Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool

Benjamin Welton and Barton P. Miller
Computer Sciences Department
University of Wisconsin - Madison
Madison, Wisconsin
(welton,bart)@cs.wisc.edu

ABSTRACT

GPU accelerators have become common on today's leadership-class computing platforms. Exploiting the additional parallelism offered by GPUs is fraught with challenges. A key performance challenge faced by developers is how to limit the time consumed by synchronization and memory transfers between the CPU and GPU. We introduce the feed-forward measurement (FFM) performance tool model that automates the identification of unnecessary or inefficient synchronization and memory transfer, providing an estimate of potential benefit if the problem were fixed. FFM uses a new multi-stage/multi-run instrumentation model that adjusts instrumentation based application behavior on prior runs, guiding FFM to problematic GPU operations that were previously unknown. The collected data feeds a new analysis model that gives an accurate estimate of potential benefit of fixing the problem. We created an implementation of FFM called Diogenes that we have used to identify problems in four real-world scientific applications.

KEYWORDS

GPUs, Performance Tools, Performance Analysis, Feed Forward Measurement, Multi-stage/Multi-run Performance Tool Design

ACM Reference Format:
Benjamin Welton and Barton P. Miller. 2019. Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3295500.3356213

1 INTRODUCTION

There are several challenges that a developer faces when porting or creating applications on today's leadership-class high performance computing platforms. One of the more challenging aspects of programming for high performance computing platforms is handling interactions between the CPU and GPU accelerator efficiently. Synchronizations and memory transfers between the CPU/GPU are two of the most time intensive operations that can be performed by an GPGPU application. Knowing the high cost of these operations, developers attempt to limit their usage to only locations in their program to where they think these operations are necessary. However, even in applications developed by expert GPU programmers, problematic synchronizations and memory transfers can account for as much as 85% of execution time in real world applications [28].

Developers need tools to help unlock performance lost due to problematic synchronizations and memory transfers. Developers typically use a performance tool, such as HPCToolkit [17], TAU [14, 26], and NVProf [21], to help identify potentially problematic synchronizations and memory transfers. However, the help existing tools provide is limited by gaps in the performance data that they collect, including 1) performance data not recorded for all GPU operations, 2) incomplete performance data recorded for some GPU operations, and 3) for the information that is collected, its often at an insufficient granularity to make a determination if an operation that appears to be problematic can actually be improved. These gaps are caused by vendor supplied interfaces (on which all current tools depend) that provide incomplete information about the operations taking place. In addition, these interfaces are too coarse-grained because providing the level of detail needed to detect and correct problematic operations is considered to be too costly.

The results produced by existing tools describe the resource consumption at points in the program, but not *the benefit that could be obtained* if those points were made more efficient. The assumption is that points of high resource consumption correlate to the points with the highest obtainable benefit. However, as early work on critical path analysis [10] showed, resource consumption was not always a good predictor of the obtainable benefit. When a potential problematic operation is identified by a tool, a detailed manual analysis of the operation is still required to determine if the operation is problematic and to determine what action to take. The result is that programmers spend time optimizing operations that produce limited benefit, while missing others that might provide significant improvements to performance. Providing an estimate of expected benefit for fixing a problem would enable programmers to identify these missed performance

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356213

opportunities and spend their time effectively when improving application performance.

Delivering actionable feedback requires a targeted approach to performance data collection and analysis. To this end, we introduce a multi-stage, multi-run performance measurement and analysis approach called *feed forward measurement* (FFM). The principal idea of FFM is that the insertion of instrumentation into an application and the performance data that is collected is guided by the application's behavior. The applications behavior during execution guides FFM to potentially problematic GPU operations, including those that are hidden from existing tools with a reliance on vendor supplied interfaces. The collection of performance data is split over multiple stages of instrumentation conducted over multiple runs, allowing for potentially problematic operations that are discovered to be profiled and traced at increasing levels of detail. The changing level of detail over multiple stages allows the FFM approach to collect the fine-grained details needed to automate analyses that are too costly for other methods to collect. The analysis performed by FFM gives targeted feedback on what operations are problematic along with an estimate of the performance benefit that could be obtained if the problem were corrected.

FFM is inspired by the dynamic instrumentation approach originally developed in the Paradyn Performance Tool [9]. Unlike Paradyn's approach of running each stage of instrumentation in a single run of the application, FFM runs each stage in a separate complete run of the application. The multi-run approach was chosen to gather the information from a complete run before making decisions on what level of detail to collect for an operation. With Paradyn's single run approach, if an operation is not known to be potentially problematic before it's last occurrence, the chance to collect additional detail on the operation is missed.

FFM consists of five stages, four data collection stages that take place in separate runs of the application and an analysis stage that uses the data collected to identify problematic operations. FFM uses binary instrumentation of CPU code to collect performance data on synchronization and memory transfer events, capturing events such as synchronizations that are missed by vendor-supplied performance data collection frameworks and library interposition methods. Binary instrumentation allows FFM to maintain compatibility with applications written in a wide range of parallelization frameworks (such as CUDA [19], OpenACC [29], and OpenMP [3]). The five stages of the FFM model are:

Stage 1 - *Baseline Measurement*: Collect the list of application called functions performing a GPU synchronization and measure overall application execution time. The baseline measurement stage is designed to be low overhead to ensure that application execution time and behavior closely match its uninstrumented form. The starting point of the FFM model is a list of functions called by the application that perform a synchronization operation. This list dictates where more detailed information will be collected in stages 2 and 3. We describe the Baseline Measurement stage in more detail in Section 3.1.

Stage 2 - *Detailed Tracing*: Trace calls to functions performing synchronization and memory transfers. For each transfer and synchronization operation, we record the amount of time spent in the call and a stacktrace. The traced functions are the ones identified in stage 1 as performing a synchronization and a predefined set of GPU driver function calls known to perform memory transfers. We discuss the Detailed Tracing stage in Section 3.2.

Stage 3 - *Memory Tracing and Data Hashing*: Collect the data needed to determine if an operation is problematic. Two different collection approaches are employed based on the type of the operation. For a synchronization operation we collect a stacktrace of the synchronization, the location of the instruction that first accesses a memory location containing data that could be modified by the GPU, and a stacktrace of the instruction location that performed the access. For memory transfers, we collect hashes of the data being transferred to and from the GPU. We discuss the details of the Memory Tracing and Data Hashing stage in Section 3.3.

Stage 4 - *Sync-Use Analysis*: Collect timing information to determine if a synchronization is misplaced. The time between a synchronization and the first instruction that accesses data computed by the GPU after the synchronization is recorded. A large time gap indicates a potentially misplaced synchronization and is used by the Analysis stage to determine if the operation is problematic. The instruction accessing GPU computed data is obtained from the Memory Traces collected in stage 3. We discuss the details of the Sync-Use Analysis stage in Section 3.4.

Stage 5 - *Analysis*: Determine if an operation is problematic and what the potential benefit might be from correcting the operation. For a synchronization operation, we use a simple data flow analysis approach to determine the necessity of the synchronization. We look for accesses to the data protected by the synchronization on the CPU to detect if the synchronization operation could be moved (or removed) to improve CPU/GPU overlap safely. For data transfers, we use a content-based data deduplication approach to detect problematic data transfers. We describe the Analysis stage in Section 3.5.

We have implemented the FFM model in a tool we call Diogenes. Diogenes identifies problematic synchronizations and memory transfers, estimating an expected benefit for the correction of each issue. Through the use of the FFM model, Diogenes is able to identify operations that are unreported by existing performance tools (including vendor supplied tools such as NVProf [21] and CUPTI [20]) and provides actionable feedback on what problematic operations are correctable. Note that for evaluation purposes, we built Diogenes specifically to identify problematic synchronization and memory transfer operations. Diogenes is not a replacement for a general purpose profiling tool but a supplement that aids in the identification of these problematic operations. Our next step is to integrate our collection and analysis approaches into an existing general purpose profiling tool. Diogenes collected performance data is stored in a standard format (JSON) that can be read by other tools. While we generate estimates of

expected benefit for synchronization and memory transfers, our techniques can be applied to other problem types and be used in other tools.

In Section 4, we describe the implementation of Diogenes. In Section 5, we evaluate Diogenes by using it to identify problematic operations in four real world applications (cumf_als [27], cuIBM [12], AMG [31], and Rodinia [4]). By fixing the problems identified by Diogenes, we were able to improve the performance of these applications by as much as 17%. We also compare Diogenes results to the performance tools HPCToolkit [17] and NVProf [21]. Diogenes expected benefit output differs significantly, in both output order and magnitude, for synchronizations and memory transfer operations. The difference in magnitude can be as much as 99% for these operations. While the implementation details of Diogenes focus on applications accelerated with Nvidia's GPUs, the general techniques described in this paper of the FFM model can be applied to different types of accelerators.

2 THE PERFORMANCE TOOL GAP

We benefit from a long history of performance tool research. Tools such as TAU [14, 26], ParaProf [2], HPCToolkit [17], Periscope [7], ScoreP [11], Scalasca [6], Paraver [23], Quartz [1], Jumpshot [30], and KOJAK [18], plus the contributions of many others, built the foundation on which modern parallel performance tools are based. While a survey of these tools is outside the scope of this paper, the influence they have had on performance tool development can be felt today in the new generation of tools built to support profiling and tracing of GPU applications. Existing tools have been modified and new tools created to detect GPU idleness [5, 11, 15, 16, 21], CPU idleness waiting on GPU completion [5, 11, 15, 21], warp occupancy [5, 15, 21], cache behavior [15, 21], and on-device synchronization issues [5, 21].

The performance tools developed for new architectures today share a common structure with their ancestors. These tools typically operate with a single stage of instrumentation performed on a single run of the application, focus on the task of collecting resource utilization information at points in the program, and rely on vendor supplied performance data collection frameworks when resource information cannot be collected directly. While this structure has helped to produce tools that can find performance issues in applications, there are problems that are hard to detect even with the help of a performance tool. In this section, we describe the benefits and drawbacks that a single stage instrumentation structure imposes on tools and their reliance on vendor supplied black box frameworks for performance data collection.

2.1 Single Stage Instrumentation

Most existing tools are structured such that the instrumentation inserted and the types of performance data collected is static for a single run of the program. The instrumentation used for a given run of the program is set, typically by the tools user, before execution even begins and is not adjustable during execution. For the problems that can be exposed in a single run of a program, this design allows a general purpose tool to be applied to a wide array of different problem types.

However, it is often the case that a single fixed set of instrumentation performed during a single run of the program is not sufficient to fully diagnose a problem. Diagnosing a problem may require the measurement of multiple different resources where the measurement of one resource impacts the accuracy of the measurements of another or changes application behavior in such a way to make the measurements inaccurate. Identifying problematic synchronization and locking behavior are classic examples of problem types that experience these issues. Both problems are sensitive to even small changes in application behavior but require detailed instrumentation to detect that changes the behavior of the application. The result of these limitations is that in practice a user must run a tool (or multiple tools) many times to gather the resources information necessary and perform their own analysis to detect a problem.

The notable exceptions to single stage tool structure are Paradyn [9] and NVProf [21]. Paradyn performs multiple stages of instrumentation over a single run of the application. The multi-stage approach allows Paradyn to focus the collection of additional detail on only the most resource consuming operations. As the application executes, the operations consuming more resources are instrumented at increasing levels of detail. However, operations that are impactful can be missed if the operation completes before Paradyn determines the operation is important. To avoid potential gaps in collection and analysis, FFM uses a multi-run model to ensure that all important operations are known in advance so that detail is not missed.

NVProf uses selective multi-run instrumentation when collecting performance counter information from the GPU. NVProf will rerun a GPU kernel within an application multiple times to record the values of different performance counters. The rerun of a GPU kernel is required due to hardware limits on the number of performance counters that can be recorded in a single execution. Unlike NVProf, FFM performs reruns on the entire application and targets CPU code for its multi-stage instrumentation approach.

2.2 Black Box Collection Frameworks

The introduction of accelerators into HPC platforms, such as GPUs, has changed the way in which performance tools collect performance data. HPC accelerator vendors attempt to limit details about their physical hardware and software subsystems, instead providing developers an abstracted framework when using their platforms. Without detailed hardware and software information, performance tools must rely on closed source vendor supplied frameworks for performance data collection. With closed source collection frameworks, tools have no means to check if the performance data collected is accurate and complete.

For Nvidia GPUs, virtually all GPU profiling and tracing tools rely on the closed source performance data collection framework CUPTI [20]. Existing tools rely on CUPTI to

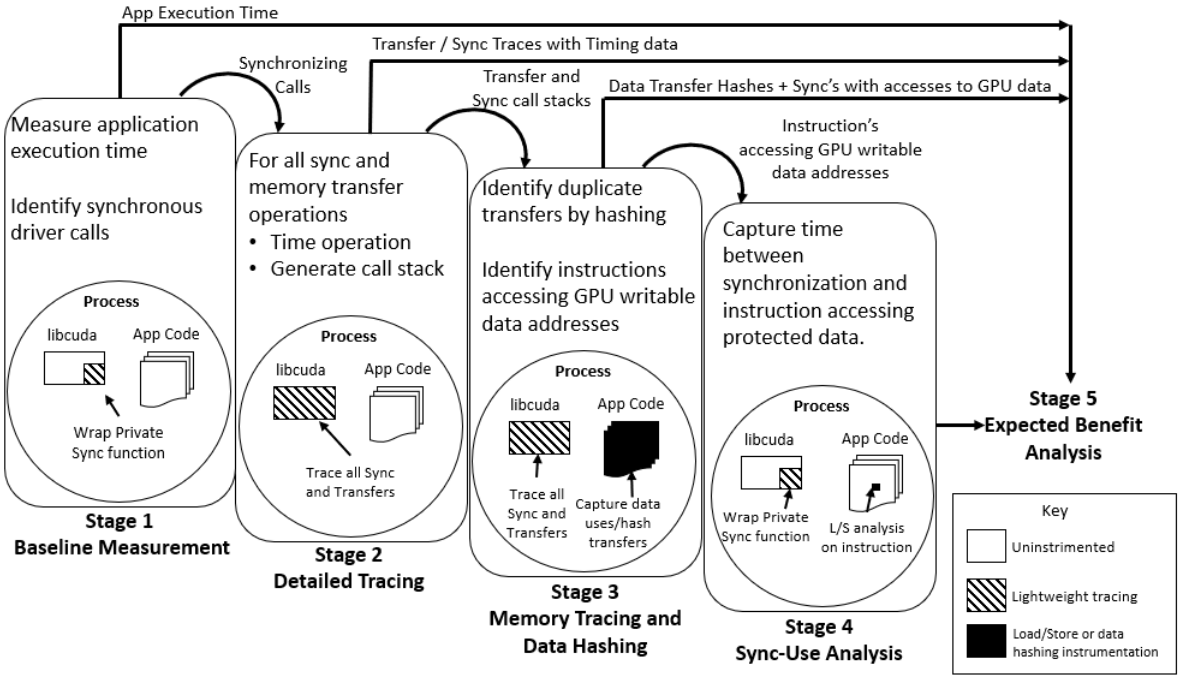

Figure 1: Overview of the stages of the FFM model

report when a driver call is made, when certain high impact operations like memory transfers and synchronizations take place, and to collect performance counter data from the GPU itself. During the course of creating the FFM model, we have discovered that CUPTI does not report when all CPU/GPU synchronizations take place or all the function calls made to the driver.

For most implicit and conditional synchronization operations, CUPTI does not create synchronization (`CUPTI_ActivitySynchronization`) records containing information on how long the synchronization with the GPU took. Implicit synchronization operations are those that occur as a side effect to another operation such as a memory transfer (e.g, `cudaMemcpy`). Conditional synchronization operations are those that occur if certain arguments are supplied to a GPU API call. For example, `cudaMemcpyAsync` performs an unreported synchronization when a device-to-host transfer is performed to a CPU memory address not allocated via `cudaMallocHost`. These behaviors are not reported by CUPTI and are not always well documented. In addition, they may be subject to change based on driver version. We believe that CUPTI only generates synchronization timing information for explicit synchronization operations such as `cudaStreamSynchronize`.

In certain circumstances, CUPTI does not record driver calls and operations. If an operation is performed via the proprietary non-public part of Nvidia's driver, the call and the operation it performs are not reported. The proprietary driver components are used by Nvidia-created libraries like cuBLAS and can perform all the same operations as the public facing driver API. The extent to which these proprietary components are used and how their behaviors effect application performance is still being explored. Finally, CUPTI might omit calls to the public API if they are called from Nvidia-created libraries.

The lack of a full detailed accounting of GPU operations results in the tools and techniques built using CUPTI being potentially less effective. While we hope that these problems are fixed in future versions of CUPTI, FFM does not use CUPTI for performance data collection. We directly instrument the internal functions of the GPU user space driver using binary instrumentation to capture when operations such as synchronizations take place. FFM can capture and time the synchronization delay of implicit, conditional, and non-public API synchronizations.

CUDAAdvisor [25] is one of the few existing tools that does not rely on vendor supplied frameworks for GPU performance data collection. CUDAAdvisor is an LLVM-based runtime profiler that performs fine grained memory and control flow analysis of GPU kernels, detecting performance issues such as inefficient GPU kernel memory access patterns and branching behavior. Memory, arithmetic, and control flow operations performed on the GPU are traced by CUDAAdvisor. An application GPU kernel is modified at compile time by an LLVM plugin to insert instrumentation directly into the GPU kernel. The collected GPU trace data is associated with memory allocations and transfers performed by the CPU, allowing a data flow to be constructed to show which GPU kernels are accessing the same underlying data. Using the data flow graph, CUDAAdvisor can detect potentially problematic memory access behaviors such as differences in GPU kernel memory access patterns accessing the same

underlying data. First party performance tool frameworks were deemed insufficient by the authors because they could not collect the fine-grained GPU trace data necessary to perform their analysis. FFM targets a different set of problems than those of CUDAAdvisor, though FFM also relies on binary modification for instrumentation. Note that FFM focuses on the collection of fine-grained details of operations performed on the CPU instead of the GPU.

3 FEED FORWARD PERFORMANCE MODEL

The fundamental problem that the Feed Forward Performance Model (FFM) was created to address is improving the actionability of feedback given to the user. For the task of improving performance, a tool user needs to know what problems exist and the potential benefit they may obtain from fixing the problems.

The Feed Forward Performance Model (FFM) is a multi-stage/multi-run approach to performance analysis that is designed to allow more actionable feedback to be delivered to the user. The multi-stage/multi-run instrumentation style enables FFM to adjust instrumentation based on application behavior and allows high overhead instrumentation to be used without hiding problems sensitive to overheard. The results produced by FFM's analysis list specific problems in the application with an estimate of potential benefit. We apply these principals to create a model for the identification of problematic synchronizations and memory transfers.

The FFM model for identifying problematic synchronizations and memory transfers consists of five stages where each stage is driven by the instrumentation inserted and the data collected in the proceeding stages. Figure 1 shows an overview of the stages of FFM, the data each stages collects, and how the stages interact. It is important to note, no user interaction is required between stages and the execution of these stages is designed to be automated.

The FFM model detects two types of problematic synchronization operations, unnecessary and misplaced. When the removal of a synchronization operation would have no impact on application correctness, the synchronization is potentially unnecessary. When a synchronization operation is required but could be moved to a more performance-advantageous location, the synchronization is potentially misplaced. The problematic memory transfers that the FFM model detects are duplicate data transfers. When a problem is identified, FFM's analysis will display the potential benefit that could be obtained by fixing the problem.

In this section, we describe the role that each stage plays in detecting problematic operations including the data they collect, how each stage of the multi-stage approach uses prior performance data to make instrumentation decisions, and the analysis we perform to generate targeted actionable feedback that a user can use to improve application performance.

3.1 Baseline Measurement

The baseline measurement stage is responsible for recording application execution time and recording stack traces of where synchronization operations are performed. Application execution time is stored for use by the analysis stage to determine the percentage of execution time a problematic synchronization or memory transfer consumes. The stack traces are used to determine the GPU driver functions called by the application that synchronize with the GPU. This list of functions is then traced in the Detailed Tracing stage. We collect the list of synchronizing functions in advance of the Detailed Tracing stage to ensure complete trace information can be collected for all synchronization operations.

The stack traces are obtained by inserting binary instrumentation into the internal driver function (See Figure 3) that waits for completion of compute stream activity. This underlying function is called by all operations, including conditional and private API operations, that need to synchronize (such as `cuMemcpy` and `cuCtxSynchronize`). The direct instrumentation of the function implementing the wait allows FFM to detect synchronization operations that are missed by the vendor supplied performance data collection methods. We identify the underlying function that performs the wait by a set of simple tests that launches a never completing GPU kernel, calling known synchronous functions (such as cuCtxSynchronize) to identify the function where the CPU waits.

3.2 Detailed Tracing

The detailed tracing stage traces all synchronization and memory transfer operations performed by the application. For each operation, we collect a stacktrace of the operation, the time spent performing the synchronization (if applicable), and the total time spent in the driver function performing the operation. This information is used by the analysis stage to determine the time that could be saved removing an operation.

We insert exit/entry instrumentation into three classes of functions: 1) synchronizing functions identified in the Baseline Measurement stage, 2) functions described by the GPU driver API as performing memory transfers (such as `cuMemcpy`), and 3) internal synchronization function shown in Figure 3.

3.3 Memory Tracing and Data Hashing Stage

The Memory Tracing and Data Hashing stage detects if an operation is problematic. An operation is problematic if it can be removed or moved to a more performance-advantageous location while maintaining application correctness. Problematic synchronizations are ones that are not required to maintain correctness and ones that are required for correctness but unnecessarily reduce CPU/GPU overlap. Problematic memory transfers are duplicate transfers where the data being transferred has been previously transferred.

FFM relies on binary modification to collect the information needed to determine if an operation is problematic.

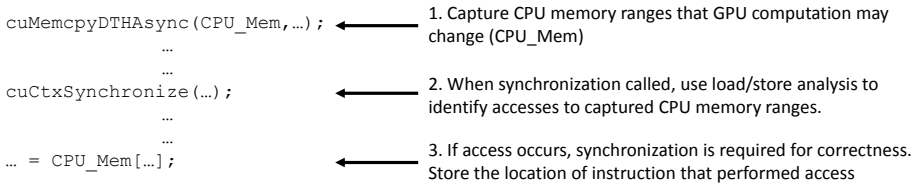

Figure 2: An illustrative example of the steps the FFM model takes to identify problematic synchronizations

For synchronization operations, we use memory tracing. For memory transfer operations, we use a content-based data deduplication strategy.

3.3.1 Identifying Problematic Synchronizations. FFM determines if a synchronization operation is required by identifying the accesses to the data protected by the operation. If a synchronization is not protecting data accessed by the program, the synchronization is problematic. FFM must identify the locations of protected data and identify if any instruction accesses the data after a synchronization takes place.

Figure 2 shows an example of how FFM identifies problematic synchronizations. FFM first identifies the locations of protected data by intercepting calls that transfer data and allocate pages shared between the CPU/GPU. We record the CPU memory addresses and the size of the memory region used in the operation. After the synchronization completes, load/store analysis is used to determine if any instruction accesses data in these regions. If an instruction accesses GPU computed data, the instruction's address and a callstack of the synchronization are saved.

3.3.2 Identifying Problematic Memory Transfers. Problematic memory transfers are transfers that contain data that has already been transferred between the CPU/GPU. FFM uses a content-based data deduplication approach to identify problematic memory transfers. FFM intercepts calls, such as `cuMemcpy`, to obtain the location of the data being transferred. The data being transferred is hashed and then compared to the stored hashes from prior transfers. If a match is found, FFM marks the transfer as being a duplicate. FFM collects a stacktrace of the duplicate transfer, the location of the first transfer of the duplicated data, and the hash of the data that was transferred.

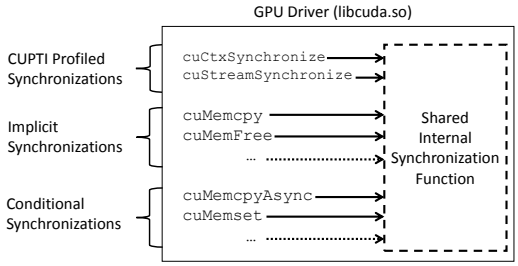

Figure 3: The internal synchronization function instrumented by FFM

3.4 Sync-Use Analysis

The Sync-Use Analysis stage collects timing information to determine if a synchronization is misplaced. For synchronizations identified as being required for correctness, we record the time between the end of the synchronization and the first access of protected data. Sync-Use analysis is based on load/store instrumentation of those instructions identified as accessing protected data in stage 3.

3.5 Analysis Stage

The actual benefit obtained from (re)moving a problematic operation is impacted by the duration of the problematic operation and the operations that remain. As first observed in early work on critical path analysis [10], changes in the behavior of remaining operations can eliminate any benefit from fixing problematic operations. An example can be seen in Figure 4. Removing the first wait operation ($CWait_0$) from both examples results in different outcomes, even though the removed wait has an identical duration. The difference is due to the impact the removal has on the second wait. In the limited-benefit case, the second wait grows to fill up most of the time saved from the first wait. Modeling the behavior of the (re)moved problematic operation on the remaining operations is critical to generating an effective estimate. For each problematic operation identified in stage 3 and 4, we want to model the effect of fixing the problem has on application execution time.

We model application execution as a graph (see Figure 4) $G = (N, V)$, where N is the set of events on each processor and V is the set of edges. An edge between processors denotes communication to signal the other processor. $N = C, G$ where C is the set of CPU nodes in the graph and G is the set of GPU nodes in the graph.

Each node has attributes $(NType, STime, Problem, FirstUseTime)$ associated with it, where $NType$ denotes the event performed by the node, $STime$ is the start time of the event, $Problem$ is the problematic operation identified in stages 3 and 4 (None, Unnecessary Synchronization, Misplaced Synchronization, Unnecessary Transfer), and $FirstUseTime$ is the duration between a synchronization event and the first use of protected data on the CPU (calculated in stage 4). $NType$ can be a wait event where a processor is waiting on the other processor ($CWait$ on the CPU, $GWait$ on the GPU), a work event where the processor is performing computation ($CWork$ on the CPU, $GWork$ on the GPU),

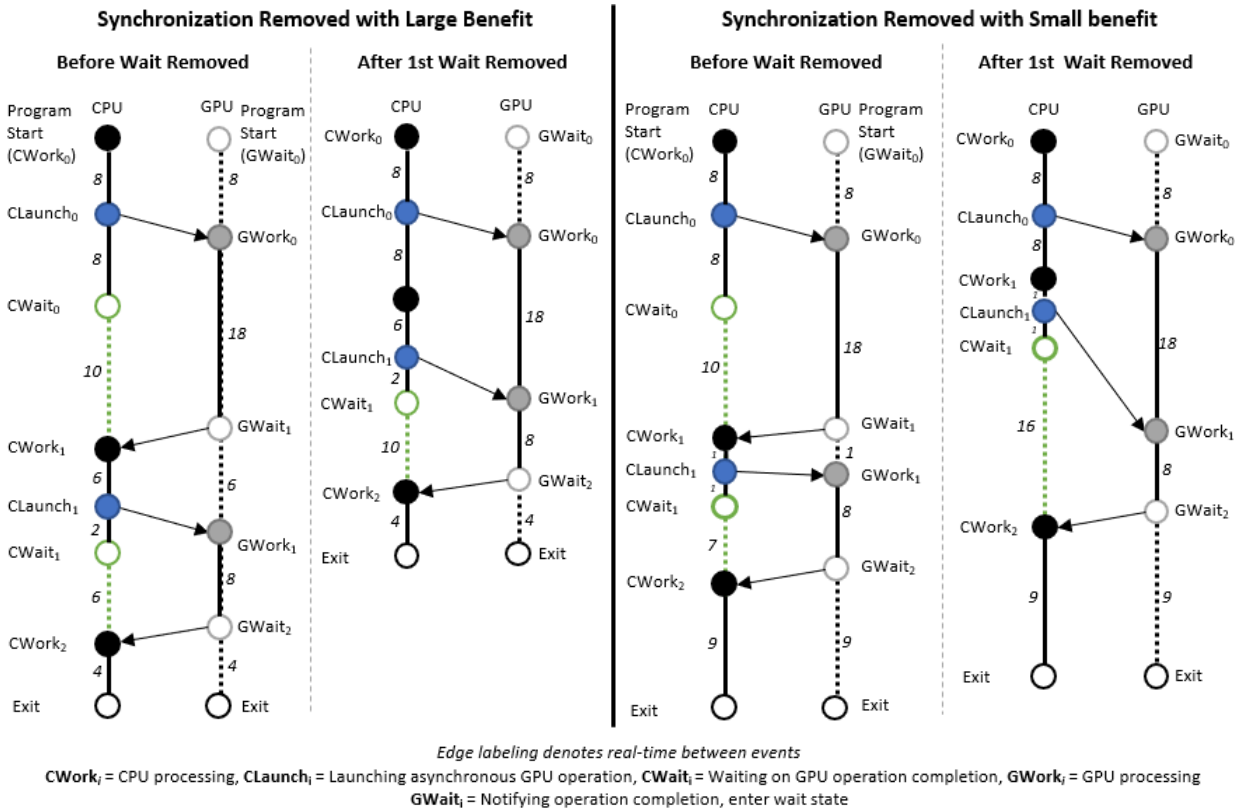

Figure 4: Example of the different outcomes from removing a problematic synchronization

or a CPU event that requests that the GPU perform work ($CLaunch$).

Edges have a label $Duration$ that denotes the real-time duration of the event. We define the functions $OutGPUEdge(N)$ and $OutCPUEdge(N)$ to obtain the out-edge from node N that ends on a node with the given processor type. There can be only one edge leading from N to a node of a given processor type.

There are three problem types that we model: unnecessary synchronization, misplaced synchronization, and unnecessary memory transfer. For unnecessary synchronizations, we model the removal of the event performing the unnecessary synchronization. To model the removal of an unnecessary synchronization from node N (of type $CWait$), we set the label of the edge to zero ($OutCPUEdge(N)_{\ Duration} = 0$). For misplaced synchronizations, we model moving the event performing the synchronization. To model moving a misplaced synchronization from a node M (of type $CWait$), we subtract the time to first use (collected in stage 4) from the current label of the edge ($OutCPUEdge(N)_{Duration} - FirstUseTime$). For an unnecessary transfer, we model the removal of the event performing the transfer. To model the removal of the transfer from a node T (of type $CLaunch$), we set the label of the edge to zero ($OutCPUEdge(N)_{Duration} = 0$). The expected benefit algorithm alters the graph based on the problem types present, calculating the expected performance improvement that is obtainable by fixing the problematic operation.

3.5.1 Expected Benefit Algorithm. Figure 5 shows the algorithm for calculating expected benefit. We assume that the graph has already been annotated with the data collected in stages 1-4. In function `ExpectedBenefit` we iterate through the graph evaluating nodes that represent problematic operations.

If a node performs an unnecessary synchronization, the function `RemoveSyncronization` on line 10 removes the synchronization and returns the expected benefit. `RemoveSyncronization` updates the duration of the next synchronization at node $NextSync$ (line 19) and sets the duration of $Node$ to zero (line 21). The removal of $Node$ results in $NextSync$ starting $Node$ duration earlier ($NextSync.STime = NextSync.STime - OutCPUEdge(Node)_{Duration}$). The duration of the synchronization operation started in $NextSync$ potentially increases due to having to wait on GPU events that started prior to $Node$ to complete. The increase of $OutCPUEdge(NextSync)_{Duration}$ can be as large as $OutCPUEdge(Node)_{Duration}$, negating any benefit. $OutCPUEdge(NextSync)_{Duration}$ is determined by the amount of GPU work remaining when $NextSync$ starts. We must estimate how the GPU graph will change when $Node$ is removed.

The removal of a synchronization does not alter the work events that are performed by the GPU, so the duration of

$GWork$ events stays the same. However, the duration of $GIdle$ events between $GWork$ events is reduced. The reduction is caused by $CLaunch$ events that take place between the nodes $Node$ and $NextSync$ having their start time reduced by $OutCPUEdge(Node)_{Duration}$. The total GPU idle time between $Node$ and $NextSync$ can contract by as much $OutCPUEdge(Node)_{Duration}$. GPU idle time cannot be negative, so the contraction of GPU idle duration is limited to the sum of the duration of GPU idle events between $Node$ and $NextSync$.

We have found that an effective estimate for the change in GPU idle duration between $Node$ and $NextSync$ can be made with only the CPU graph. With only the CPU graph, we can determine the upper bound of the change in GPU idle duration after $OutCPUEdge(Node)_{Duration}$ is set to zero. In practice, we have found that the benefit typically is close to the upper bound. On line 16, we estimate GPU Idle time to be the duration of all nodes between $Node$ and $NextSync$. This is the maximum duration that the GPU can be idle before the next synchronization. The estimated benefit is calculated on line 18 to be the minimum of the duration of the synchronization removed ($OutCPUEdge(Node)_{Duration}$) and the maximum period of GPU idle time. On line 19 we calculate the new duration of $NextSync$ by adding ($OutCPUEdge(Node)_{Duration} - EstBenefit$) to the current duration of $NextSync$.

If the node has a misplaced synchronization, the function `MisplacedSynchronization` on line 14 calculates the effect on the edge label of the node performing the synchronization if it was moved. For a misplaced synchronization at node $Node$, we model how $OutCPUEdge(Node)_{Duration}$ would change if $Node_{STime}$ was increased. $Node_{STime}$ increases by the time to first use ($FirstUseTim\ e$), the time between the end of the synchronization event and the first use of protected data collected in stage 4. Moving the synchronization forward in time results in some CPU and GPU work being moved forward in time. While the start time of some work events change, their duration still does not. The only events with durations that change are GPU idle events.

The calculation of change in expected benefit for moving a misplaced synchronization is similar to removing a synchronization. On line 25, we calculate the estimated benefit to be $Node.FirstUseTime$. This is the maximum duration that the GPU can be idle between $Node_{STime}$ and its new location ($Node_{STime} + FirstUseTime$). We calculate the new duration of $OutCPUEdge(Node)_{Duration}$ on line 26 to be the original duration subtracted by $Node.FirstUseTime$.

If a node has a unnecessary memory transfer, the function `RemoveMemoryTransfer` on line 37 calculates the effect of removing the transfer. A transfer operation consists of a CPU event of type $CLaunch$ and a GPU event of type $GWait$. The $CLaunch$ event performs setup and initiates the transfer while the $GWait$ event waits for the transfer to complete. We estimate that the expected benefit to be the duration of $CLaunch$ (line 31). The net effect is that the node's duration is set to zero (line 32).

```
1  // SumDuration([Nodes]) sums the duration of a list of nodes
2  // GetNextNode(Node) returns the node at OutCPUEdge(Node)
3  // GetNextSyncNode(Node) returns the next synchronization node
4  //             in the CPU graph after Node
5  def ExpectedBenefit(Graph):
6    for Node in Graph.ProblematicNodes:
7      if Node.Problem == UnnecessarySynchronization:
8        EstBenefit = RemoveSyncronization(Graph, Node)
9      else if Node.Problem == MisplacedSynchronization:
10       EstBenefit = MoveSynchronization(Graph, Node)
11     else if Node.Problem == UnnecessaryTransfer:
12       EstBenefit = RemoveMemoryTransfer(Graph, Node)
13
14 def RemoveSyncronization(Graph, Node):
15   NextSync = GetNextSyncNode(Node)
16   EstMaxGPUIdle = SumDuration(CPUNodesBetween(Node, NextSync,
17                   CLaunch or CWork))
18   EstBenefit = min(EstMaxGPUIdle, OutCPUEdge(Node).duration)
19   OutCPUEdge(NextSync).duration += max(0,
20                   (OutCPUEdge(Node).duration - EstBenefit))
21   OutCPUEdge(Node).duration = 0
22   return EstBenefit
23
24 def MisplacedSynchronization(Graph, Node):
25   EstBenefit = Node.FirstUseTime
26   OutCPUEdge(Node).duration = max(0,
27                    (OutCPUEdge(Node).duration - EstBenefit))
28   return EstBenefit
29
30 def RemoveMemoryTransfer(Graph, Node):
31   EstBenefit = OutCPUEdge(Node).duration
32   OutCPUEdge(Node).duration = (OutCPUEdge(Node).duration -
33                                EstBenefit)
34   return EstBenefit
35
36 def CPUNodesBetween(StartNode, EndNode, Type):
37   ret = list()
38   while (StartNode = GetNextNode(StartNode)) != EndNode:
39     if StartNode.NType == Type:
40       ret.append(StartNode)
41   return ret
```

Figure 5: The expected benefit algorithm

3.5.2 Node Groupings. In real applications, multiple problematic operations often have the same underlying cause. For example, a single line of source code or a single function might be responsible for many problematic operations. Making a single fix can result in multiple problematic operations being corrected. We group problematic nodes together to expose problems where a single fix could be applied at a single point in the program (single point), to a single function in the program (folded function), and to problematic nodes that appear in a contiguous sequence (sequence).

The single point grouping combines the expected benefit of nodes with identical stack traces that are matched by instruction address. We modify the `ExpectedBenefit` function in Figure 5 to combine the estimated benefit of nodes with the same stack trace. The stack traces for the nodes in the graph were collected in stage 2.

The folded function grouping combines the expected benefit of nodes with identical stack traces that are matched by function name. We compare stack traces by the base function name. For C++ functions, we demangling the function name and discard template parameter type information before matching. Template function calls with the same function name with instances that differ only by template parameter types often are the same function in source code. A fix to

Application Name (Version)	Application Size (Lines of Code)	Organization	Application Description	Diogenes Discovered Issues	Diogenes Estimated Benefit (% of exec)	Actual Runtime Reduction (% of exec)
cumf_als [27] (git rev: a5d918a)	5 K	IBM/UIUC	Matrix Factorization	Sync and Mem Trans	137s (10.0%)	106s (8.3%)
cuIBM [12] (git rev: 0b63f86)	36 K	Boston University	Immersed Boundary Method	Sync	202s (10.8%)	330s (17.6%)
AMG [31] (v2.14)	65 K	LLNL	Algebraic Multigrid Solver	Sync	0.34s (6.8%)	0.29s (5.8%)
Rodinia [4] (v3.1)	< 1 K	UVA	Gaussian (CUDA)	Sync	0.13s (2.2%)	0.12s (2.1%)

Table 1: Applications improved by correcting a subset of Diogenes discovered issues

a problem in the source code for the template would affect all instances. The `ExpectedBenefit` function in Figure 5 is modified in an identical manner to the single point grouping.

The sequence grouping combines the expected benefit of problematic nodes that appear in a contiguous sequence on the CPU graph. A sequence starts at a problematic node N_0 and traverses the CPU graph, ending when a node N_i is discovered that performs a synchronization that is necessary. No synchronizations need to take place in the sequence set $\{N_0, ..., N_{i-1}\}$. This property allows for the spreading of unnecessary synchronization delay across a wider timespan, increasing the number of $GWait$ events with durations that could be reduced, allowing for large unnecessary synchronization delays to be profitable corrected. Supporting sequences requires a small modification to `RemoveSyncronization` to carry forward unrealized savings ($OutCPUEdge(Node).duration$ that could not be absorbed by GPU idle time) to future nodes that may have GPU idle time that could be reduced.

4 DIOGENES

We created a prototype implementation of FFM called Diogenes that targets applications running on systems with Nvidia GPUs. Diogenes is a dynamic binary instrumentation performance tool that automates the data collection and analysis of FFM's stages, leveraging Dyninst [24] to create and insert instrumentation into the application. Diogenes is launched in a similar fashion to HPCToolkit's hpcprof and NVProf, no user involvement is necessary to advance diogenes through the stages of FFM.

Diogenes has a simple terminal-based command line interface to explore data analyzed by FFM. The results are sorted by potential benefit and then exported in the JSON format, allowing other tools the ability to access data collected by Diogenes. Diogenes runs stages 1 through 3 to separately collect performance data for problematic synchronization and memory transfer operations, combining the results in the analysis stage.

Diogenes requires the application program to be compiled with debug symbols and must be linked against a version of CUDA greater than 9.0. Our initial prototype of Diogenes targets the PowerPC 8/9 architectures with most of the development and testing taking place on Coral early access machines at Lawrence Livermore National Laboratory. However, Diogenes is not limited solely to PowerPC architectures and can be used on any architecture supported by Dyninst with minor modifications.

5 EXPERIMENTS AND DISCUSSION

We tested the effectiveness of FFM's ability to identify problematic operations and predict benefit by applying Diogenes to four real world applications: cumf_als [27] an alternating least square matrix factorization library developed at IBM and University of Illinois Urbana-Champaign, cuIBM [12] a 2D Navier-Stokes solver using the immersed boundary method developed at Boston University, AMG [31] an MPI based parallel algebraic multigrid solver developed at LLNL, and the Gaussian GPU benchmark from Rodinia [4] developed at the University of Virginia. All experiments were run on the Ray Coral early-access cluster located at LLNL. Each compute node on Ray contains a 20-core PowerPC 8-processor node with four Nvidia Pascal-class GPUs.

For each application, we used Diogenes to identify the problems present in the application, fixed the problems with the highest potential benefit, and compared the results of Diogenes to other profiling tools. In Section 5.1 we detail the problems detected by Diogenes and the fixes applied in each application. Section 5.2 compares the output of Diogenes to NVProf and HPCToolkit. In Section 5.3 we discuss the limitations of Diogenes and the FFM model.

5.1 Application Problems

Table 1 shows the problem types Diogenes discovered in each application, the estimated benefit Diogenes produced for the problems we addressed, and the actual benefit obtained for fixing the problems. In cumf_als, we corrected a single sequence of composed of 13 different problematic operations that spanned across two functions. In cuIBM, we corrected problematic synchronizations that appeared in a template function. In AMG and Rodinia, we corrected unnecessary operations that appeared at single points in the program. The estimates produced by Diogenes were 77% (cumf_als), 61% (cuIBM), 85% (AMG), and 92% (Rodinia) accurate to the real benefit obtained. The major outlier was cuIBM,

where the fix also corrected other problematic behavior not targeted by Diogenes, resulting in a much larger benefit.

cumf_als [27] is a GPU-based large matrix factorization library that uses the alternating least square (ALS) method. We ran our experiments using the GroupLens MovieLens [8] 10M input data set, a dataset containing 10 million user ratings for movies, created by the University of Minnesota with a cumf_als iteration count of 5000. Diogenes estimated that correcting a sequence containing problematic synchronization and memory transfer operations in cumf_als could result in a reduction in execution time by 11% (see Figure 6). This sequence contained 23 problematic operations spread across two functions in two different source files. 18 operations were problematic synchronizations and 5 were problematic synchronous memory transfers (with both an unnecessary transfer and synchronization).

To remove the problematic synchronizations at the `cudaFree` operations at the beginning of the sequence in Figure 6, a major rework of the structure of GPU memory handling within the application would be needed, however we wanted to avoid making large structural changes to the application. We inspected each problematic operation in Figure 6, looking for the problems that we could fix easily. The operation at entry 10 of Figure 6 was the first one we could easily fix. We then used the subsequence feature of Diogenes, which allows a user to create a sequence between any two points of an existing sequence, to generate a subsequence from entry 10 to entry 23. Figure 8 shows that the benefit that could be obtained by fixing the subsequence was 10% of execution time, close to the estimated 11% benefit from fixing the entire sequence. Note that the evaluation of the benefit of fixing this subset of operations does not require additional data collection. It can be invoked directly from the command line interface of Diogenes. We are working on ways to automate the identification of the high-impact subsequences. To properly automate subsequence generation, we need to be able to estimate the complexity of fixing the problematic behavior and weight it against the benefit that could be obtained.

The fix applied to cumf_als removed function calls performing unnecessary synchronizations and removed memory transfer calls that would repeatedly retransfer the same data to the same destination. For problematic synchronizations

```
Time Recoverable: 155.785s (11.45% of execution time)
Number of Sync Issues: 23 Number of Transfer Issues: 5
---------------
Select start/ending subsequence to get refined estimate
1. cudaMemcpy in als.cpp at line 738
2. cudaMemcpy in als.cpp at line 739
3. cudaFree in als.cpp at line 760
...
9. cudaFree in als.cpp at line 855
10. cudaFree in als.cpp at line 856
...
23. cudaFree in als.cpp at line 987
```

Figure 6: A sequence of unnecessary operations identified by Diogenes in cumf_als

at `cudaFree`, there is no asynchronous version of `cudaFree`, we could not remove the call to `cudaFree` as it would lead to not freeing memory allocated on the GPU by the application. Instead of removing the call, we moved the `cudaFree` call and its associated `cudaMalloc` call outside of the for-loop in which they were contained, resulting in memory allocation and deallocation that occurs only once instead of once per loop iteration (approximately 5000 loop iterations).

To remove a memory transfer, we need to ensure that the removal of the transfer did not result in incorrect computation when the application was used with another data set. To guard against such incorrect application behavior, we use compiler and system based methods. Our first approach is to make use of the C/C++ `const` qualifier on the variables in the removed transfers. By using the `const` qualifier, the compiler will notify (via a compile time error) if there is an attempt to write to these variables in either the CPU or GPU code. However, since a developer can still perform an unsafe cast to get around the restrictions of `const`, we also use the system `mprotect` primitive to ensure that the data cannot be modified. We allocate the variables used in the removed transfers on page aligned boundaries and use `mprotect` to write protect the variables memory pages.

cuIBM [12] is a computational fluid dynamics (CFD) application that uses an immersed boundary method (IBM), allowing fluid flows to be calculated on a cartesian grid. We ran our experiments using the lid-driven cavity with Reynolds number 5000 dataset supplied in the public source code repository for cuIBM [13] (*lidDrivenCavityRe5000*). In cuIBM, Diogenes reported that the 22% of execution time could be saved by removing problematic `cudaFree` operations (see left hand side of Figure 7). Asking Diogenes for more details on the `cudaFree` revealed that a single template function accounted for 10.8% of application execution time (see right hand side of Figure 7). The issue Diogenes identified is one that was also recently identified via manually analysis [28]. The issue was caused by the repeated (millions of times) allocation and deallocation of temporary GPU memory regions. Each deallocation performs a synchronization with the GPU that was unnecessary. The template function allocates a temporary GPU data region via the Thrust [22] parallel algorithms library and frees it on exit. The result is many calls to `cudaFree` that synchronize with the GPU. To avoid this problem, we wrote a simple memory manager that reuses temporary GPU data regions on subsequent calls to the function. We modified cuIBM to use this method instead of allocating storage via Thurst. The fix resulted in the synchronization being eliminated. However, the fix also eliminated over 2 million `cudaFree` and `cudaMalloc` operations, providing additional benefit.

AMG [31] is a parallel algebraic solver for linear systems, specializing in 3-dimensional problems on unstructured grids. We ran our experiments using the ij matrix benchmark contained within AMG. In AMG, Diogenes estimated that 6.8% of execution time could be saved by fixing a problematic

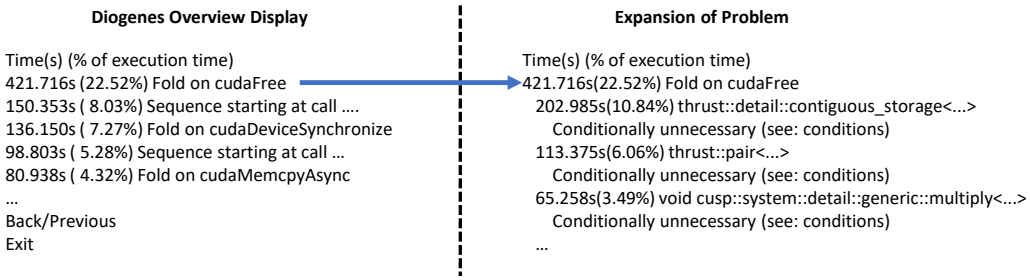

Figure 7: Diogenes overview of problematic operations (left) and the expansion of problems at `cudaFree` (right) for cuIBM

synchronization at a `cudaMemset` operation. `cudaMemset` performs a synchronization only when it used on a unified memory address (a memory address accessible by both the CPU and GPU). Since the memory pages being set were already located in CPU memory, we replaced `cudaMemset` call with a normal C `memset` operation.

Rodinia [4] is a benchmark suite for heterogeneous computing designed to study the performance effect new computing architectures have on a variety of well known algorithms. We ran our experiments using Rodinia's Gaussian GPU benchmark. In Rodinia, Diogenes estimated that 2.2% of execution time could be saved by fixing a problematic synchronization at a `cudaThreadSy nchronize` operation. There were no other operations that had potential benefits greater than 1% of execution time. We fixed the issue by commenting out the `cudaThreadSynchronize` call.

5.2 Comparison to NVProf and HPCToolkit

Table 2 shows a comparison of the expected-benefit provided by Diogenes against the results produced by NVProf [21] and HPCToolkit [17]. We compare the call times reported by each tool and, in Diogenes' case, the expected benefit for the CUDA functions called. The entries are sorted by the order in which they appear in the summary generated by NVProf.

NVProf and HPCToolkit show similar results while Diogenes differs significantly for synchronizations and memory transfer operations. An example can be seen in the profiling results for cumf_als. NVProf and HPCToolkit reported that the function `cudaDeviceSyn chronize` executed for 745 and 628 seconds respectively. Diogenes reported that only 1 second of the execution time could be saved if you removed the calls to `cudaDeviceSynchronize`. We verified that there was no impact on the execution time of cumf_als when only the `cudaDeviceSynchronize` calls were removed. Other significant differences between Diogenes and other tools for synchronization and memory transfer calls can be seen in `cudaMemcpyAsync` (in cuIBM), `cudaThreadSynchronize` (in Rodinia), `cudaDeviceSyn chronize` (in cumf_als), `cudaMemset` (in AMG), `cudaFree` (in cumf_ als), and `cudaMemcpy` (in cumf_als).

Unlike NVProf and HPCToolkit, Diogenes does not collect performance data on calls that do not contain a problematic synchronization or memory transfer operation. We collect no data on calls such as `cudaMalloc` and `cudaLaunchKernel` because they do not perform a synchronization or a memory transfer. The calls we collect data on is determined during stage 1 of the FFM model when we identify what calls are performing a synchronization or memory transfer. In the future, if these calls become synchronous or perform memory transfers, Diogenes will collect data on them automatically.

It should be noted that we were unable to run NVProf on cuIBM due to a crash of NVProf during profiling. We tried several different versions of NVProf between CUDA version 9.1 and 9.2, all of which crashed before producing a result. The crash was likely caused by the large number of cuda calls that take place during cuIBM's execution (Diogenes collected data on > 75 million cuda function calls). For HPCToolkit, the reported percentage of execution time in Table 2 is lower than expected for the applications cuIBM and cumf_als. cumf_als has an uninstrumented execution time of 1360 seconds but HPCToolkit reports that `cudaDeviceSynchronize` took 628 seconds and consumed 24.5% of execution time (when it should be closer to 40%). We are still investigating why this discrepancy exists.

```
Time Recoverable In Subsequence: 137.136s
(10.08% of execution time)
10. cudaFree in als.cpp at line 856
11. cudaDeviceSynchronize in als.cpp at line 877
12. cudaFree in als.cpp at line 878
...
22. cudaFree in als.cpp at line 986
23. cudaFree in als.cpp at line 987
```

Figure 8: The estimate of benefit reported by Diogenes for fixing a subsequence of the operations in Figure 6.

5.3 Limitations of Diogenes

While we have seen success using Diogenes, there are some limitations. Given that Diogenes runs an application multiple times, it performs best when the execution pattern of the application does not change dramatically between runs with the same inputs. While Diogenes can tolerate small changes in behavior between runs, applications with large changes in behavior could result in missed problematic behavior.

Application Name	Operation	NVProf Profiled Time (% of exec, pos in profile)	HPCToolkit Profiled Time (% of exec, pos in profile)	Diogenes Estimated Savings (% of exec, pos in profile)
cumf_als	cudaDeviceSynchronize	745s (52.0%, 1)	628s (24.5%, 1)	1s (0.07%, 3)
	cudaFree	275s (18.7%, 2)	258s (10.1%, 2)	214s (15.73%, 1)
	cudaMalloc	218s (17.3%, 3)	230s (9.1%, 3)	-
	cudaMemcpy	158s (11.8%, 4)	119s (4.7%, 4)	30s (2.2%, 2)
cuIBM	cudaFree	Profiler Crashed	447s (12.3%, 1)	421s (22%, 1)
	cudaLaunchKernel		395s (12.1%, 2)	-
	cudaMalloc		382s (10.8%, 3)	-
	cudaDeviceSynchronize		170s (4.8%, 4)	136s (7.2%, 2)
	cudaMemcpyAsync		163s (4.4%, 5)	80s (4.32%, 3)
	cudaFuncGetAttributes		154s (4.2%, 6)	-
	cudaStreamSynchronize		52s (1.4%, 7)	4s (0.23%, 4)
AMG	cudaFree	0.937s (18.7%, 1)	0.392s (3.3%, 2)	0.316s (6.34%, 2)
	cudaMemset	0.813s (16.3%, 2)	0.577s (6.0%, 1)	0.343s (6.87%, 1)
	cudaMallocManaged	0.157s (3.1%, 3)	0.069s (0.7%, 4)	-
	cudaStreamSynchronize	0.133s (2.6%, 4)	0.129s (1.3%, 3)	0.071s (1.41%, 3)
Rodinia	cudaThreadSynchronize	6.05s (94.9%, 1)	5.01s (75.7%, 1)	0.13s (2.2%, 1)
	cudaMemcpy	< 0.01s (0.9%, 2)	0.07s (1.2%, 2)	0.06s (0.9%, 2)
	cudaFree	< 0.01s (0.01%, 3)	< 0.01s (0.2%, 3)	< 0.01s (0.04%, 3)

Table 2: Comparison of cuda function call profiling results between Diogenes, HPCToolkit, and NVProf

The overhead of running Diogenes is significantly higher than that of other performance tools. The multiple runs and the use of high cost instrumentation result in data collection times between 8x (cumf_als) and 20x (cuIBM) of the applications original execution time. While the cost is high of running Diogenes, the automated nature of the tool and the targeted feedback Diogenes provides can save programmer time as compared to identifying these problems manually.

Diogenes has a limited ability to analyze applications using CUDA's unified memory. Unified memory provides a single virtual memory address space accessible by any CPU or GPU device on the system, removing the need to explicitly transfer data between the devices. The transfer of data between CPU and GPU physical memory still takes place but is automatically performed by the GPU device driver. Though the transfer of data is automatic, problematic transfers can still occur. However, unlike a normal memory transfer, the source and destination of a unified memory transfer are not known until after the transfer completes. The notification of transfer completion is not immediate. The result is that the data transfer could be modified before a hash could be calculated and the presence of a problematic transfer would be hidden. We have indirectly detected issues with unified memory transfers in AMG (cudaMemset issue) and we are looking at methods to expand Diogenes to directly detect problems with unified memory transfers.

6 CONCLUSION

We have presented the FFM model that automates the identification of unnecessary/inefficient synchronization and memory transfer operations in GPU programs. FFM gives targeted feedback on what problems exist in the application and what the benefit would be if the problem were corrected. FFM is not reliant on vendor supplied performance data collection frameworks for data collection, instead FFM uses binary instrumentation to directly capture and time events such as synchronizations. The multi-stage/multi-run method of data collection allowed FFM to collect performance data that would otherwise be missed or is too costly for other performance tools to collect.

We created a new analysis model that uses the data collected by FFM to accurately identify problematic operations. The analysis model groups problematic operations together to identify problems where a single fix could be applied and gives an estimate of the benefit of fixing the problems. The prototype implementation of FFM, Diogenes, was able to identify performance issues in four real world applications. Diogenes was able to provide accurate feedback (around 77% combined accuracy across all applications) on what the benefit would be if the problem were fixed. Using Diogenes we were able to improve the performance of these applications by as much as 17%.

The problems identified by Diogenes in the applications we tested typically had a similar underlying cause with a common remedy that could be applied to correct the problem. The existence of a common underlying cause along with a common remedy used to correct a problem signals that they may be automatically correctable if the cause and remedy can be automatically identified. An automated method would be able to correct issues that a typical user may not be able or may not want to correct, such as issues that occur in closed source binaries or those that offer low benefit. Problematic synchronizations caused by inefficient memory management and improper use of asynchronous memory transfers are of particular interest given their high impact on performance in the applications we tested. We are working to refine the process of automatic correction and to integrate this capability into Diogenes in the near future.

7 ACKNOWLEDGEMENTS

This work is supported in part by the Department of Energy under contacts 4000164398 and 4000151982 from Oak Ridge National Labs, National Science Foundation grant ACI-1449918, Lawrence Livermore National Lab contracts B617863 and B634650, and a grant from Cray Inc.

REFERENCES

[1] T. E. Anderson and E. D. Lazowska. 1990. Quartz: A Tool for Tuning Parallel Program Performance. In *The 1990 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '90)*. Boulder, Colorado, 115–125. https://doi.org/10.1145/98457.98518

[2] R. Bell, A. D. Malony, and S. Shende. 2003. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *EuroPar Conference on Parallel Processing (EuroPar '03)*, Harald Kosch, László Böszörményi, and Hermann Hellwagner (Eds.). Berlin, Heidelberg.

[3] C. Beyer, J, E. J. Stotzer, A. Hart, and B. R. de Supinski. 2011. OpenMP for Accelerators. In *the 7th International Workshop on OpenMP ((IWOMP '11))*. Chicago, IL. https://doi.org/10.1007/978-3-642-21487-5_9

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) ((IISWC '09))*. IEEE Computer Society, Austin, TX. https://doi.org/10.1109/IISWC.2009.5306797

[5] B. R. Coutinho, G. L. M. Teodoro, R. S. Oliveira, D. O. G. Neto, and R. A. C. Ferreira. 2009. Profiling General Purpose GPU Applications. In *the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '09)*. Sao Paulo, Brazil, 7. https://doi.org/10.1109/SBAC-PAD.2009.26

[6] M. Geimer, F. Wolf, B. JN. Wylie, E. Ábrahám, D. Becker, and B. Mohr. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* Vol 22, Num 6 (2010), 702–719.

[7] M. Gerndt, K. Fürlinger, and E. Kereku. 2005. Periscope: Advanced Techniques for Performance Analysis.. In *the 2005 International Conference on Parallel Computing (PARCO '05)*. Malaga, Spain.

[8] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19 pages. https://doi.org/10.1145/2827872

[9] J. K. Hollingsworth and B. P. Miller. 1993. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. In *The 7th International Conference on Supercomputing (ICS '93)*. Tokyo, Japan, 185–194. https://doi.org/10.1145/165939.165969

[10] J. K. Hollingsworth and B. P. Miller. 1994. *Slack: A New Performance Metric for Parallel Programs*. Technical Report. University of Wisconsin - Madison. https://doi.org/10.13140/RG.2.2.27600.97285

[11] A. Knüpfer, C. Rössel, D. A. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf. 2011. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *the 5th International Workshop on Parallel Tools for High Performance Computing*. Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-31476-6_7

[12] S. Layton, A. Krishnan, and L. A. Barba. 2011. cuIBM - A GPU-accelerated Immersed Boundary Method. In *the 23rd International Conference on Parallel Computational Fluid Dynamics ((ParCFD '11))*. Barcelona, Spain.

[13] S. Layton, A. Krishnan, and L. A. Barba. 2019. *cuIBM Source Code Repository* (commit 0b63f86 ed.). https://github.com/barbagroup/cuIBM

[14] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. 2000. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *2000 ACM/IEEE Conference on Supercomputing (SC '00)*. Dallas, TX. https://doi.org/10.1109/SC.2000.10052

[15] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. 2011. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *the 2011 International Conference on Parallel Processing (ICPP '11)*. Taipei City, Taiwan, 10. https://doi.org/10.1109/ICPP.2011.71

[16] A. D. Malony, S. Biersdorff, W. Spear, and S. Mayanglambam. 2010. An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using CUDA. In *the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, Tsukuba, Ibaraki, Japan. https://doi.org/10.1145/1810085.1810105

[17] J. Mellor-Crummey, R. Fowler, and D. Whalley. 2001. Tools for Application-oriented Performance Tuning. In *Proceedings of the 15th International Conference on Supercomputing (ICS '01)*. Sorrento, Italy. https://doi.org/10.1145/377792.377826

[18] B. Mohr and F. Wolf. 2003. KOJAK: A tool set for automatic performance analysis of parallel programs. In *The 2003 European Conference on Parallel Processing (EuroPar '03)*. Klagenfurt, Austria.

[19] Nvidia. 2016. *CUDA Compiler Driver NVCC - Reference Guide* (8.0 ed.).

[20] Nvidia. 2018. *The CUDA Profiling Tools Interface* (9.2 ed.).

[21] Nvidia. 2018. *The Nvidia CUDA Profiler Users' Guide* (9.2 ed.).

[22] Nvidia. 2018. *The Thrust Quick Start Guide* (9.2 ed.).

[23] V. Pillet, J. Labarta, T. Cortes, and S. Girona. 1995. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of the 18th Technical Meeting on Transputer and Occam Developments (WoTUG-18)*. Manchester, England.

[24] Paradyn Project. [n. d.]. *Dyninst: Putting the Performance in High Performance Computing*. http://www.dyninst.org

[25] 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 (CGO 2018)*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/3168831

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

[27] W. Tan, S. Chang, L. Fong, C. Li, Z. Wang, and L. Cao. 2018. Matrix Factorization on GPUs with Memory Optimization and Approximate Computing. In *Proceedings of the 47th International Conference on Parallel Processing ((ICPP '18))*. ACM, Eugene, OR, USA, Article 26, 10 pages. https://doi.org/10.1145/3225058.3225096

[28] B. Welton and B. P. Miller. 2018. Exposing Hidden Performance Opportunities in High Performance GPU Applications. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '18)*. Washington, D.C., 301–310. https://doi.org/10.1109/CCGRID.2018.00045

[29] S. Wienke, P. Springer, C. Terboven, and D. an Mey. 2012. OpenACC: First Experiences with Real-world Applications. In *the 18th International Conference on Parallel Processing ((Euro-Par '12))*. 12. https://doi.org/10.1007/978-3-642-32820-6_85

[30] C E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. 2000. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 50.

[31] U. Yang. 2018. AMG: Algebraic Multigrid Benchmark. https://github.com/LLNL/AMG. (2018).