support for analyzing GPU-accelerated applications on two platforms: (1) A node of ORNL's Summit supercomputer, which has two IBM Power9 processors and six NVIDIA Volta v100 GPUs; (2) an x86_64 system with two Intel E5-2695 processors and a single NVIDIA Volta v100 GPU. Both platforms were running CUDA-10.1.168. We evaluated the capabilities of HPCToolkit with the three codes listed below:

- RAJAPerf Suite [23] is a set of loop-based GPU kernels implemented using RAJA.
- Laghos [9] is a DOE mini-app that solves the time-dependent Euler equation of compressible gas dynamics. It has both RAJA and CUDA implementations.
- Nekbone [12] is a lightweight subset of Nek5000 that mimics the computational characteristics of Nek5000, a high-order Navier-Stokes solver based on the spectral element method.

We analyzed RAJAPerf Suite and Laghos on Summit and Nekbone on our `x86_64` system. The case study code and experiment setup [38] are available online.

6.1 RAJAPerf Suite

Section 6.1.1 illustrates the utility of heterogeneous calling context trees (CCTs) that span CPU and GPU procedure frames, especially for template-based programming models such as RAJA. Heterogeneous calling contexts provide a foundation for code analysis in our other case studies. Algorithm 3 contributes approximate reconstructions of GPU CCTs to these heterogeneous CCTs. Section 6.1.2 evaluates the accuracy of our approximate reconstructions of GPU CCTs by comparing them with exact measurements obtained using binary instrumentation of GPU machine code.

6.1.1 Top-down Calling Context View. We profiled the `Stream_DOT` kernel in RAJAPerf with PC sampling. The profiling result is shown in Figure 5. The left pane shows lowest ten frames of a CPU calling context enclosing a GPU calling context reconstructed by Algorithm 3. A GPU API Node placeholder `<gpu kernel>` separates our GPU calling context reconstruction from its enclosing CPU calling context. The right pane shows two measured GPU metrics (instruction and stall counts) and a computed instruction mix metric. Figure 6 shows a flat GPU profile view for the same `Stream_DOT` kernel as a contrast to the GPU CCT reconstruction in Figure 5. Figure 6 shows only some of the 25 GPU functions that are direct children of the GPU API node without our GPU CCT reconstruction. The flat profile in Figure 6 provides no insight into the structure of code instantiated from RAJA's templates, whereas our approximate GPU CCT reconstruction in Figure 5 is intuitive.

6.1.2 Calling Context Accuracy. The accuracy of GPU CCT reconstructions with Algorithm 3 rests on two assumptions. To test these assumptions, we used NVIDIA's NVBit dynamic binary instrumentation framework [41] to build a tool that instruments every call instruction, return instruction, and the start of each basic block in a

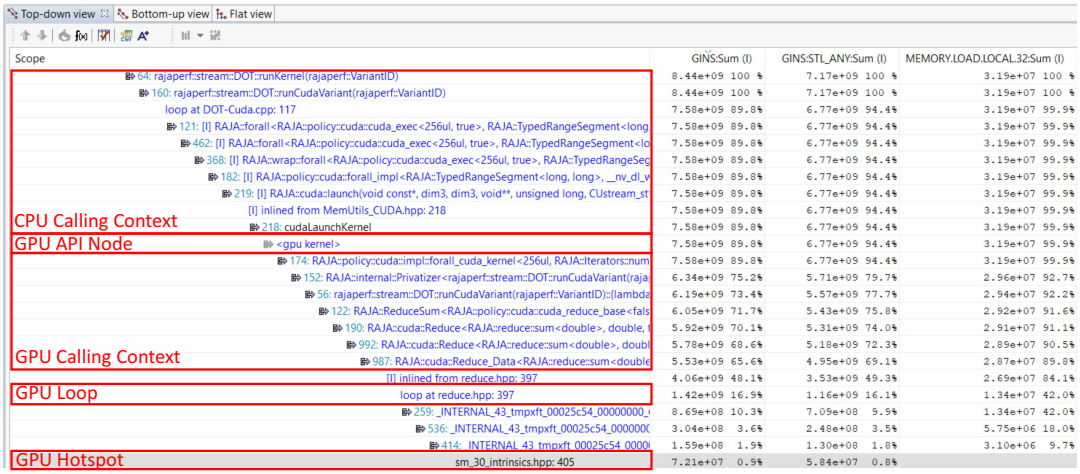

Figure 5: The RAJAPerf Suite `Stream_DOT` kernel with reconstruction of approximate GPU calling contexts.

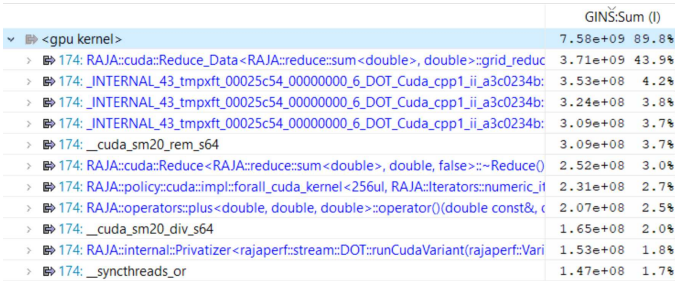

Figure 6: The RAJAPerf Suite `Stream_DOT` kernel without reconstruction of approximate GPU calling contexts.

GPU binary. At every instrumentation site, we read an instruction's PC, current thread id, thread active mask and predicate mask. We store these records into a GPU buffer that communicates between CPUs and GPUs. When the GPU buffer is full, we copy the GPU buffer to the CPU side and update the GPU calling context trees for each GPU thread. Finally, we merge the calling context trees into a single one to represent the calling context for a kernel. We have tested eight kernels from RAJAPerf Suite.

First, we tested if the Gprof assumption—that every call to a function takes the same amount of time—is sensible. We checked if different call sites for a GPU device function have similar basic block count profiles. Our experiments show that basic blocks profiles measured for different call sites for GPU functions are the same, which supports the Gprof assumption for the kernels that we studied. While we know that the assumption may not hold under some circumstances, we expect it to yield useful approximations much of the time.

We also evaluated the accuracy of our call instruction count approximation used to apportion costs among a function's call sites. Formula 12 compares NVBit's accurate call counts with HPCToolkit's approximate call counts. In the formula, $f_N(i,j)$ and $f_H(i,j)$ denote the call count of function i at call site j measured by NVBit and HPCToolkit respectively. First, we compute the root mean square error for each GPU function i of our call count approximation across the i_c call sites. Then, we compute the root mean square across all GPU functions of the root mean square error across each function's call sites. The closer the error is to 0, the more accurate the appromixated call counts.

Table 2: Call Count Errors

Test Case	Unique Call Paths	Error
Basic_INIT_VIEW1D_OFFSET	9	0
Basic_REDUCE3_INT	113	0.03
Stream_DOT	60	0.006
Stream_TRIAD	5	0
Apps_PRESSURE	6	0
Apps_FIR	5	0
Apps_DEL_DOT_VEC_2D	3	0
Apps_VOL3D	4	0

$$Error = \sqrt{\sum_{i=0}^{n-1} \frac{\left(\sqrt{\sum_{j=0}^{i_c-1} \frac{(f_N(i,j)-f_H(i,j))^2}{i_c}}\right)^2}{n}} \tag{12}$$

Table 2 shows Formula 12's results for all eight cases. Most cases have zero error because their device functions only have one call site. Even for Basic_REDUCE3_INT that has 113 unique call paths, the measured error is only 0.03.

In summary, for the cases we studied, the assumptions underlying the quality of our approximate GPU CCT reconstruction are sensible and provide useful and accurate results for compute-intensive GPU kernels such as those in RAJAPerf Suite.

6.2 Laghos

6.2.1 Laghos-CUDA. The overall $I_{C,P}$ for the CUDA version of Laghos is 80%. Among all calling contexts, we observed that function *mfem::LinearForm::Assemble*'s G_i is zero and $I_{C,i}$ is 60%. It suggests developers that offloading this computation to the GPU could be more beneficial than optimizing GPU kernels present.

Next, we delved into the bottlenecks of the code's GPU execution. We used $I_{G,i,j}$ metrics to check the importance of each GPU API. Overall, GPU memory copies take 11% of the GPU execution time. We show the top three GPU memory copies in their CPU calling contexts in Figure 7.

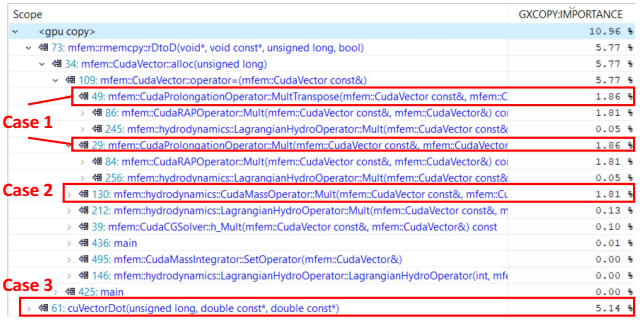

Figure 7: A bottom up view of memory importance in Laghos

We used different methods to optimize each of the three memory copies in Figure 7. For the first case, we noted `mfem::CudaProlongationOperator::Mult` and `mfem::CudaProlongationOperator::MulTranspose` do nothing but copy memory in a single process environment. In `mfem::CudaRAPOperator::Mult`, we can use the original memory without copying. Thus, we move environment check conditions to `mfem::CudaRAPOperator::Mult`, eliminating the two memory copies when using a single process.

In the second case, we observed that an array *distX* is copied from input x and is modified only if *ess_tdofs_count* is not zero. Thus, we propose an optimization that conditionally copies x to *distX* based on *ess_tdofs_count*.

For the third case, we examined the memory copy's invocation times and transferred bytes, inferring that a small piece of data is moved frequently from the GPU to the CPU. Using pinned host memory avoids extra copies between pageable and pinned memory.

Applying the above optimizations to Laghos increased overall performance by 6%, and the code sections that contain GPU APIs increased by 25%. Note that blindly applying optimizations without consulting metrics in the calling contexts can have hurt performance. For example, in the third case, if the amount of transferred memory is massive, using pinned memory reduces the amount of memory available to the system and thereby may slow down the performance as a whole.

6.2.2 Laghos-RAJA. We ran the RAJA version of Laghos with the same input and parameters as the CUDA version. We observed that the RAJA version is 22% slower than the CUDA version, but $I_{G,P}$ of the RAJA version is higher than the CUDA version. Unlike the CUDA version, the RAJA version has high $I_{G,P,sync}$ while the CUDA version does not induce any explicit synchronization. By examining the synchronization contexts, we found that a stream synchronization occurs every time a kernel is launched, because Laghos developers used the default RAJA kernel launch pattern. We fixed the problem by making the kernel launches asynchronous.

After applying the first optimization, we noticed the RAJA version is still slower than the CUDA version. The major difference lies in the `rMassMultAdd2D<3,4>` kernel, where the RAJA version takes 66% more time than the CUDA version. The RAJA code and CUDA code are the same except for the RAJA template wrapper. By comparing every latency reason of the two codes, we found that the RAJA code has 3× the memory dependency latencies and 2× the memory throttling latencies of the CUDA code, which means the RAJA code may cause more memory requests. Next, we checked the instruction mix of the two codes and located a specific line that causes the problem. With the RAJA template wrapper, NVIDIA's `nvcc` compiler cannot infer that store operations in a loop access the same address on the global memory in each iteration so that it generates 4× as many `STG` instructions as the CUDA code.

6.3 Nekbone

Listing 1 A hotspot in Nekbone's Poisson operator

```
1  // G, DT: global memory arrays
2  // ur, us, ut, ul: shared memory arrays
3  for (int it = 0; it < tileSize; it += blockDim.x) {
4    transform(it, i, j, k);
5    for (int n = 0; n < N; ++n) {
6      wr = wr + DT[i][n] * ul[j][k][n];
7      ws = ws + DT[j][n] * ul[j][n][i];
8      wt = wt + DT[k][n] * ul[n][k][i];
9    }
10   double *g = &G[blockId.x][i][j][k];
11   ur[i][j][k] = g[0] * wr + g[1] * ws + g[2] * wt;
12   us[i][j][k] = g[1] * wr + g[3] * ws + g[4] * wt;
13   ut[i][j][k] = g[2] * wr + g[4] * ws + g[5] * wt;
14 }
```

The Poisson operator is a core component of Nekbone and consists of two steps—first calculate partitioned results on each node and then use an MPI exchange to collect the results. In this paper, we focus on the calculation. We describe the most important loop in Listing 1. With the help of HPCToolkit, we identified a sequence of optimization opportunities, as shown in Figure 8.

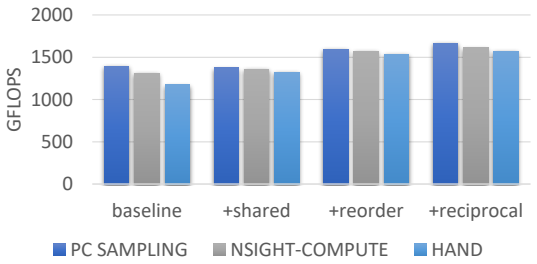

Figure 8: Nekbone optimization steps

We first located a hotspot at Loop Lines 5–9, which takes 32% execution time of which 43% of the latencies are memory throttling. This phenomenon indicates that too many global memory requests occur from accesses to *DT*, and there is no guarantee that all accesses will be satisfied by L1 cache. Since *DT* is small, to reduce the number of global memory requests, we loaded *DT* into shared memory. This *+shared* optimization improved performance by 12%.

After *+shared* optimization, we found Line 11 is problematic. It takes 14% execution time, of which 93% latencies are caused by memory dependencies. By observing that Line 12 and Line 13, which also read global memory, do not cause many memory dependency latencies, we deduced that global memory load instructions (`LDGs`) are not reordered properly to hide latencies. We confirmed our hypothesis by checking the assembly code, finding that `LDG` instructions in Line 11 are not reordered before Line 5. We manually reordered instructions by reading *g[0]-g[6]* before Line 5. This *+reorder* optimization improved performance by 15%.

Next, Loop Lines 5–9 became a hotspot again with 33% execution time. 98% latencies of the hotspot are from execution dependencies and pipeline busyness. For the busy pipe problem, templatization does not help because the index calculation at Lines 6-8 leads to high integer instruction throughput. For the execution dependency problem, we noticed that Line 4 involves integer division operations. Our tool showed that the hotspot's `MUFU` instruction usage is high, which indicates the use of Special Function Units (SFUs) that cause low throughput and long latency. A division between two integers involves complicated steps on GPUs: (1) the dividend and divisor are converted to floating numbers using SFUs, (2) the reciprocal of the divisor is calculated using an SFU, (3) the dividend is multiplied with the reciprocal, (4) the result is converted to an integer using an SFU. To alleviate the problem, we pre-calculated *1/shape_output[j]* and passed them as an array to the kernel. In this way, step 1 only converts an integer, and step 2 is eliminated. This *+reciprocal* optimization improved performance by 3% .

Figure 8 shows HPCToolkit's estimated GFLOPS for all code versions. Except for *baseline*, our estimated results are within 6% of the hand calculation. For *baseline*, neither HPCToolkit nor Nsight-compute reported GFLOPS accurately. We found that counting issued instructions is misleading for this code because some predicated instructions were not executed, while PC samples do not provide predicate information. Nsight-compute by default measures executed instructions at warp level and therefore also has an error in *baseline* measurement.

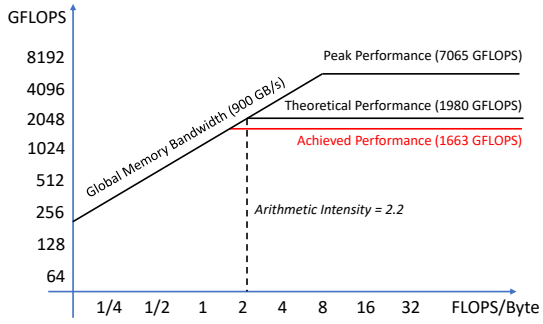

Figure 9: Roofline model of Poisson operator with Double precision. *Arithmetic Intensity* denotes Poisson operator's Double precision operations over global memory traffic

Our tool generated a roofline model [43] to analyze the performance upper bound. We plotted the global memory roofline of *+reciprocal* in Figure 9. Our tool's estimated arithmetic intensity is 8% higher than the actual arithmetic intensity. The theoretical peak performance of this kernel is 1980 GFLOPS, and we achieved 1663 GFLOPS, which is 84% of the peak. As instruction mix implies, the gap between our implementation and the peak performance is caused by unfused multiply instructions and add instructions. If all multiply instructions and add instructions are fused, it could improve performance by 19%, yielding 99% of peak bandwidth-limited performance.

7 RELATED WORK

As mentioned in the introduction, most existing GPU performance tools focus on trace analysis. Only a few tools provide a basic profile view at the cost of high profiling overhead. Instead, HPCToolkit supports a complete profile view with calling contexts for both CPUs and GPUs.

There has been much research work on pinpointing heterogeneous applications' bottlenecks. Malony et al. [25] introduced primitive performance measurement models on GPUs to support tool implementation. Schmitt et al. [31] used trace analysis to compute the critical path for MPI-CUDA applications. Welton and Miller [42] investigated performance issues that have impacted several HPC applications by CPU profiling and GPU API interception. Kousha et al. [21] presented a tool for monitoring communication patterns of applications running on inter-connected GPUs. Prior GPU performance measurement support in HPCToolkit [4] measures and attributes GPU performance metrics with CPU calling contexts; however, these metrics are collected with counters and they provide no insight into the structure or detailed performance of GPU computations.

Like our work, CUDABlamer [44, 45] uses PC sampling to collect heterogeneous call path profiles that include call paths on GPUs and attribute performance at the source line level in GPU code. While the extensions to HPCToolkit described in this paper were designed to measure the performance of GPU-accelerated programs with distributed-memory and multithreaded parallelism on supercomputers, CUDABlamer (1) is fundamentally a proof-of-concept tool that lacks the ability to measure programs with parallelism other than CUDA and records CPU calling contexts as well as GPU measurements in ASCII, (2) only measures GPU activity, (3) only applies to programs compiled with LLVM as it use LLVM-IR for static analysis of calling contexts, (4) distributes inclusive costs of GPU functions equally across all of their call sites whereas we use samples of GPU call instructions to infer a more accurate distribution of costs, and (5) presents only PC samples and doesn't use them to compute a wide range of metrics as we do.

Reconstructing calling contexts has been studied extensively. Graham et al. [14] introduced `Gprof` that constructs a call graph and reports costs at each function. Unlike `Gprof`, our method propagates sampled call instructions and reconstructs a calling context tree. Previous call instruction propagation studies, including Grove et al. [15, 16] and Tip et al. [39], principally focused on static call graph reconstruction but not dynamic calling context tree.

There is also prior work on fine-grained GPU performance analysis that does not leverage PC sampling. Zhou et al. [46] and Hong et al. [17] introduced an analysis framework on assembly code to derive performance bottlenecks. Hong et al. [17] simulated kernel execution several times after changing resource parameters.

Both of the aforementioned studies are based on either simulation or programmer specified notations (e.g., number of iterations) in programs. In contrast, HPCToolkit profiles applications on real hardware and analyzes GPU binaries.

Recently, there has been research on GPU kernel instrumentation. NVBit [41] and SASSI [35] provide callback APIs to inject code for assembly instructions. Users can leverage these APIs to collect fine-grained metrics, such as branch divergence, memory divergence, and value profiling. Unlike the above tools that instrument GPU binaries, CUDAAdvisor [33] and CUDA Flux [3] are LLVM-based instrumentation tools for exploring additional insights. However, identifying code hotspots with instrumentation introduces significant overhead because of instrumented measurement instructions and thus affects the accuracy. Goroshov et al. [13] measure block latencies to reduce overhead. Even with this approach, instrumented kernels are on average 25% slower than the original code. In HPCToolkit, we use PC sampling to identify hotspots of GPU kernels. Since the hardware counter collects PC samples, it does not slow down the GPU kernel itself and renders accurate results. Therefore, these fine-grained instrumentation APIs are complementary to what our sampling-based measurement tool provides.

8 CONCLUSIONS

With the extensions described in this paper, HPCToolkit can efficiently collect performance metrics for GPU-accelerated applications, attribute metrics to heterogeneous calling contexts, loops, and source lines. Through case studies, we demonstrated that HPCToolkit's extensions for analyzing GPU-accelerated codes can provide insights for both individual kernels and whole applications. We have begun to apply HPCToolkit to sophisticated applications that employ MPI, OpenMP, CUDA, and OpenACC. We are continuing to improve the capabilities of HPCToolkit with the aim of turning measurement data and metrics for GPU-accelerated applications into high-level guidance for performance tuning.

9 ACKNOWLEDGEMENTS

We thank the reviewers for their feedback and suggestions. We thank members of the HPCToolkit project for contributing infrastructure that made this work possible. In particular, we thank Dr. Xiaozhu Meng, Tijana Jovanović, and Aleksa Simović, who helped develop components of HPCToolkit's GPU measurement infrastructure. We also thank Lixiong Liu from IBM for providing a tensor transpose code for our experiments.

This research was supported by Ken Kennedy Institute ExxonMobil Graduate Fellowship, Subcontract B633244 from Lawrence Livermore National Laboratory under DOE/NNSA Prime Contract DE-AC52-07NA27344 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[2] Matthew Arnold and Peter F. Sweeney. 2000. *Approximating the calling context tree via sampling.* Technical Report RC-21789. IBM T.J. Watson Research Center, Yorktown Heights, NY.

[3] 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.

[4] Milind Chabbi, Karthik Murthy, Michael Fagan, and John Mellor-Crummey. 2013. Effective sampling-driven performance tools for GPU-accelerated supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* ACM, 43.

[5] NVIDIA Corporation. 2019. *NVIDIA Nsight Compute.* https://developer.nvidia.com/nsight-compute [Accessed January 26, 2020].

[6] NVIDIA Corporation. 2019. *NVIDIA Nsight Systems.* https://developer.nvidia.com/nsight-systems [Accessed January 26, 2020].

[7] NVIDIA Corporation. 2019. *PC Sampling.* https://docs.nvidia.com/cupti/Cupti/r_main.html#r_pc_sampling [Accessed January 26, 2019].

[8] NVIDIA Corporation. 2019. The user manual for NVIDIA profiling tools for optimizing performance of CUDA applications. https://docs.nvidia.com/cuda/profiler-users-guide [Accessed January 26, 2020].

[9] Veselin A Dobrev, Tzanio V Kolev, and Robert N Rieben. 2012. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM Journal on Scientific Computing* 34, 5 (2012), B606–B641.

[10] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[11] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. 2005. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05).* ACM, New York, NY, USA, 81–90. https://doi.org/10.1145/1088149.1088161

[12] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. 2016. Nekbone performance on GPUs with OpenACC and CUDA Fortran implementations. *The Journal of Supercomputing* 72, 11 (2016), 4160–4180.

[13] Anton V Gorshkov, Michael Berezalsky, Julia Fedorova, Konstantin Levit-Gurevich, and Noam Itzhaki. 2019. GPU Instruction Hotspots Detection Based on Binary Instrumentation Approach. *IEEE Trans. Comput.* 68, 8 (2019), 1213–1224.

[14] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, Vol. 17. ACM, 120–126.

[15] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[16] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices* 32, 10 (1997), 108–124.

[17] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. 2018. Gpu code optimization using abstract kernel emulation and sensitivity analysis. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 736–751.

[18] Richard D Hornung and Jeffrey A Keasler. 2014. *The RAJA portability layer: overview and status.* Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[19] Christopher January, Jonathan Byrd, Xavier Oró, and Mark O'Connor. 2015. Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead. In *Tools for High Performance Computing 2014.* Springer, 25–35.

[20] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. 2008. The vampir performance analysis tool-set. In *Tools for High Performance Computing.* Springer, 139–155.

[21] Pouya Kousha, Bharath Ramesh, Kaushik Kandadi Suresh, Ching-Hsiang Chu, Arpan Jain, Nick Sarkauskas, Hari Subramoni, and Dhabaleswar K Panda. 2019. Designing a Profiling and Visualization Tool for Scalable and In-depth Analysis of High-Performance GPU Clusters. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, 93–102.

[22] Lawrence Berkeley National Laboratory. 2019. *Cray Reveal.* https://docs.nersc.gov/programming/performance-debugging-tools/reveal [Accessed January 26, 2020].

[23] Lawrence Livermore National Laboratory. 2019. *RAJA Performance Suite.* https://github.com/LLNL/RAJAPerf [Accessed January 26, 2020].

[24] Robert Lim, Allen Malony, Boyana Norris, and Nick Chaimov. 2015. Identifying optimization opportunities within kernel execution in GPU codes. In *European Conference on Parallel Processing.* Springer, 185–196.

[25] Allen D Malony, Scott Biersdorff, Sameer Shende, Heike Jagode, Stanimire Tomov, Guido Juckeland, Robert Dietrich, Duncan Poole, and Christopher Lamb. 2011. Parallel performance measurement of heterogeneous parallel systems with gpus. In *2011 International Conference on Parallel Processing*. IEEE, 176–185.

[26] NVIDIA Corporation. 2019. *CUPTI User's Guide DA-05679-001_v10.1.* https://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf.

[27] University of Wisconsin-Madison. [n.d.]. *Dyninst.* https://github.com/dyninst/dyninst [Accessed January 26, 2020].

[28] OpenACC-Standards.org. 2019. *The OpenACC Application Programming Interface Version 3.0.* https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf.

[29] OpenMP Language Committee. 2018. *OpenMP Application Programming Interface Version 5.0.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[30] James Reinders. 2005. VTune performance analyzer essentials. *Intel Press* (2005).

[31] Felix Schmitt, Robert Dietrich, and Guido Juckeland. 2017. Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications. *The International Journal of High Performance Computing Applications* 31, 6 (2017), 485–498.

[32] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. 2008. Open| SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming* 16, 2-3 (2008), 105–121.

[33] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 214–227.

[34] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[35] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R Johnson, David Nellans, Mike O'Connor, and Stephen W Keckler. 2015. Flexible software profiling of gpu architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 185–197.

[36] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2019. *Top500 ranking June 2019.* https://www.top500.org/lists/2019/09/ [Accessed January 19, 2020].

[37] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[38] HPCToolkit Team. 2020. *HPCToolkit ICS-2020 Artifacts.* https://github.com/HPCToolkit/paper-artifacts/tree/master/ICS-2020 [Accessed April 26, 2020].

[39] Frank Tip and Jens Palsberg. 2000. *Scalable propagation-based call graph construction algorithms*. Vol. 35. ACM.

[40] R Kent Treiber. 1986. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research.

[41] 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 International Symposium on Microarchitecture*. ACM, 372–383.

[42] Benjamin Welton and Barton Miller. 2018. Exposing hidden performance opportunities in high performance GPU applications. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 301–310.

[43] Samuel Williams, Andrew Waterman, and David Patterson. 2009. *Roofline: An insightful visual performance model for floating-point programs and multicore architectures.* Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[44] Hui Zhang. 2018. *Data-centric performance measurement and mapping for highly parallel programming models.* Ph.D. Dissertation. University of Maryland—College Park.

[45] H. Zhang and J. Hollingsworth. 2019. Understanding the Performance of GPGPU Applications from a Data-Centric View. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. 1–8. https://doi.org/10.1109/ProTools49597.2019.00006

[46] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. 2017. A performance analysis framework for exploiting GPU microarchitectural capability. In *Proceedings of the International Conference on Supercomputing*. ACM, 15.