The Feed-Forward Measurement Model: A Performance Measurement Model for CPU/GPU Architectures

by

Benjamin R. Welton

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 03/12/2020

The dissertation is approved by the following members of the Final Oral Committee:

Barton Miller, Professor, Computer Sciences
Michael Swift, Professor, Computer Sciences
Matthew Sinclair, Assistant Professor, Computer Sciences
Dan Negrut, Professor, Mechanical Engineering
Todd Gamblin, Researcher, Lawrence Livermore National Laboratory

to Anita, Selene, Artemis, and Louis

ACKNOWLEDGMENTS

On the journey to completing my PhD, I have had help and support from many in both refining my ideas and bringing them to fruition.

I would like to thank my advisor Barton Miller for his assistance and guidance not only on the work presented in this document but also on the work of Mr. Scan. Without Bart's guidance and his push for high impact research on real world problems, all of the work conducted during my PhD would not exist. The insights on both GPU performance and density-based clustering we discovered are already creating an impact well beyond what I anticipated and without Bart's assistance these achievements would not have been possible.

I would also like to thank the other members of my committee. Todd Gamblin for his suggestions throughout the years. His assistance in helping me find programs to test the feed-forward measurement model on was vital and gave me invaluable insights on how to improve my techniques. Michael Swift for his help early on through the comments and suggestions he made on how to improve the techniques. These comments helped me come up with new ideas to bypass problems and potential limitations of this research. Dan Negrut for access to his GPU cluster, Euler, that allowed us to do the initial testing of Mr. Scan. Without access to Euler, Mr. Scan may not have been created and the general GPU performance problems we faced developing Mr. Scan would have remained unexposed.

A special thanks to Lawrence Livermore National Laboratory and Oak Ridge National Laboratory for their generous assistance over the last decade. Their compute resources and the expertise of their staff were instrumental in the development of the research works created as part of my PhD.

I have been very fortunate to have worked with some really fun, interesting, and helpful people throughout my time in graduate school. Xiaozhu

Meng helped me survive my long PhD journey and his assistance with Dyninst was invaluable in helping me complete my PhD. Matt Legendre for his assistance with fleshing out my ideas and his support during my time at LLNL. I am extremely grateful for the assistance given by the other Paradyn group members, both past and present (Mike Brim, Bill Williams, and many others) as well as the many external collaborators that have assisted me throughout the years.

Thank you to my family for their emotional support throughout my PhD and dealing with my eccentric personality over the last 30+ years. I must thank my grandfather Roger Welton, without his support of my family during a trying time I likely would not have been able to attend college at all. One of my only regrets in life was not completing my PhD before his passing so that he could celebrate the achievement with me that he made possible.

Thank you to all the friends who supported me throughout my PhD. I will always cherish the friendship and time spent with Seth and Ashia Pollen, Peter Ohmann, Tyler Harter, and the many others who I met during my PhD.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

From mobile computing to the largest leadership-class supercomputers, many-core accelerators are relied upon to provide the computational capabilities necessary to make today's applications possible. Machine learning, image processing, n-body simulations, and a host of other applications are increasingly reliant on the computational capability provided by many-core accelerators to achieve the performance necessary to target real-world problems. For modern applications, especially those running on leadership-class supercomputers where the number of GPUs can outnumber traditional CPUs three to one, effective exploitation of many-core compute resources is a must to achieve high efficiency. Effectively exploiting the additional parallelism provided by many-core accelerators requires developers identify where accelerator parallelization would provide benefit and ensure efficient interaction between the CPU and accelerator. While these issues appear straightforward and well understood, we have found that significant untapped performance opportunities still exist even in well-studied, heavily optimized, real world applications created by experienced developers. By addressing hidden performance opportunities, we were able to reduce execution time by up to 87% for the applications we have tested.

In this dissertation, we develop a new performance measurement and modeling technique called the feed-forward measurement model (FFM) that exposes previously hidden performance opportunities and delivers actionable feedback to developers on the potential benefit if the problem were corrected. We first explore a set of real-world applications to identify hidden performance opportunities common among them, developing techniques that tools can implement to detect their presence. FFM refines and expands these detection techniques by including a model that can estimate the performance benefit of fixing these problems. We have cre-

ated a performance tool called Diogenes that implements FFM to test its effectiveness on real world applications. The result was the discovery of problems that, when fixed, reduced execution time by up 17% in applications we tested. Last, we expanded FFM to give guidance on how to remedy the problems it identified and the ability to automatically apply remedies to the program. By automatically correcting problems, FFM was able to reduce execution time of tested applications by up to 43%.

1 INTRODUCTION

As many-core accelerators have become standard on high performance computing platforms, developers have had to adapt their applications to exploit the additional parallelism afforded by many-core architectures. The adaptation process begins with the identification of code suitable for parallelization, the writing of an efficient many-core parallelization of that code, handling the interaction between the CPU and many-core device, and ends with the integration of the new many-core component into existing CPU code. Developers often further optimize their applications by iterating over this process many times.

When viewed in isolation, each stage of the adaptation process appears straightforward and well understood. However, when adapting and tuning real-world applications, the identification of performance opportunities within each stage becomes increasingly difficult. The sheer size and complexity of real-world code bases, which can number in the hundreds of thousands of lines of code or more, makes manual identification of performance opportunities increasingly difficult. For instance, a synchronization can be hidden from source code analysis when it is hidden behind many levels of indirection or located in a closed-source binary. The code bases of real-world applications are also frequently changing the result of these changes can make once efficient interactions identified by previous optimization passes of the application inefficient. The reliance of developers on many-core accelerated libraries to add functionally to their applications increases the difficulty of optimization further. These libraries are efficient internally but when combined with other many-core accelerated codes create a undesirable interaction patterns between the CPU and the accelerator. Real-world applications also typically support many different many-core accelerator types (such as OpenMP, CUDA, and Phi). Assumptions made by application developers on usage of these

accelerator types can lead to bad behavior when they are not true (such as the developer assuming that multiple modes of acceleration will not be used together). The end result of the increased complexity makes seeming straight forward performance opportunities difficult to identify in practice by skilled developers.

Developers turn to performance tools to help unlock the hidden performance opportunities present in their programs. Developers typically use a performance tool such as HPCToolkit [56], TAU [52, 82], and NVProf [66] to help identify potentially problematic behavior. However, the help existing tools provide is limited by gaps in the performance data that they collect, including:

1. Performance data is not recorded for all GPU operations. Not all occurances of GPU operations, such as synchronizations, have performance data collected for them. This leaves the tool and the developer unaware of the occurrence of these operations during program execution.

2. Incomplete performance data is recorded for some GPU operations. Only a partial record of operations are recorded for GPU operations that during the course of their execution perform other GPU operations.

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 problematic operations is considered to be too costly.

When existing tools can identify and properly record the behavior of a GPU operation, the information they provide to the developer is often not directly actionable. 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 [99] showed, resource consumption was not always a good predictor of the obtainable benefit. When a point of high resource consumption is identified by a tool, a detailed manual analysis of the operation is still required to determine if the operation is problematic. The result is that programmers spend time identifying problematic operations that produce limited benefit when they are fixed, while missing others that might provide significant improvements to performance.

The developers work is not complete after they have identified a problem. After discovering a problem, the developer must determine how to remedy the problem. This process typically requires another round of manual analysis to determine the underlying cause of the problem. Problems can be caused by a single operation or may be a component of a larger construct that exhibits a problem spanning over many operations. If a larger construct exhibiting a problem is present, the developer could select a remedy that would fix the larger problem seeing improved benefit over fixing a single operation. The lack of guidance on what remedy to select can result in the developer missing the presence of larger problems resulting in the selection of the wrong remedy, lowering performance benefit or impacting program correctness.

Finally, after a developer has identified the correct remedy for a problematic operation, the developer is still responsible for applying the remedy to the program. Applying the remedy manually to the source code of an application can require significant effort from a developer. In some

cases, the problem may be in a closed source application where the developer does not have access to the source code to fix the problem. In practice, a developer tends to fix only the easiest of the most problematic operations discovered, leaving harder to fix problems with significant performance benefit untouched.

1.1 Techniques and Contributions

The key contribution in this dissertation is a performance data collection and analysis technique called the *Feed-Forward Measurement Model* (FFM). The primary focus of FFM was to create a technique capable of delivering actionable feedback for problematic operations in programs. FFM is able to automatically identify problematic operations, provide an estimate of benefit if the problem were corrected, provide remedies for the problems identified, and automatically apply remedies for common problems seen in applications.

We start our discussion on FFM by first describing the hidden performance opportunities that we manually detected in GPU applications and their effect on application performance. We give an overview of the FFM model, the hidden performance problems that FFM targets, and FFM's multi-stage/multi-run approach to data collection and analysis. We then go into detail on the performance model used in FFM to identify what operations are problematic and to generate an estimate of expected benefit if the problems were corrected. We end our description of the FFM model by describing two extensions to the FFM model to allow for the automatic identification of remedies to problematic operations and the automatic application of remedies to binary codes. We then describe the construction and challenges that we faced when creating *Diogenes* the performance tool that implements the FFM model.

1.1.1 Exposing hidden performance opportunities in GPU applications

We investigated performance problems that are often missed in real-world parallel applications [91]. We identified four prominent performance problems in a set of five real world scientific applications that, when manually corrected, resulted in a reduction of execution time by up to 87%. These four issues were: unnecessary synchronizations between the CPU and GPU, duplicate memory transfers between the CPU and GPU, unnecessary Just-In-Time compilation of GPU code, and missed GPU parallelization opportunities.

We focus our efforts on building techniques that target unnecessary synchronizations and duplicate memory transfers. These problems were the two most prominent in terms of performance impact and occurrence in the GPU applications we have tested. In Chapter 3, we discuss the four hidden performance opportunities, their effect on application performance, why they were hidden from other performance measurement techniques, and techniques that could be used to identify their presence in applications.

1.1.2 The Feed-Forward Measurement Model

The *Feed-Forward Measurement Model* (FFM) is a technique that pairs a multi-stage/multi-run approach to performance measurement with a new performance model that identifies problematic operations and generates an expected benefit for fixing the problematic operations [92]. The multi-stage/multi-run approach to performance measurement allows for the capture of previously uncollectable measurement data, giving the performance model the measurements necessary to discover hidden performance problems.

The principal idea behind FFM's multi-stage/multi-run approach for collecting performance measurements is that the insertion of instrumen-

tation into an application and the performance data that is collected is guided by the application's behavior. The application's 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.

We present an overview of the FFM technique in Chapter 4.

1.1.3 The Performance Model of FFM

Generating an estimate of expected benefit with reasonable accuracy requires that we understand the effect a correction to a problematic operation will have on total application execution time. The actual benefit obtained from correcting a problematic operation is impacted by the duration of the problematic operation corrected and the effect the correction has on the operations that remain. We created a performance model capable of modeling the behavior of the correction applied to fix problematic operations. The model takes into account how the applied fix alters the behavior of the remaining unchanged operations in the program. By modeling the effect that the correction has on the unchanged operations, we can determine the overall effect that a correction would have on total application execution time.

We describe the performance model of FFM, the output generation of FFM, and our experiments with with the *Diogenes* implementation of the performance model in real-world applications in Chapter 5.

1.1.4 Automatic Remedy Identification

Automatic remedy identification is an extension to the FFM model that gives guidence on the type of problem present at a point in the program and how to correct that problem [93]. This feature gives FFM the ability to identify if a problem was caused by single operation or if it is a com-

ponent of a larger construct that exhibited a problem that FFM identified. An example of a larger construct is a frequently occurring unnecessary synchronization caused by a memory free operation (such as `cudaFree`) that could indicate that a larger memory management problem is present. If larger constructs exhibiting this problem could be identified, it would result in the elimination of memory allocation and free operations, significantly increasing the potential performance benefit.

FFM was extended with capability of identifying two common larger constructs seen within an applications that exhibit a problem: unnecessary synchronizations caused by memory management issues and unnecessary synchronizations caused by synchronous memory transfers. For each of these problems, we identify the remedy that should be employed to fix the problem, including the non-synchronizing operations that need to be changed.

We describe the extensions made to FFM to support automatic remedy identification and our experiments with remedy identification in Chapter 6.

1.1.5 Automatic Correction of Problematic Synchronizations

Fixing problems identified by FFM can require a significant restructuring of application code or the modification of closed-source binaries. Developers sometimes are left with a tough choice of leaving these issues unresolved or potentially spending significant effort refactoring their code. If they choose to address the issues, the benefit they get may not have been worth the effort they place into fixing the problem. The high cost of developer time to fix problematic operations in combination with the potential risk of limited performance benefit results in the choice being made to not address these issues.

We extended the FFM model to automatically apply corrections to the two common larger constructs exhibiting synchronization problems identified by automated remedy identification. The automated correction modifies the binary at application startup to remove problematic behavior, allowing for remedies to be applied to applications even if the problem appears in a closed-source binaries such as libraries provided by the GPU vendor.

We describe the extensions to the FFM model to support automatic correction and our experiments with automatic correction in Chapter 7.

1.1.6 Diogenes: A Tool that Implements FFM

We have implemented the FFM model in a tool we call *Diogenes*. 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 [66] and CUPTI [65]) 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. We use Diogenes to evaluate the effectiveness of FFM on real world applications. Diogenes was tested on a set of four real world scientific applications to identify and correct problems, reducing their execution time by up to 43% automatically.

The techniques used in FFM touch on topics in four areas of research: performance measurement tools, profile-guided/feedback-driven optimization, and autotuners. We describe the contributions from previous work in these areas, the benefits and limitations of these contributions, and how FFM differs to solve limitations seen in existing techniques.

2.1 Performance Measurement Tools

We benefit from a long history of research in the area of performance measurement tools. Performance measurement tools have been created to help tune applications to run on a variety of platforms. Tools such as Gprof [37], OProfile [24], AMD uProf [2], and Intel VTune [10] focus on providing performance measurement and analysis to applications running on a single machine. The initial work on single machine performance measurement tools led to the development of techniques capable of profiling parallel applications. Parallel performance measurement tools such as Faust [38], HPCToolkit [56], Jumpshot [98], KOJAK [58], Paradyn [57], Paraver [74], Periscope [35], Quartz [5], Scalasca [34], ScoreP [47], and Vampir [46], extended single machine performance measurement techniques to provide scalable methods to collect measurement data from many processes and developed new analysis techniques capable of processing this data.

With the introduction of GPU accelerated applications, new performance measurement techniques were needed to support problems specific to these applications. Existing tools were modified and new tools created to detect GPU idleness [26, 47, 53, 54, 66], CPU idleness waiting on GPU completion [26, 47, 54, 66], warp occupancy [26, 54, 66], GPU cache behavior [54, 66], on-device synchronization issues [26, 66], missed parallelization opportunities [6, 9, 15, 44, 61, 68], and workload balance

between accelerators on the same node [22].

The performance tools developed to support GPU applications share a common structure with their parallel and single machine performance measurement tool ancestors in that they typically operate with a single stage of instrumentation performed on a single run of the application to collect measurement data. An analysis is then performed on the collected data to produce output for a developer to view. Where GPU performance tools differ from their ancestors is their reliance on vendor-supplied black box performance measurement collection frameworks for data collection. The measurements that can be collected, their completeness, and their accuracy often are controlled by the GPU vendor.

While this common structure has helped to produce tools that can help to find performance issues in applications, there are problems that are hard to detect even with such tools. In this section, we describe the benefits and limitations that a single stage instrumentation structure imposes on tools, how the reliance on vendor-supplied black box performance measurement collection frameworks can result in the incomplete collection of measurement data, and an overview of notable analysis methods used by GPU performance measurement tools.

2.1.1 Single Stage Instrumentation

Most existing tools are structured such that the instrumentation inserted and the type of performance data collected is static for a single run of the program. The instrumentation is set before execution and is not adjustable during execution.

The instrumentation that a single-stage/single-run performance measurement tools inserts can be fixed by the tool maker [6, 9, 15, 16, 37, 44, 61, 68] or adjustable by the user [2, 10, 26, 46, 47, 53, 54, 56]. Fixed instrumentation tools have the inherent limit that they cannot adjust the performance

data they collect; users who need additional performance data must rerun the application with another tool to collect that data.

Adjustable performance measurements tools allow the user some control over the performance data collected. The user can control the types of measurements that are taken and in some cases the locations measurement instrumentation is inserted. However, user adjustable tools cannot collect measurements on everything at the same time and thus the user must still choose what to collect. A user is forced to guess what measurements might be useful, run the program to collect those measurements, and likely rerun the program to collect new measurements. Getting effective feedback from single-stage/single-run tools requires the user to manually manage multiple stages of instrumentation taken over multiple runs from different performance tools.

The notable exceptions to the single stage tool structure are Paradyn [57], NVProf [66], and the Nvidia Visual Profiler [66] (NVVP). Paradyn adds and removes instrumentation during a single run of an application. This dynamic 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, an operation that is impactful can be missed if the operation completes before Paradyn determines that the operation is important.

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.

NVVP extends the functionality of NVProf to combine multiple single-stage/single-run instrumentation approaches into a tool with a graphical

interface where the user can select from different analyses, one such analysis is *dependency analysis* (described in Section 2.1.3). NVVP automatically launches the application once per analysis, to collect the data needed to perform that analysis. The output of each analysis is aggregated into a graphical interface that allows the user to browse the output produced by each analysis. The main advantage of this approach is that it lessens the need for the user to manually manage multiple runs of the program to collect data for some problem types. However, NVVP is still limited to a single-run to collect the data needed for a single analysis. If data from multiple runs is necessary, the user must manually collect and manage that additional data.

2.1.2 Black Box Performance Measurement Collection Frameworks

The use of accelerators such as GPUs by applications has changed the ways that performance tools collect measurement data. 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 example we see black box performance data collection interfaces for Nvidia GPUs [65], Google TPUs [36], and Intel GPUs [10]. However, AMD's performance collection frameworks and GPU driver are both open source [3].

For Nvidia GPUs, GPU profiling and tracing tools rely on the CUPTI performance data collection framework [65]. CUPTI reports when a call is made to the vendor-supplied library (libcuda) that interfaces with the GPU driver and to collect performance counter data from the GPU. During the

course of our research, we have discovered that CUPTI does not account for all CPU/GPU synchronization operations or calls made to libcuda.

For all but four functions in libcuda that perform CPU/GPU synchronization operations, CUPTI does not provide information on the CPU/GPU synchronization operations that take place. The unreported operations include implicit and conditional synchronization operations. Implicit synchronization occurs as a side effect to another operation such as a memory transfer (e.g, `cuMemcpy`). Conditional synchronization occurs when certain arguments are supplied to a GPU API call. For example, `cuMemcpyAsync` performs an unreported synchronization when a device-to-host memory transfer is performed to a CPU memory address not allocated via `cuMemAllocHost`. These behaviors are not documented in some cases and are not reported by CUPTI. In addition, the operations that perform synchronizations may be subject to change based on driver version.

Out of approximately 450 API functions, we found that CUPTI only generates synchronization timing information for the explicit synchronization operations `cuStreamSynchronize` and `cuCtxSynchronize`. The vendor claims that there are two additional synchronization operations involving CUDA events that are reported but did not provide further detail.

In certain circumstances, CUPTI does not provide information on calls made to libcuda and the operations they performed. Libcuda contains a significant proprietary non-public interface that is used by Nvidia-created libraries like cuBLAS. This interface can perform all the operations available to the public interface, but these operations are not tracked by CUPTI. 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 libcuda's public API if they are called from other Nvidia-created libraries.

The result of having these untracked libcuda calls is that the tools built using CUPTI provide incomplete performance data to the user. While we hope that these problems will be fixed in future versions of CUPTI, it proved insufficient for our needs. We developed techniques to instrument the internal functions of the GPU user space driver using binary instrumentation to capture when operations such as synchronization take place, allowing us to capture and time the synchronization delay of implicit, conditional, and non-public API synchronizations.

CUDAAdvisor [81] 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 graph 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. Vendor-supplied instrumentation frameworks were insufficient because they could not collect the fine-grained GPU trace data necessary to perform the analysis. Our research targets a different set of problems than those of CUDAAdvisor, though we also rely on binary modification for instrumentation. Note that we focus on the collection of fine-grained details of operations performed on the CPU instead of the GPU.

2.1.3 Analysis Methods

Early work in GPU performance measurement tools focused on the collection of performance counters from the GPU and the collection of basic statistical information about the functions called in GPU vendor-supplied libraries. This information often is presented to the developer by adapting existing techniques. For example, the flat profile and call graph presentations pioneered in the CPU performance tool Gprof [37] are the two most common among GPU performance tools. The flat profile presents statistical information by aggregating the measurements taken from individual executions of a CPU function or GPU kernel. A call graph profile annotates the graph with measurements taken during execution. These basic analysis techniques were used in tools to detect GPU idleness [26, 47, 53, 54, 66], CPU idleness waiting on GPU completion [26, 47, 54, 66], low warp occupancy [26, 54, 66], inefficient GPU cache behavior [54, 66], workload balance between accelerators on the same node [22], and on-device synchronization issues [26, 66].

Blame analysis was created to build on these early GPU analysis techniques by correlating the idleness of a processor with the operations that were responsible for the idleness occuring. Blame analysis groups the computation being delayed with the operations on which the computation is waiting. First introduced in HPCToolkit [16] and later adopted by other tools [80] including NVProf (as *dependency analysis*) [66], blame analysis treats the idleness of a processor as a symptom. The cause is then identified as being the behavior that resulted in that idleness. The fundamental idea is that by exposing the computations that are delaying a processor, a developer may be able to identify ways to improve those computations or eliminate the dependencies between them to reduce idleness. For example, if the CPU is waiting for the GPU to complete processing of a kernel, the blame for the time spent waiting is placed on the GPU kernel that caused the delay. If the kernel could be made more efficient or the

dependence causing the CPU to wait could be removed, idleness could be reduced. If paired with critical path analysis [99], blame analysis can point the developer in a direction that may help them find issues with the potential to improve performance. Blame analysis techniques rely on GPU vendor-supplied measurement data to detect when a processor becomes idle. Note that blame analysis does not give guidance as to how to correct the issues that may be present and does not provide an estimate of the benefit if the issue were fixed.

2.2 Profile-Guided/Feedback-Driven Optimization

Profile-guided optimization (PGO), sometimes referred to as Feedback-driven optimization (FDO), is a compiler code optimization technique that uses profile data to assist the compiler and the linker in generating an optimized binary. PGO approaches compile and run the application using a representative input data set to collect profile data. Profile data is then used by the compiler or linker to apply transformations during code generation to improve the performance of the application. PGO techniques target problems such as reducing cache miss rates [21, 55, 70, 73], loop restructuring [25], and identifying functions where inlining would improve performance [11, 18, 25]. More recently, PGO techniques have expanded their reach beyond performance issues and have been used to identify issues in other areas such as application security [41] and I/O performance [87].

PGOs require a profiling run of the application to collect the data needed by the compiler or linker to apply transformations. The information collected typically takes the form of statistical information such as basic block execution and function call counts as well as the values of hardware counters. While there are two ways PGOs collect profile data,

instrumentation and sampling, the data they collect is often used to solve the same problem. For example, PGOs that reduce instruction cache miss rates have been have created using both instrumentation [17, 25, 55, 73, 79] and sampling [20, 21, 25, 51, 69, 70].

Where instrumentation- and sampling-based approaches differ is in how the profile data is collected. Instrumentation approaches insert instrumentation into the application to collect profile data while sampling approaches probe the application at regular intervals recording hardware counters and other information about the applications state. Sampling has lower overhead than instrumentation for profile data collection but trade this off against lower accuracy. PGOs using only sampling obtained 60% - 78% of the performance benefit seen from instrumentation based counterparts [20, 21]. This gap is caused by the approximate nature of sample based profiles [22]. While the lower benefit is not ideal, the lower overhead of sampling allows profiles to be collected in production environments as shown by AutoFDO [21].

PGOs identify and correct simple problems introduced during code generation by the compiler. These corrections have been limited to problems where the fix is a simple transformation, such as reordering basic blocks. Complex problems introduced by developers that require transformations changing the operations performed by the program, such as those FFM targets, are outside the scope of what a PGO has been able to identify. The single-stage/single-run profile component of PGOs is also limited by the problems described with single-stage instrumentation described in Section 2.1.1.

2.3 Autotuning

Autotuning is the process of identifying efficient input and configuration parameters for an application. Given a programmer defined search

space, the program is automatically run with different combinations of parameters to identify the ones that attempt to optimize a given criteria. Commonly, the end goal for a developer is to find the parameters that result in the largest reduction in execution time. Autotuning techniques have been applied to identify the best choice for parameters such as program options [13, 97, 101], system configurations [13, 59], compiler settings [8, 85], and algorithm configurations [7, 27] to use to increase performance.

Effective use of autotuning techniques require that a developer define the parameters that influence performance. The parameters that influence performance are often unique to the application being tuned, requiring application specific alterations to the autotuning program to allow for searching the parameter space. For large search spaces, a pruning method must also be devised that limits the number of potential parameter combinations to reduce the number of times the application must be run to a feasible amount. This process can miss out on potential optimizations if the developer incorrectly identifies parameters that influence performance, incorrectly prunes the search space, or makes a mistake in the construction of the autotuner. FFM does not need extensive manual setup nor application specific alterations to function.

3 HIDDEN PERFORMANCE OPPORTUNITIES IN GPU APPLICATIONS

Our interest in GPU performance problems stems from the challenges we encountered when creating the GPU-based extreme scale density-based clustering algorithm Mr. Scan [89, 90, 94]. Mr. Scan was the first implementation of the clustering algorithm DBSCAN [29] capable of processing multi-billion point datasets and the first capable of scaling to 8,192 GPU-equipped nodes. All other parallel DBSCAN implementations up until the release of Mr. Scan only demonstrated the ability to cluster up to 100 million points.

One of the most significant challenges we faced in the construction of Mr. Scan was efficient handling of interactions between the GPU and CPU. Months of work were spent improving the efficiency of interactions, such as synchronization operations and memory transfers, to obtain the performance necessary to process billions of points. A frustrating aspect was that tools provided limited assistance in identifying these problems, forcing us to perform significant manual analysis to uncover these hidden problems. The performance issues we identified in Mr. Scan caused us to wonder if what we experienced was unique to our application or if these problems were shared among other GPU applications.

As a result, we performed an in-depth study to identify hidden performance opportunities in highly optimized real-world applications. The hidden performance opportunities we identified take the form of missed many-core parallelization opportunities and inefficiencies in handling interactions with the accelerator, such as duplicate memory copies and unnecessary synchronization. These issues were identified with a combination of source code review and manual instrumentation to gain details about the runtime of functions within the applications and memory structure; manual corrections were inserted when an issue was identified. In

App Name (Version)	Organization	Description	Original Execution Time (Min:Sec)	Reduction in Execution time	Problems Found
Hoomd-Blue [4] (v1.1.1)	Univ of Michigan	MDS	08:36	37%	SYN
Qbox [39] (v1.63.5)	UC Davis	MDS	38:54	85%	DD, SYN
QBall [28] (Apr 24 2017)	LLNL	MDS	67:55	87%	DD, SYN
LAMMPs [75] (Mar 31 2017)	Sandia	MDS	03:34	18%	MP
cuIBM [49] (Sep 21 2016)	Boston Univ	CFD	31:42	27%	SYN, JT

MDS: Molecular Dynamics Simulation, CFD: Computation Fluid Dynamics, MP: Unobvious Missed Parallelization SYN: Synchronization, JT: Just-In-Time GPU Compilation

Table 3.1: Applications improved by adding parallelism and correcting inefficient behavior

our initial experiments with the real-world GPU applications run on Oak Ridge National Laboratory's Cray Titan supercomputer (Table 7.1), we found that the exploitation of these performance opportunities reduces application execution time between 19-87%.

We needed to use a manual process to identify these problems because current performance tools did not provide information that was precise enough to detect the presence of these problems nor accurate enough to explain the cause. Some missing features of existing profilers and tracers included not examining the contents of memory transfers, not associating synchronization operations with the data that they protected, and not examining the actual memory-access pattern of a loop at run time to determine its vectorizability. As a result, a tool such as Nvprof [66] could identify (in some cases) time consuming data transfers, synchronization operations, and loops but did not provide information such as whether these operations were necessary or could be improved.

The goal of our work is to help developers reveal and ultimately correct these inefficiencies in their applications. In this chapter, we (1) characterize missed performance opportunities in many-core applications and why they are difficult to identify, (2) show the performance benefit of correcting these issues, and (3) describe detection methods that can be used by perfor-

mance tools to identify these missed opportunities in other applications. The detection methods we created lay the groundwork for our later work on FFM. To guide our discussion, we group the performance opportunities we have discovered into four categories:

Unobvious missed parallelization opportunities in areas of the application where using the GPU would improve performance: What makes an unobvious region for conversion unobvious is the unknown benefit of converting the region to the GPU. The uncertainty of the conversion is caused by the assumption that the region does not have the necessary characteristics for profitable parallelization on the GPU. The characteristics needed are high parallelism, a flat memory structure (single dimensional arrays), and workload levels high enough to overcome the overheads associated with moving the computation to the GPU. Reducing the uncertainty of converting a region to the GPU is key to discovering unobvious parallelization opportunities. Reducing uncertainty requires that we identify CPU regions contributing significantly to runtime, determine the underlying memory structure of variables accessed within the region, and estimating the overheads of transferring work to/from the GPU.

We describe the issue of missed parallelizations and techniques to address them in more detail in Section 3.1.

Duplicate data transfers causing unnecessary transfers of data already residing in physical memory on the CPU or GPU: The existence of unnecessary transfers is caused by the way GPU accelerated functionality is introduced into applications. The most common method of adding GPU functionality to existing applications is by dropping-in GPU replacements to CPU functionality. GPU replacements often taking the form of a "GPU-ized" library (such as the use of accelerated libraries like cuFFT [64], CUSP [33], and others [23, 62]), a parallel code section inserted by the compiler (such as those generated by OpenACC [96] and OpenMP [14]), or a block of user written code. Duplicate data transfers can occur when

multiple replacements are in use by the application or when CPU-style behavior must be emulated to conform to the existing application structure. When multiple replacement libraries are in use, duplicate transfers to the GPU can occur because the replacements cannot communicate with one another what data they have already moved to the GPU. When CPU behavior must be emulated, such as when the GPU library is in use by an application that does not perform GPU computation, the replacement library cannot assume that CPU data will not change between calls to the library, requiring that all CPU data needed by GPU computation be transferred at every call. The underlying cause of duplication is the lack of reuse of GPU resident memory and the assumption that data has been modified in between calls to dropped-in replacements. A survey of large science applications conducted by Oak Ridge National Laboratory [43] lists the lack of GPU data reuse as one of the key performance issues faced by many high performance accelerated applications.

We describe the issue of duplicate data transfers and techniques to address them in more detail in Section 3.2.

Synchronization between the CPU/GPU that are unnecessary or performed before needed, reducing CPU/GPU computation overlap. Misplaced or unnecessary synchronization occur when a synchronous operation happens before data is actually needed by the CPU or GPU. The existence of synchronization errors is typically due to the drop-in replacement method used by applications, such as by usage of a "GPU-ized" library. Dropped-in replacements are typically required to emulate CPU-style behavior to operate within existing application structures. A requirement of emulating CPU-style behavior is ensuring that the results of a GPU computation are in CPU memory before returning to the application framework, requiring a synchronous memory transfer upon exit of the library. However, applications may not need the GPU data immediately on exit of the library (or even at all) making the synchronous operation

unnecessary.

We describe synchronization issues and techniques to identify misplaced and unnecessary synchronization operations in more detail in Section 3.3.

Unnecessary Just-In-Time (JIT) compilation of GPU code on every execution of the application, increasing the overhead of using a GPU within the application. JIT compilation occurs when the application contains native GPU code that is incompatible with the GPU in use on the system [63]. The incompatibility is the result of specifying the incorrect GPU architecture at compile time or requiring the code to be generated from virtual code by the GPU device driver during execution. When an application is compiled for an incompatible architecture, application performance is affected due to the cost of performing the JIT compilation and by GPU code inefficiencies introduced by selecting the wrong virtual architecture at compile time. The effect in HPC environments can be magnified because the JIT-generated native code is not cached for subsequent executions. In addition, if the default virtual architecture targeted by the compiler is not a good match for the devices actually in use on the system, then the code may not be able to efficiently exploit to the GPU. When these easily correctable inefficiencies exist in the application, *no* notice is given to the user that performance is being negatively affected.

We describe the JIT compilation issue and techniques to address them in more detail in Section 3.4.

3.1 Unobvious Parallelization Opportunities

Unobvious parallelization opportunities exist in applications primarily for two reasons: (1) they have source code structures that do not appear to be favorable to parallelization and (2) they appear to have a minor benefit or even negative performance impact on application performance.

```
for (int i = 0; i < nlocal; i++) {
   if (mask[i] & groupbit) {
     double dtfm;
     dtfm = dtf / mass[type[i]];
     v[i][0] += dtfm * f[i][0];
     v[i][1] += dtfm * f[i][1];
     v[i][2] += dtfm * f[i][2];
}}
```

Figure 3.1: Example of a missed parallelization opportunity from LAMMPs

An example of one of these missed unobvious parallelization opportunities can be seen in the code excerpt taken from the 208K line molecular dynamics application LAMMPs [75] from Sandia National Laboratory (shown in Figure 3.1). This code from LAMMPs shows characteristics that are bad for GPUs: unknown number of loop iterations, multiple multi-level pointer reads and writes, and the presence of a branch condition. When you look at the code, it appears that the CPU-to-GPU memory transfers for v[i] and f[i] need to be done in separate short (three elements at a time for each loop iteration) transfers, resulting in an inherently expensive pattern of transfers. So, the amount of work sent to the GPU may be too small to overcome the overhead of the multiple data transfers. It appears from the description given for existing techniques for detecting parallelization opportunities [6, 15, 44, 61, 68], that these techniques would make the same assumptions that developers would for this code region. Our goal was to test these research tools to verify our assessment of their techniques, however none are publicly available, and have never been tested on real-world code bases of this size.

We were able to obtain a 10% improvement to total application runtime by migrating the code in Figure 3.1 to the GPU. Previous techniques make inaccurate assumptions about variables `v` and `f` and the unknown value of `nlocal`. The assumption that can be drawn from source code is that `v[i]`

and `f[i]` point to completely disjoint memory regions for every value of `i`. That assumption is wrong; `v` and `f` are each allocated in a contiguous manner where all indices point into the same contiguous memory region. Thus the accesses at `v` and `f` can be rewritten as a single dimensional index. The contiguous allocation of `v` and `f` is hidden behind the memory management structure of LAMMPs. It would not be apparent to a developer that these variables are contiguous in memory without in-depth knowledge of the memory management framework in use. The unknown and possibly changing value of `nlocal` adds additional uncertainty since the loop may not operate long enough for any reasonable benefit to be achieved. In our experiments with LAMMPs, we found that the value in `nlocal` was high (over 400,000).

Approaches to detect missed parallelization opportunities need to be able to reveal information about the actual memory access pattern in use and the length of time spent executing within these code segments.

3.1.1 Detecting Unobvious Parallelization Opportunities

The behavior of long running loops with sequential memory access patterns indicates the presence of a loop that is favorable to conversion to the GPU. We view long running loops as GPU favorable because the computation is likely to run long enough to outweigh the overheads associated with GPU computation, such as memory transfer time and the latency of launching the kernel. Loops with only a small amount of execution time on the CPU may have overhead that outweighs any computational benefit, so we consider these to be *unlikely* candidates for conversion. A sequential memory access pattern is often favorable because it allows the GPU to combine memory operations by different threads into a single memory transaction. GPUs are well-suited to codes with high memory bandwidth requirements [30, 60], so identifying codes with this characteristic indicates GPU favorability.

A detection method that could be used by performance tools to identify these behaviors in applications is a dynamic approach combining CPU profiling with memory tracing. The first step uses CPU profiling to obtain information about the execution time of loops within the application. A loop is considered for conversion if it constitutes a large enough fraction of application execution time to be worth the effort of conversion. We can leverage existing performance profilers [1, 47, 72, 83] to accomplish this since they collect the needed CPU profiling information already. The second step uses memory tracing to determine if the loop under consideration has a memory access pattern suitable for parallelization. For each candidate loop, memory tracing is performed on a single representative instance of the candidate loop. Instrumentation inserted into the loop records the addresses used by all load and store operations. A separate memory tracing run of the application would be used to collect traces so profiling results are not perturb. We determine the favorability of the loop to parallelization by analyzing the memory access patterns contained in the trace, looking for contiguous ranges of virtual memory addresses accessed during loop execution. If contiguous virtual memory address ranges can be formed from the individual virtual memory addresses captured, the loop is identified as containing a sequential memory access pattern suitable for the GPU parallelization. Loops identified by both performance profiling and memory tracing as being suitable for the GPU would be marked as a missed unobvious parallelization opportunity.

3.2 Duplicate Data Transfers

Duplicate data transfers are unnecessary transfers of data between CPU and GPU memory. A transfer is unnecessary if the data already exists in the memory space to which it is being written. Unnecessary transfers occur when developers cannot make assumptions about data modifications

between regions of code, such as between functions or libraries, within the application. A region processing data using the GPU may conservatively decide to re-transfer data already resident on the GPU if it could have been modified by another region. Typically, unnecessary transfers occur when libraries are used to add GPU acceleration to applications or when multiple dropped-in GPU replacements to CPU functionality are introduced into an application.

Figure 3.2 shows an example of an unnecessary transfer from the 87K line QBox [39] molecular dynamics application developed at U.C. Davis. The unnecessary transfer is caused by QBox's usage of the discrete Fourier transform library, cuFFT [64]. cuFFT is a library developed by Nvidia as a drop in replacement for the CPU discrete Fourier transform library, FFTW [31]. Maintaining compatibility with FFTW requires that all of the steps needed to setup the transform on the GPU, such as transferring data, must be done within the cuFFT library itself. In the example shown in Figure 3.2, QBox is performing a Fourier transform on data starting at location `data[i]` where `data` is a flat single dimensional array. cuFFT transfers `N` elements starting at position `data[i]` to the GPU. `N` is defined by the application on initialization of the FFT library and is stored in the variable `plan`. The transform is computed on the GPU and the results are transferred back to `data[i]`. Since the values located within `data` are not modified between each subsequent transform, each transfer after the initial iteration contains duplicated data that does not need to be transferred. The duplicate transfers increase application runtime by approximately 40%.

The issue present in Qbox shown in Figure 3.2 extends to QBall [28], an enhanced version of QBox created by Lawrence Livermore National Laboratory. QBall contains experimental features, such as support for f-projectors and the implementation of a highly-scalable algorithm to calculate the time-dependent Density Functional Theory on a many-body

```
...
for(int i = 0; i < n; i++)
   fftw_execute_dft(plan, &data[i], &data[i]);
...
```

A: *Excerpt invoking cuFFT*

```
void fftw_execute_dft(plan, in, out){
   ...
   cuMemcpyHtoD(dev,in);
   [Compute FFT on GPU]
   cuMemcpyDtoH(out,dev);
   ...
}
```

B: *cuFFT library code for function fftw_execute_dft*

Figure 3.2: QBox and Qball's usage of Nvidia's cuFFT library to accelerate discrete fourier transform calculations

system. QBall inherits its application structure, including the structure of the FFT computation, from QBox. By inheriting QBox's FFT structure, QBall also inherited the performance issue seen in QBox when linked with cuFFT. The same performance issue described above for QBox shown in Figure 3.2 appears in QBall.

3.2.1 Detection of Duplicate Data Transfers

We developed a content-based data deduplication approach to identify duplicate transfers. Content based data deduplication approaches compare the hash values of data regions to identify duplicates [78, 86]. Our implementation intercepts calls to cudaMemcpy (and its derivatives such as cudaMemcpyAsync and cuMemcpyHtoD) to obtain the location of data being transferred between the CPU and GPU. If a match to a previous

transfer is detected, a stack trace at the location of the duplicate transfer is recorded. We created a prototype tool of this technique which was incorporated into our later work on FFM.

With a small extension to our detection technique, we would also be able to automatically correct duplicate transfers as they occur. We still must address the fact that the duplicate transfers we identify are not guaranteed to be duplicates on subsequent runs of the application with different inputs. To overcome this limitation, permanent instrumentation will be inserted at data transfers containing duplicate data to always perform a hash check before the transfer. If a transfer that we expect to be a duplicate is not, we perform the transfer and record a stack trace to alert the user. This approach relies on the ability to generate a hash of the data in a transfer request faster than the transfer could take place. The time cost of performing a data transfer can be decomposed into startup costs, the time it takes to move the first byte of data, and the per byte transfer cost after startup. GPU data transfers have high startup costs but low per byte data transfer costs [32] while hash checking has very low startup costs with higher per-byte data costs than a GPU transfer. The fastest CPU hashing algorithm we have tested so far, xxHash [95], adds approximately 3% overhead to application execution time on the applications we have tested so far. Using a hashing approach to identify and eliminate duplicate transfers in QBox [39] we can achieve an estimated 16 - 35% of the benefit we obtained via manual tuning.

3.3 Synchronization

There are two types of GPU synchronization operations that we have identified: implicit and explicit. Implicit synchronization is caused as a side effect of operations such as memory transfers or allocations. Explicit synchronization operations are manually invoked by the application to

synchronize the CPU with the GPU. When a synchronization takes place, the CPU waits for the GPU to complete all existing operations before continuing. First, we want to remove an unnecessary or redundant synchronization operation. Second, we want to delay for as long as possible any operation requiring a synchronization with the GPU to maximize CPU - GPU computational overlap. Ideally, the point where an application performs a synchronization operation is right before the result of the operation is needed by the CPU. We discuss how synchronization errors present themselves in applications and describe an automated method to detect synchronization issues.

3.3.1 Implicit Synchronization Issues

Implicit synchronization operations occur when a library call made by an application synchronizes with the GPU before returning control. The most typical implicit synchronization operations are associated with synchronous data transfers and memory allocation requests. The challenge developers face is determining how to delay or replace operations that implicitly synchronize. The problem of avoiding implicit synchronization is made more challenging when the synchronization is hidden from application code, such as when a library in use by the application is itself making an implicit synchronization call.

The interaction between QBox/QBall with cuFFT, shown in Figure 3.2, is an example of an implicit synchronization. Figure 3.2B shows cuFFT making two calls to cudaMemcpy where each call to cudaMemcpy performs an implicit synchronization. However the result from the second cudaMemcpy operation is not used until after the for-loop in Figure 3.2A. The cumulative effect of removing duplicate data transfers and implicit synchronization operations from both QBox and QBall was a reduction in execution time by 85%.

In cuIBM [49], a 36K line computational fluid dynamics simulator

developed at Boston University, the implicit synchronization operations of `cudaMalloc` and cudaFree delay CPU execution unnecessarily. The `cudaMalloc` and `cudaFree` operations take place on the creation and destruction of temporary GPU vectors. A vector in cuIBM would be created (causing a synchronization), filled with data via an asynchronous memory transfer, used by GPU computation, and then in most cases would be destroyed (causing another synchronization). This pattern of creating and destroying temporary memory spaces for vectors is common throughout the execution of cuIBM. The result is an unnecessary delay of CPU code not dependent on calculations from the GPU. We corrected the problem by allocating vectors that would be reused only once. The result was a reduction in cuIBM's execution time by 8%.

3.3.2 Excessive Explicit Synchronization

Explicit synchronization operations are used to wait for the completion of in-progress asynchronous operations such as data transfers. The challenge that developers face is determining when a explicit synchronization is necessary and where to place it. When a developer does this incorrectly, application performance can be reduced significantly.

In Hoomd [4], a 112K line molecular dynamics simulator, the removal of an explicit synchronization operation reduced execution time by 37%. The explicit synchronization, shown in Figure 3.3, is used to wait for the GPU to update the shared variable `sharedStatus`. `sharedStatus` indicates whether the GPU computation failed because not enough GPU memory was allocated for the operation. The value of `sharedStatus` is true (successful GPU completion) for every iteration of the for-loop except the first iteration when GPU memory is initially allocated by the CPU. Even though the value of `sharedStatus` is true for iterations `2 to N` of the for-loop, the application still synchronizes with the GPU on every iteration causing the reduction in performance by delaying the unrelated CPU computation.

(a) Original version of Hoomd's main compuational loop

```
// Status variables shared between CPU and GPU
bool sharedStatus; int * GPUData;
// Size of GPUData
int size = 0;
cudaMalloc(&GPUData, size);
// Main computational loop of hoomd
for(step = 0; step < nsteps; step++) {
    ...
    do {
        GPUComputation<<< >>>(sharedStatus,
                              GPUData,
                              size,
                              ...);
        // Synchronize to get GPU updates
        // to sharedStatus
        cudaDeviceSynchronize();
        // If sharedStatus is false...
        // allocate more GPU memory and retry
        if(sharedStatus == false) {
            size = len(GPUData) + ...;
            cudaMalloc(&GPUData, size);
        }
    } while(sharedStatus == false);
    // CPU work not dependant on GPU data
    for(i = 0; i < count; i++)
        ...
    ...
    // Existing Implicit Synchronization
    cudaMemcpy(...)
    ...
}
```

(b) Manually improved version with reduced CPU delay

```
// Status variables shared between CPU and GPU
bool sharedStatus; int * GPUData;
// Size of GPUData
int size = MAX_DATA_SIZE;
cudaMalloc(&GPUData, size);
// Main computational loop of hoomd
for(step = 0; step < nsteps; step++) {
    ...

        GPUComputation<<< >>>(sharedStatus,
                              GPUData,
                              size,
                              ...);









    // CPU work not dependant on GPU data
    for(i = 0; i < count; i++)
        ...
    ...
    // Existing Implicit Synchronization
    cudaMemcpy(...);
    ...
}
```

Red statements depend on results from GPU

Blue statements have no GPU dependencies

Figure 3.3: A flat representation of an explicit synchronization error in the main computational loop in Hoomd.

3.3.3 Dectecting Implicit and Explicit Synchronization Opportunities

A synchronization opportunity is present when a synchronization operation causes unnecessary delay. We view delay as unnecessary when CPU computation blocks for the GPU but does not access data shared with the GPU. The result of CPU computation being delayed unnecessarily is a reduction in CPU - GPU overlap. We have identified three types of unnecessary delay: (1) when the CPU does not access shared data (data shared with the GPU) after the synchronization, (2) when the placement of the synchronization is far from the first access of shared data by the CPU, and (3) when CPU computation not dependent on GPU data is delayed by

(a) Original version with synchronizing too early

```
cudaMemcpyAsync(dest, src, len,
                ...);
...
cudaDeviceSynchronization();
v = 1;
if (cond1) {
  ...  // Long CPU Exec Time
} else if (cond2) {

  v = dest[0];
  ...
} else {
  ...  // Long CPU Exec Time

  v = dest[1];
}
result = v;
```

(b) Improved version with reduced CPU delay

```
cudaMemcpyAsync(dest, src, len,
                ...);
...

v = 1;
if (cond1) {
  ...  // Long CPU Exec Time
} else if (cond2) {
  cudaDeviceSynchronization();
  v = dest[0];
  ...
} else {
  ...  // Long CPU Exec Time
  cudaDeviceSynchronization();
  v = dest[1];
}
result = v;
```

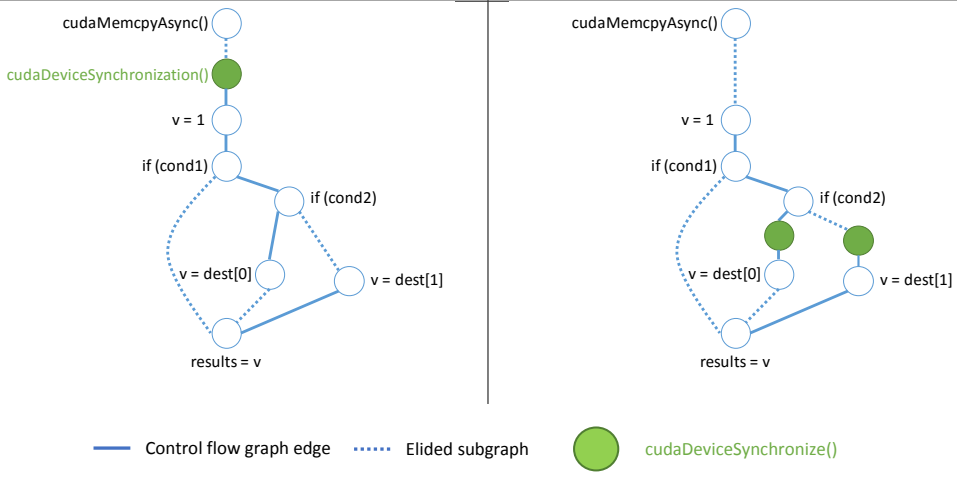

Figure 3.4: Illustrative example of an early synchronization causing unnecessary delay

a synchronization. Unnecessary delay can be reduced by removing the synchronization, moving the synchronization closer to shared data access, or by reordering CPU computation to make the synchronization obsolete.

Figure 3.4 shows an example of how delaying (or removing) a synchronization can reduce the amount of time the CPU is delayed. This is a general example representing the behavior seen in QBox, cuIBM, and Hoomd. In the code section shown in the left of Figure 3.4, a synchronization operation occurs after a memory transfer even though the shared data (stored in `dest`) may not be accessed. If `cond1` is true, the synchronization is unnecessary, blocking the CPU for no reason. In the other cases, there may be enough CPU work performed before the access to `dest` that a delay could be avoided by moving the synchronization closer to the shared data access. The impact of the unnecessary delay can be magnified if this code section is called multiple times, such as within a loop.

Our technique to identify unnecessary synchronization operations uses a dynamic approach combining the techniques of profiling, memory tracing, and program slicing [45, 88] to identify where a synchronization opportunity is located and how to correct its behavior. Our approach can be broken down into five steps: (1) identify the synchronization operations that cause long delays on the CPU, (2) determine what data is shared between the CPU and GPU, (3) identify the CPU instructions that access shared data, (4) determine how far the instruction performing the synchronization is from the first CPU instruction that accesses data shared, and (5) determine if CPU computation exists that does not depend on shared data. Using this information, we will generate the corrective measure that should be taken and an estimate of the amount of time that could be saved if the measure was taken. We explain how to gather this information below.

We target synchronization operations with long CPU delays because a change in their synchronization behavior can result in significant im-

provements in CPU - GPU overlap. Due to the synchronization reporting issues described in Chapter 2.1.2, existing tools cannot be used to accurately identify and time synchronizations. We needed to create a method for detecting and timing synchronization behavior. Our approach is to identify the function(s) that perform synchronization operations between the CPU and GPU, inserting timers to identify synchronizations with long delays.

At each synchronization, we must identify what data is shared between the CPU and GPU. Data can be shared between the CPU and GPU using one of two methods: a memory transfer or through the mapping of CPU memory pages to the GPU. Both methods are initiated through requests made through a standardized API. We identify data being shared between the CPU and GPU by intercepting these requests and recording the memory location (and size) of the data being shared.

We identify where shared data is accessed by CPU computation using memory tracing. Memory tracing is used to determine the instructions that access a memory location containing shared data at runtime. The ordered set of instructions accessing shared data allows us to identify two types of unnecessary delay: no shared data access by the CPU and synchronization far from the use of shared data. If the set of instructions accessing shared data is empty, no access to shared data occurs and the synchronization can be removed. If the total number of instructions between the end of the synchronization and the first access to shared data is large, we know that CPU delay could be reduced by moving the synchronization closer to this access. The corrective measure is to move the synchronization to the location of the access. The synchronization opportunities in QBox [39], QBall [28], and cuIBM [49] fall under these types and are identified here.

The third type of unnecessary delay is when CPU computation not dependent on GPU data is being delayed by a synchronization. A CPU computation is not dependent on GPU data if the values of variables used

in the computation are not affected by changes to data shared with the GPU. The delay is unnecessary because it can be reduced by performing the CPU computation before the synchronization, increasing CPU - GPU overlap. Identifying this case of unnecessary delay requires that we locate instructions that do not depend on GPU data. We use program slicing [45, 88] to identify instructions that do not depend on GPU data. An instruction is dependent on GPU data if the values used by the instruction are affected by data shared with the GPU. A forward slice is created starting at the synchronization operation with the locations of shared data being used as the criterion for the instructions to be included in the slice. The result is a slice containing the instructions that *may depend* on data shared with the GPU. We are interested in the set of instructions that are *not* in the slice since they do not depend on data shared with the GPU. If the number of instructions not in the slice is large (say, greater than a few hundred instructions), then moving these instructions before the synchronization operation could have a noticeable benefit. The synchronization opportunity in Hoomd [4] falls under this type of unnecessary delay and would be detected here.

3.4 JIT Compilation

GPU native code may need to be generated, from a GPU virtual architecture at runtime because there is no native GPU code present in the executable file, or the code that is present is for the wrong model GPU. The lack of native GPU support is a product of the misconfiguration of the applications at compile time. Common reasons for misconfiguration are a developer not knowing the correct native architecture of the GPU, build systems such as CMake incorrectly identifying the native architecture, and use of compiler defaults that produce incompatible binaries for most GPUs. When a miss-configuration of the architecture occurs, application

users are not notified that their application is misconfigured, either at compile or execution time.

cuIBM [49] is an example of the impact a misconfiguration can have on performance. 18% of cuIBM's execution time is spent performing JIT compilation because the wrong architecture is selected by the build system. cuIBM defaults to compiling to the virtual architecture "compute_20", while the GPUs in the system actually support "compute_35". Since we ran cuIBM in an HPC environment (the Cray Titan supercomputer at Oak Ridge), the JIT compilation is not cached and must be performed at every execution.

Application incompatability with its GPU codes seems to be quite simple and is surprising (but widely present). We can detect GPU code incompatibility at application startup and provide explicit instructions to the user as how to produce a more efficient executable.

4 THE FEED-FORWARD MEASUREMENT MODEL

Delivering actionable feedback that provides an estimate of expected benefit requires refining the techniques introduced in Chapter 3 to address three challenges:

1. Identify and trace hidden synchronization operations.
2. Predict what the application performance would be if a problematic operation were corrected.
3. Combine these techniques with the detection of problematic operations into a one that could be implemented by a performance tool.

We focus our efforts on problematic synchronization and memory transfer operations. These problems were the most common and most performance detrimental in the applications we studied. In this work, we have not addressed missed parallelization opportunities and JIT compilation problems. For missed parallelization opportunities, generating an estimate of expected benefit would require accurately predicting performance of a CPU loop on the GPU. Predicting the performance of CPU codes on the GPU is an active area of research [9, 12] but we feel is not mature enough where we would be comfortable relying on these techniques. As these techniques mature, the problem of identifying missed parallelization opportunities would be worthy of revisiting. Due in part to the publication of our original work where we describe the JIT problem [91], the vendor has taken steps to mitigate this issue by reporting when JIT operations take place.

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 [57]. 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 its 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 the CPU code to collect performance data on synchronization and memory transfer operations, capturing synchronization operations 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

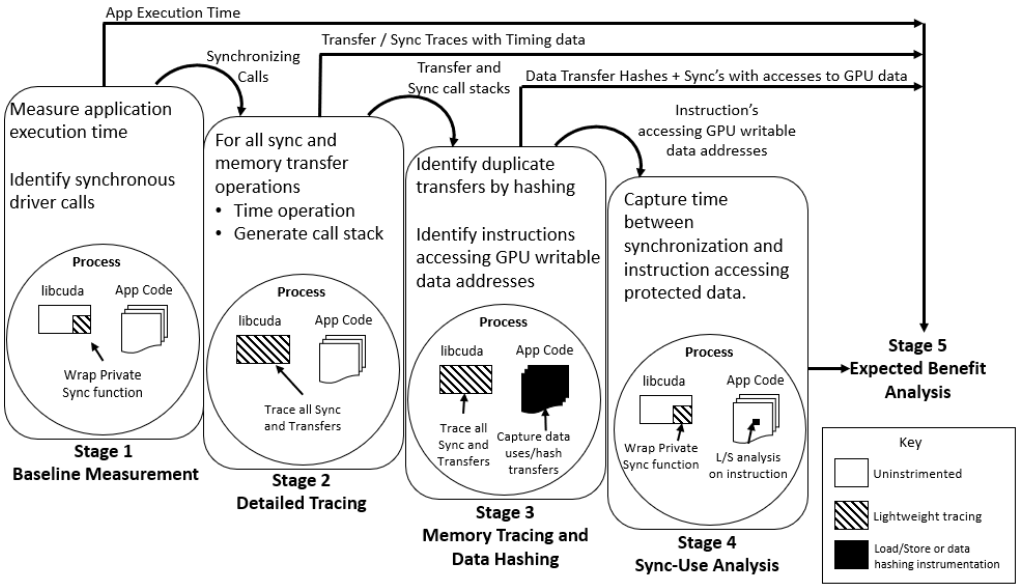

Figure 4.1: Overview of the stages of the FFM model

of parallelization frameworks such as CUDA [63], OpenACC [96], and OpenMP [14]. Figure 4.1 shows an overview of the stages of FFM, the data each stages collects, and how the stages interact. The five stages of the FFM model are:

Stage 1 - *Baseline Measurement*: Collect the list of application functions that performed a GPU synchronization operation 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 synchronization. This list dictates where more detailed information will be collected in stages 2 and 3.

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 stack trace. The traced functions are the ones identified in stage 1 as performing a

synchronization and a predefined set of user-space GPU driver function calls known to perform memory transfers.

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 stack trace of the synchronization operation, the location of the instruction that first accessed a memory location containing data that could be modified by the GPU, and a stack trace of the instruction location that performed the access. For memory transfers, we collect hashes of the data being transferred to and from the GPU.

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 instructions that access GPU-computed data are identified from the Memory Traces collected in stage 3.

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 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.

In this chapter, we describe each of these stages and the information they collect.

4.1 Baseline Measurement Stage

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 that waits for completion of a sequence of operations on the GPU (see Figure 4.2). 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 created a method that can automatically identify the internal synchronization function of the user space driver. The technique starts by creating call graphs for functions known to perform synchronization operations. We intersect the call graphs of these functions to generate a list of common functions that are called by the known synchronous functions. To identify the function in which we are interested, we run a small program that live-locks on a synchronization with the GPU. We instrument the list of common functions and record which functions never return. This generates a small stack, typically 1 or 2 functions, that are do not return when a synchronization is performed. We select the deepest function on the stack as the synchronization function.

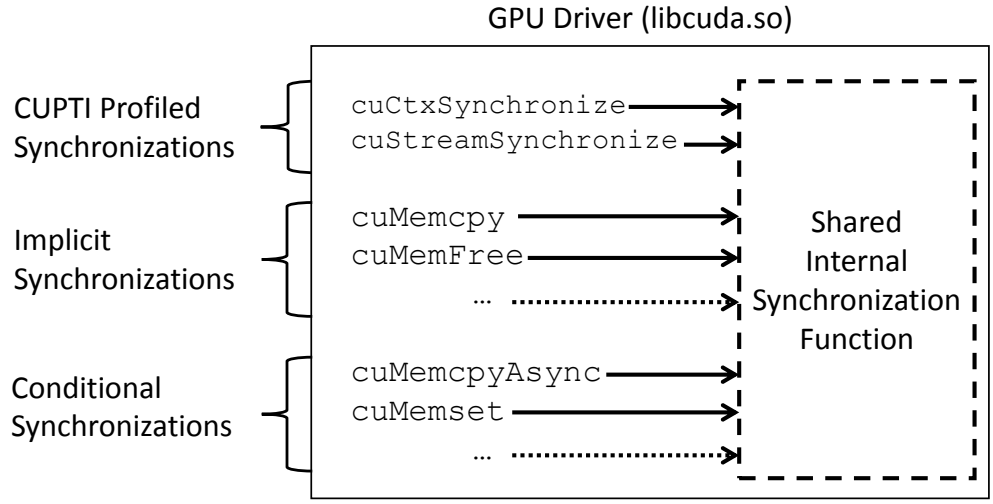

Figure 4.2: The internal synchronization function instrumented by FFM

4.2 Detailed Tracing

The detailed tracing stage traces all synchronization and memory transfer operations performed by the application. For each operation, we collect a stack trace 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: synchronizing functions identified in the Baseline Measurement stage, functions described by the GPU driver API as performing memory transfers (such as `cuMemcpy`), and the internal synchronization function.

4.3 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

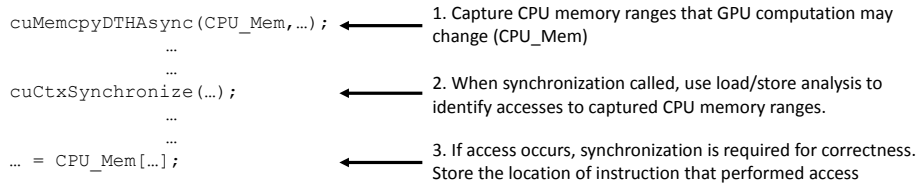

Figure 4.3: An illustrative example of the steps the FFM model takes to identify problematic synchronization operations

correctness. Problematic synchronization operations are ones that are not required to maintain correctness and ones that are required for correctness but unnecessarily reduce CPU/GPU overlap. We target the two types of problematic synchronization operations described in Chapter 3.3.3: when the CPU does not access shared data (data shared with the GPU) after the synchronization and when the placement of the synchronization is far from the first access of shared data by the CPU. 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. For synchronization operations, we use memory tracing. For memory transfer operations, we use a content-based data deduplication strategy.

4.3.1 Identifying Problematic Synchronization Operations

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 4.3 shows an example of how FFM identifies problematic syn-

chronization operations. 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.

4.3.2 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 buffer of 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 stack trace of the duplicate transfer, the location of the first transfer of the duplicated data, and the hash of the data that was transferred.

4.4 Sync-Use Analysis Stage

The Sync-Use Analysis stage collects timing information to determine if a synchronization is misplaced. For synchronization operations 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.

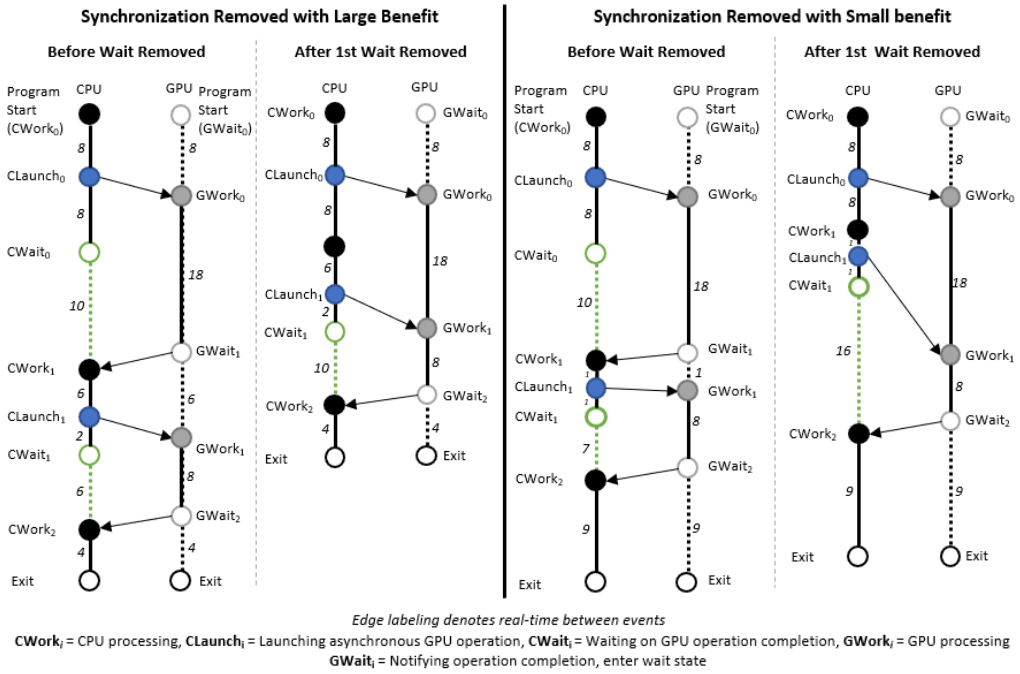

Figure 4.4: Example of the different outcomes from removing a problematic synchronization

4.5 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 [99], changes in the behavior of remaining operations can eliminate any benefit from fixing problematic operations. Two examples can be seen in Figure 4.4. 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 model the effect of fixing the problem has on application execution time. The model generates an estimate of the time saved if the problematic operation were (re)moved. The estimate takes into account the effect the changing the problematic operation will have on other unchanged operations in the program.

5 THE PERFORMANCE MODEL OF FFM

The time that a developer can spend correcting performance problems often is limited. Given a list of performance problems in their program, they may need to chose on which problems they will focus. A developer starts by choosing the problems that, if corrected, will have the greatest impact on performance. With current tools, the developer makes this choice based how much time a given operation consumed. However, the amount of time consumed by an operation is not always a good predictor of the benefit that could be obtained from correcting a problem. Selecting and correcting the problems that have consumed the most time in the program can result in skipping problems that have substantial performance benefit. Giving the developer insight on the potential benefit of correcting a problem is key to unlocking this hidden performance potential.

The FFM performance model generates an estimate of expected benefit if a problem were corrected. After an estimate is generated, problems are grouped together if they have the same underlying cause, allowing easier identification of where a single corrective measure may fix multiple problems. Giving developers an estimate of expected benefit and grouping problems together by their cause allows them to focus their time on problems that are the most impactful.

FFM models application execution as a graph $G = (N, V)$, where N is the set of events on each processor and V is the set of edges. $N = \{C, G\}$ 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), or a CPU event that requests that the GPU perform work (CLaunch).

An edge describes the Lamport happens-before [48] ordering between events. Edges have a label Duration that denotes the real-time duration of the event. On the same processor, an outbound edge from a node n_x to a node n_y denotes the operation performed by n_x completes processing before n_y starts executing. An edge between processors denotes a dependency where a node n_x must wait for an operation on the other processor to complete before beginning execution. 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 synchronization operations, 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) $_{\text{Duration}} = 0$). For misplaced synchronization operations, 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)$_{\text{Duration}} -$ 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)$_{\text{Duration}} = 0$). 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.

5.1 Expected Benefit Algorithm

Figure 5.1 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 -* $\mathrm{OutCPUEdge(Node)_{Duration}}$*)*. 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)* $_{\mathrm{Duration}}$ can be as large as $\mathrm{OutCPUEdge(Node)_{Duration}}$, negating any benefit. $\mathrm{OutCPUEdge(Next\ Sync)_{Duration}}$ 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 $\mathrm{OutCPUEdge(Node)_{Duration}}$. The total GPU idle time between Node and NextSync can contract by as much $\mathrm{OutCPUEdge(Node)_{Duration}}$.

```
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 5.1: The expected benefit algorithm

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 $\mathrm{OutCPUEdge}(\mathrm{Node})_{\mathrm{Duration}}$ 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 $\mathrm{NextSync}$. 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 ($\mathrm{OutCPUEdge}(\mathrm{Node})_{\mathrm{Duration}}$) and the maximum period of GPU idle time. On line 19 we calculate the new duration of $\mathrm{NextSync}$ by adding $(\mathrm{OutCPUEdge}(\mathrm{Node})_{\mathrm{Duration}} - \mathrm{EstBenefit})$ to the current duration of $\mathrm{NextSync}$.

$\mathrm{EstBenefit}$ on line 22 is the estimated time that could be saved if *only* the synchronization operation were removed. If the operation also performs a duplicate memory transfer, the estimate on line 22 is combined with the estimated benefit produced by `RemoveMemoryTransfer` to form the final expected benefit. If removing an unnecessary synchronization requires removing the operation performing the synchronization, the estimate on line 22 can be combined with the execution time of the operation performing the synchronization to form the final expected benefit. We perform this combination on `cudaFree` operations since there is no asynchronous version. Correcting a `cudaFree` synchronization problem would require its removal.

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 $\mathrm{OutCPUEdge}(\mathrm{Node})_{\mathrm{Duration}}$ would change if $\mathrm{Node}_{\mathrm{STime}}$ was increased. $\mathrm{Node}_{\mathrm{STime}}$ increases by the time to first use (*FirstUseTime*), 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).

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 5.1 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 a problem in the source code for the template would affect all instances. The `ExpectedBenefit` function in Figure 5.1 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 synchronization operation needs 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.

5.3 Diogenes: A Performance Tool Implementing FFM

We have implemented the FFM model in a tool we call Diogenes. Diogenes identifies problematic synchronization and memory transfer operations, 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 [66] and CUPTI [65]) 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.

5.4 Experiments with Automatic Problem Identification

We tested the effectiveness of FFM's model to identify problematic operations and predict benefit by applying Diogenes to four real world applications: cumf_als [84] an alternating least square matrix factorization library developed at IBM and University of Illinois Urbana-Champaign, cuIBM [49] the computational fluid dynamics simulator developed at

Boston University, AMG [100] an MPI based parallel algebraic multigrid solver developed at LLNL, and the Gaussian GPU benchmark from Rodinia [19] developed at the University of Virginia. The applications we tested, outside of cuIBM [49], differ from those in Chapter 3 to show that the problems we target appear in a wider variety of application types than just computational fluid dynamics and molecular dynamics simulations. However, we revisit Qbox [39] in our experiments in Chapters 6 and 7. 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 5.4.1 we detail the problems detected by Diogenes and the fixes applied in each application. Section 5.4.2 compares the output of Diogenes to other performance tools, and Section 5.4.3 discusses the limitations of Diogenes and the FFM model.

5.4.1 Application Problems

Table 5.1 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 synchronization operations 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

App Name (Version)	App Size (Lines of Code)	Organization	Application Description	Diogenes Discovered Issues	Diogenes Estimated Benefit (% of exec)	Actual Runtime Reduction (% of exec)
cumf_als *(git rev: a5d918a)*	5 K	IBM/UIUC	Matrix Factorization	Sync and Mem Trans	137 s (10.0%)	106 s (08.3%)
cuIBM *(git rev: 0b63f86)*	36 K	Boston University	Immersed Boundary Method	Sync	202 s (10.8%)	330 s (17.6%)
AMG (v2.14)	65 K	LLNL	Algebraic Multigrid Solver	Sync	0.34s (06.8%)	0.29s (05.8%)
Rodinia (v3.1)	<1 K	UVA	Gaussian (CUDA)	Sync	0.13s (02.2%)	0.12s (02.1%)

Table 5.1: Applications improved by correcting a subset of Diogenes discovered issues

Diogenes, resulting in a much larger benefit.

cumf_als [84] is a GPU-based large matrix factorization library that uses the alternating least square (ALS) method. We ran our experiments using the GroupLens MovieLens [40] 10M input data set, a dataset containing 10 million user ratings for movies created by the University of Minnesota. The MovieLens data set was run with cumf_als iteration count set to 5000. Diogenes estimated that correcting a sequence containing problematic synchronization and memory transfer operations in cumf_als would result in a reduction in execution time by 11% (see Figure 5.2). This sequence contained 23 problematic operations spread across two functions in two different source files. The sequence contained 18 problematic synchronization and 5 problematic synchronous memory transfer (with both an unnecessary transfer and synchronization) operations.

To remove the problematic synchronization operations at the `cudaFree` operations in the beginning of the sequence in Figure 5.2, 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 5.2, looking for the problems that we could fix easily. The operation at entry 10 of Figure 5.2 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 5.4 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 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 an unnecessary synchronization and removed memory transfer operations that would repeatedly retransfer the same data to the same destination. For problematic synchronization operations at `cudaFree`, we could not simply remove the `cudaFree` operation as it would lead to a memory leak. We also could not swap `cudaFree` for an asynchronous operation as there is no asynchronous version of `cudaFree` in the CUDA API. 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

```
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 5.2: A sequence of unnecessary operations identified by Diogenes in cumf_als

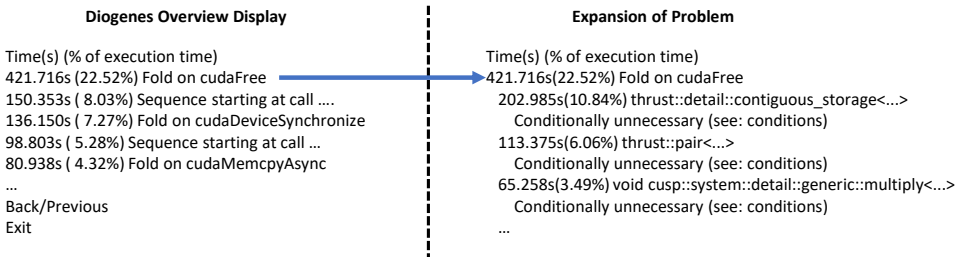

Figure 5.3: Diogenes overview of problematic operations (left) and the expansion of problems at `cudaFree` (right) for cuIBM

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 report an 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 [49] is a computational fluid dynamics (CFD) application that uses the immersed boundary method (IBM) to calculate fluid flows on a cartesian grid. We ran our experiments using the lid-driven cavity with Reynolds number 5000 dataset supplied (*lidDrivenCavityRe5000*) in the code repository for cuIBM [50]. For cuIBM, Diogenes reported that the 22% of execution time could be saved by removing problematic `cudaFree` operations (left side of Figure 5.3). Asking Diogenes for more details on the `cudaFree` revealed that a single template function accounted for 10.8%

of application execution time (right side of Figure 5.3). 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 [67] 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 [100] 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 provided with AMG. For AMG, Diogenes estimated that 6.8% of execution time could be saved by fixing a problematic 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. The result was a 5.8% improvement, close to the predicted value.

```
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 5.4: The estimate of benefit reported by Diogenes for fixing a subsequence of the operations in Figure 5.2.

App 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	745 s (52.0%, 1)	628 s (24.5%, 1)	1 s (<0.1%, 3)
	cudaFree	275 s (18.7%, 2)	258 s (10.1%, 2)	214 s (15.7%, 1)
	cudaMalloc	218 s (17.3%, 3)	230 s (09.1%, 3)	-
	cudaMemcpy	158 s (11.8%, 4)	119 s (04.7%, 4)	30 s (02.2%, 2)
cuIBM	cudaFree	Profiler Crashed	447 s (12.3%, 1)	421 s (22.0%, 1)
	cudaLaunchKernel		395 s (12.1%, 2)	-
	cudaMalloc		382 s (10.8%, 3)	-
	cudaDeviceSynchronize		170 s (04.8%, 4)	136 s (07.2%, 2)
	cudaMemcpyAsync		163 s (04.4%, 5)	80 s (04.3%, 3)
	cudaFuncGetAttributes		154 s (04.2%, 6)	-
	cudaStreamSynchronize		52 s (01.4%, 7)	4 s (00.2%, 4)
AMG	cudaFree	0.93s (18.7%, 1)	0.39s (03.3%, 2)	0.31s (06.3%, 2)
	cudaMemset	0.81s (16.3%, 2)	0.57s (06.0%, 1)	0.34s (06.8%, 1)
	cudaMallocManaged	0.15s (03.1%, 3)	0.06s (00.7%, 4)	-
	cudaStreamSynchronize	0.13s (02.6%, 4)	0.12s (01.3%, 3)	0.07s (01.4%, 3)
Rodinia	cudaThreadSynchronize	6.05s (94.9%, 1)	5.01s (75.7%, 1)	0.13s (02.2%, 1)
	cudaMemcpy	<0.01s (00.9%, 2)	0.07s (01.2%, 2)	0.06s (00.9%. 2)
	cudaFree	<0.01s (<0.1%, 3)	<0.01s (00.2%, 3)	<0.01s (<0.1%, 3)

Table 5.2: Comparison of cuda function call profiling results between Diogenes, HPCToolkit, and NVProf

Rodinia [19] 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, obtaining a benefit close to the predicted value.

5.4.2 Comparison to NVProf and HPCToolkit

Table 5.2 shows a comparison of the expected-benefit provided by Diogenes against the results produced by NVProf [66] and HPCToolkit [56]. NVProf is Nvidia's profiling tool supplied as part of the CUDA [63] software distribution and HPCToolkit is a sample-based profiling tool created

at Rice University. NVProf and HPCToolkit are two of the most widely used profilers for GPU applications, especially on high performance computing platforms. 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 synchronization and memory transfer operations. An example can be seen in the profiling results for cumf_als. NVProf and HPCToolkit reported that the function `cudaDeviceSynchronize` 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.

There were six other differences in results between Diogenes and the other tools for the four programs tested. The gap between Diogenes and other tools was caused by two differences in the way results are generated: *Diogenes expected benefit analysis provides an estimate of the potential reduction in synchronization delay for an operation while other tools only provide the time consumed by an operation*. As we described in the section 5.1, the removal of a synchronization operation can increase the delay of the next synchronization. The increase in delay of the next synchronization operation reduces the benefit that can be obtained. The removal of the synchronization operations at `cudaFree` in cumf_als, `cudaMemcpy` in cumf_als, `cudaStreamSynchronize` in cuIBM, `cudaMemcpyAsync` in cuIBM, `cudaMemset` in AMG, and `cudaThreadSynchronize` in Rodinia substantially increase the delay of the next synchronization operation. Diogenes accounts for this increased delay by reducing the expected benefit of removing the synchronization operation while other tools do not.

Diogenes expected benefit analysis excludes required operations while other tools do not. Diogenes determines whether a memory transfer or synchronization is required, while other tools simply report how much time was consumed by the operation. If Diogenes detects that an operation is required, the expected benefit of the operation is set to zero since it cannot be (removed. `cudaMemcpy` (in cumf_als), `cudaStreamSynchronize` (in cuIBM), and `cudaMemcpyAsync` (in cuIBM) have occurrences where the operation being performed is required and thus those occurrences are excluded from Diogenes results but not from the results of other tools.

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 5.2 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.

5.4.3 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.

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. When testing with other applications not included in this study, we saw performance penalties as high as 45x. While the cost is high for 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, so the data could be modified before a hash could be calculated, hiding the presence of a problematic transfer.

6 AUTOMATIC REMEDY IDENTIFICATION

While FFM was able to identify problems missed by other approaches, there were still gaps in its analysis that resulted in unexposed performance opportunities. FFM could label a synchronization as problematic, but it could not identify if the synchronization is a component of a larger construct that exhibit a problem that spanned many operations. For example, a frequently occurring unnecessary synchronization caused by a memory free operation (such as `cudaFree`) could indicate that a larger memory management problem is present. If larger constructs exhibiting this problem could be identified, it would result in the elimination of memory allocation and free operations, significantly increasing the potential performance benefit.

When a problem is discovered, identifying the correct remedy to employ requires an understanding of the cause of the problem. We focus on identifying the cause of four of the most common types of synchronization problems we have seen in the real-world applications: 1) memory transfer issues, 2) memory management issues, 3) unnecessary operations, and 4) misplaced operations. Fixing the cause of these problems can result in a reduction in execution time by up to 43%. In this chapter, we describe the challenges of automating the identification of the cause of the problem, remedies for each problem type, and the extensions made to FFM to support these capabilities.

6.1 Memory Transfer Issues

The unnecessary use of synchronous memory transfer operations is a problem that found to be common in large applications, such as within cuIBM, cumf_als, and cuFFT when used with Qbox. One such instance can be seen in Figure 3.2 with the synchronization operations performed

in Nvidia's cuFFT library when used as a drop-in replacement for FFTW. In this example, `cuMemcpyHtoD` and `cuMemcpyDtoH` both perform an implicit synchronization during the process of performing a transfer. The implicit synchronization operations can be unnecessary depending on how the application uses `fftw_execute_dft`. The molecular dynamics application Qbox, when linked against cuFFT in compatibility mode, has instances where the implicit synchronization operations performed by `cuMemcpyHtoD` and `cuMemcpyDtoH` are unnecessary.

The obvious remedy for an unnecessary synchronization performed at a transfer is to convert these call to their asynchronous form, such as converting `cuMemcpyDtoH` to `cuMemcpyDtoHAsync`. However, performing only this conversion may result in the synchronous behavior remaining. In cases where the transfer being performed is from the GPU to the CPU, such as with `cuMemcpyDtoH`, converting only the call site to its asynchronous form will not eliminate the synchronous behavior unless the CPU memory is also pinned. A proper remedy for this issue requires identifying and converting all CPU memory used in a transfer to use pinned pages, such as by allocating the CPU memory with `cuMemAllocHost`. Using pinned pages gives the added benefit of increasing the transfer speed itself, reducing transfer time by as much as 50%. In large programs, containing 100K+ lines of code with multiple levels of indirection between the allocation of an address and its use in a transfer, identifying the CPU memory allocations that would need to be changed can be difficult without assistance.

Providing actionable feedback requires not only identifying the problem but also describing what needs to change to remedy the problem. The requirements to create a remedy that is actionable by the developer are shown in Figure 6.1. We must identify 1) the locations in the program where CPU memory is allocated, 2) the problematic transfers and the CPU memory used in the transfer request, and 3) the locations where the CPU memory is freed. For the locations of memory allocation and free opera-

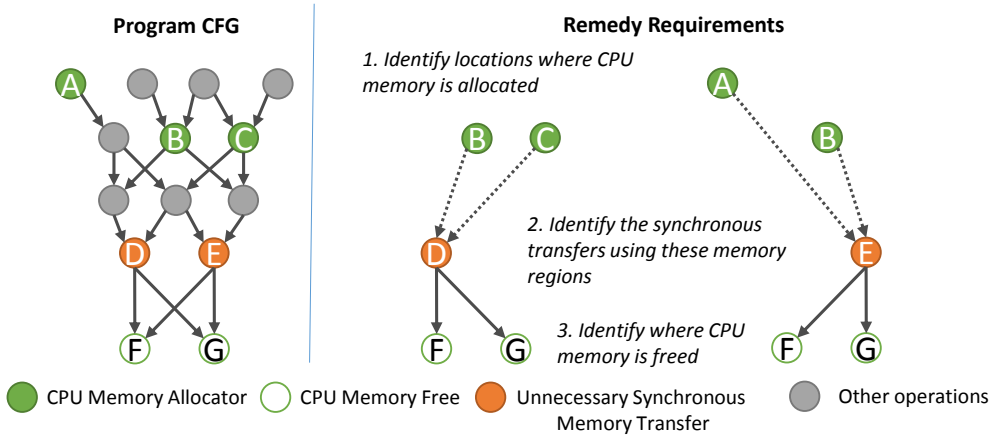

Figure 6.1: Information required to remedy a problematic synchronization caused by a memory transfer

tions to be useful, they should relate to a line (or a collection of lines) in application source code. Relating back to the source code gives context to allocation and free operations when they performed by an external library, such as an allocator in the C++ standard library, allowing the developer to place blame for the improper transfer behavior on a specific library, class, and function.

6.1.1 Extensions to FFM for Memory Transfer Issues

We extend stages 2 and 5 of FFM to support the identification and automatic remedy generation of memory transfer problems (see Figure 6.2). We extend stage 2 to construct a simple dynamic data flow graph to track the location where memory addresses are allocated, what memory addresses are used by transfers, and the location that allocated memory was freed. We use a binary interposition approach that leverages the tool GOTCHA [76] developed by Lawrence Livermore National Laboratory to track these operations. GOTCHA is a tool that allows a user to programatically define and insert function wrappers into an application.

We interpose on `malloc`, `free`, and memory transfer operations to

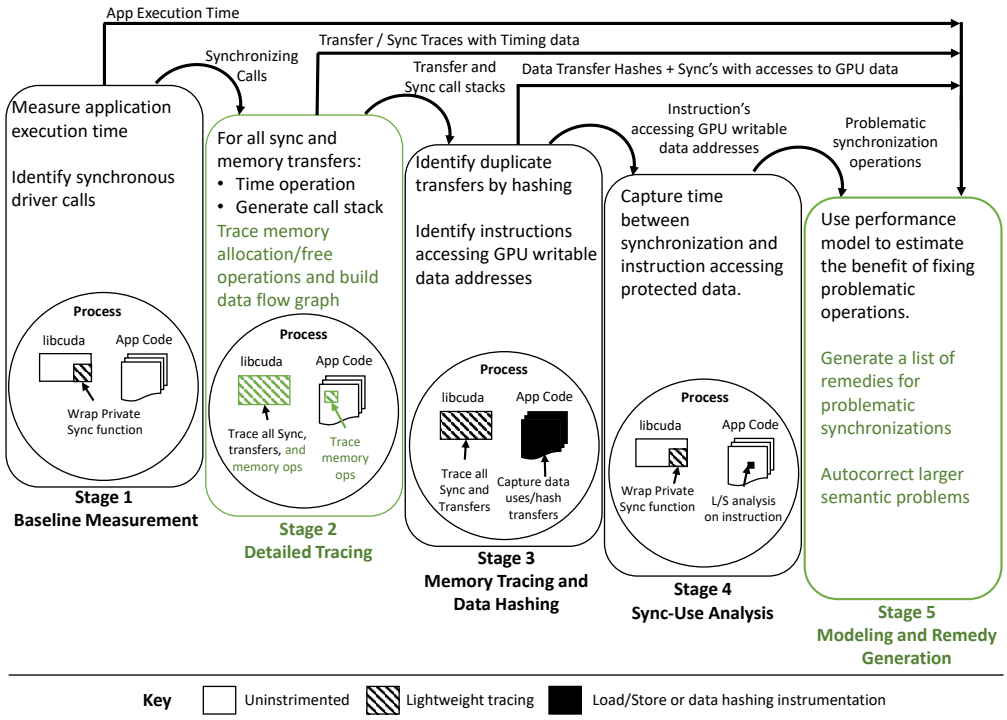

Figure 6.2: Overview of the stages of the FFM model with extended FFM components listed in green.

capture the locations where CPU memory addresses are used or created. Memory transfer operations, such as `cuMemcpyHtoD` and `cuMemcpyDtoH`, are interposed to identify the CPU addresses used as parameters to the call. When a memory transfer operation is requested by the application, we check the CPU address that is used in the call to determine if it was allocate by `malloc`. If the address was allocated via `malloc`, we record the location of the the transfer and the location of the allocation. When the recorded address is freed by `free`, we record the location it was freed. We also interpose on the pinning functions of CUDA, such as `cuMemAllocHost`, to identify memory that was already pinned by the application.

Stage 5 was modified to generate a remedy based on how the memory was allocated for these transfers. We use the existing data that Diogenes

```
cuMemcpyHtoD_v2 called at...
  Offset 131568 in libcufftw.so.8.0
  ...
  cosft1(int, double*) at line 146 in sinft.C
  Species::initialize_ncpp() at line 411 in Species.C
  ...
Problem:  Unnecessary synchronous transfer, replace with cudaMemcpyAsync and
          pin CPU memory address
Pin non-pinned CPU memory allocated at:
  operator new(unsigned long) at line 56 in new_op.cc
  ...
    CPU memory freed at:
      operator delete(void*) at line 46 in del_op.cc
      ...
```

Figure 6.3: Example Diogenes remedy output for unnecessary synchronization operations caused by memory transfers in Qbox

collects to determine if the synchronization performed by the transfer is unnecessary. If the transfer is unnecessary, we generate a remedy listing the transfer operation with the unnecessary synchronization along with the memory allocation and free operations that would need to be modified to correct the problem. We obtain source code line information using Dyninst [77] to tell the user the lines that would need to be modified to correct the problem. If source code information is not available, such as when the synchronization occurs in a proprietary binary like cuFFT, we report the offset address in the binary at which the transfer/allocation occurs. An example of a remedy generated by Diogenes for Qbox using cuFFT is shown in Figure 6.3. Figure 6.3 shows a stack of the call site of the problematic synchronous transfer (`cuMemcpyHtoD_v2` in `libcufftw`), a brief textual description of the problem, and the locations of memory management operations that would need to change to correct the issue.

```
void cusp::system::detail::generic::multiply<...>(...)  {
   // temporary_array<...> inherets thrust::temporary_array<...>
   // eventually resulting in cudaMalloc(...)  being called
   cusp::detail::temporary_array<...> rows(...);
   cusp::detail::temporary_array<...> vals(...);
   ...
   // cudaFree called by temporary_array object deconstructor
}
```

Figure 6.4: Example of an unnecessary synchronization as a result of memory management issue in cuIBM

6.2 Memory Management Issues

Frequent unnecessary synchronization caused by `cudaFree` operations is one of the most common mistakes that we detected in GPU programs. Two factors play a key role in the overuse of `cudaFree` operations: the application structure can hide where these operations take place and fixing problematic behavior requires modifying other operations in the program. Figure 6.4 is an excerpt from the application cuIBM that shows how identifying the location of `cudaFree` operations can be difficult in modern programs. The function `cusp::system::detail::generic::multiply<...>` is a template function in the header-only library CUSP [33] used by cuIBM. This function is called after multiple levels of template indirection. In this function, two `temporary_array` objects are instantiated. While the name implies that a temporary array will be created, it is only after several more layers of indirection and template function calls that a `cudaMalloc` is performed. Only when the `temporary_array` object is being destroyed, and after calling another several layers of destructors, is the `cudaFree` operation performed. An extra level of complexity is added when the compiler optimizes this code by removing some of these layers of indirection, increasing the difficulty in locating the problem.

Unlike other unnecessary synchronization operations that FFM detects,

cudaFree is unique in that there is no asynchronous version of the operation available. Fixing a synchronization issue at cudaFree is limited to removing or moving the operation. However, fixing a cudaFree operation cannot be done without also addressing the cudaMalloc that allocated the memory being freed. Without moving or removing the cudaMalloc, a memory leak would result.

6.2.1 Extensions to FFM for Memory Management Issues

Providing actionable feedback requires that we identify the cudaFree calls that perform unnecessary synchronization and identify the corresponding cudaMalloc calls.

The first step is to link the cudaMalloc call with the cudaFree call that freed the memory. We use a similar method to that used in identifying remedies for memory transfer issues described in Chapter 6.1.1. In stage 2, a simple dynamic data flow graph is constructed for cudaMalloc and cudaFree operations. Stage 5 was modified to generate a remedy that reports the cudaFree call was unnecessary along with the cudaMalloc call that allocted the memory being freed. Figure 6.5 shows an excerpt from the output of Diogenes for the program cumf_als. The output contains the stack with line numbers at which cudaFree was called, the type of problem idenfied (unnecessary synchronization), and the stack with line numbers for all GPU malloc sites that allocated memory freed at the cudaFree call site.

6.3 Unnecessary and Misplaced Explicit Synchronization Operations

Detecting unnecessary explicit synchronization operations, such as cuCtxSynchronize, was part of the original FFM design. While FFM could

```
cudaFree called at:
   doALS(...)  at line 1031 in als.cu
   main at line 146 in main.cpp
   ...
 Problem:  Unnecessary sync at cudaFree
   GPU Malloc Site:
     doALS(...)  at line 689 in als.cu
     main at line 146 in main.cpp
     ...
```

Figure 6.5: Example Diogenes remedy output for unnecessary synchronization operations caused by memory mangement issues in cumf_als

detect the presence of these operations, it did not give the user a remedy for these problems. FFM also did not differentiate between a synchronization operation that was unnecessary and one that needed to be moved. The result was that a developer needed to do manual analysis to determine the type of problem that was present and how to fix it. The extended FFM model addresses this issue by modifying stage 5 to output the type of problem present at the explicit synchronization (unnecessary or misplaced).

6.4 Experiments with Automatic Remedy Identification

We tested the effectiveness of the remedy identification by creating Diogenes 2.0 that implements the extended FFM model. We employed the modified version of Diogenes on three real world applications: cumf_als (git revision a5d918a), cuIBM (git revision 0b63f86), and Qbox (version r140b) using cufft (version 8.0) in compatibility mode. We described these applications in detail in Chapters 3 and 5.4.1. 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

Application Name	Source Size (lines of code)	FFM Perscribed Remedies		
		Memory Mgmt Problems	Transfer Sync Problems	Explicit Sync Problems
cumf_als	5K	22	3	10
cuIBM	36K	539	31	168
QBox	87K	0	79	1

Table 6.1: Number of remedies perscribed for synchronization problems identified by FFM

Pascal-class GPUs.

Table 6.1 summarizes the remedies prescribed by Diogenes 2.0 for each application. We categorize the remedies based on the type of problem: memory management, memory transfer, and explicit synchronization. Each remedy represents the suggested correction to a problem of a specified type that occured at a unique execution stack during program execution. If the same execution stack is responsible for multiple occurences of a problem, the remedies necessary to correct those occurrences are combined to form a single remedy that would address all occurrences.

We validated, via manual source code analysis, that the remedies identified in cumf_als and the memory transfer remedies identified in QBox addressed actual problems that existed in the program. Due to the complexity of the source code and high number of problems identified for cuIBM, we did not manually validate every remedy. However, for the problems that we employ autocorrection on (most memory and transfer issues), we validated that the program output remained identical. The performance benefit that we have observed from applying remedies to all three applications resulted in performance gains between 9% and 43%. We describe the performance obtained from actually applying these remedies in Chapter 7.

Our experiments for cumf_als were run using the GroupLens MovieLens 10M data set run with an iteration count of 5000. Diogenes 2.0

identified remedies for 22 memory management, 3 memory transfer, and 10 explicit synchronization issues. The memory management issues identified in cumf_als were primarily `cudaMalloc` and `cudaFree` pairs that were inside the main execution loop of the program. The memory transfer issues identified were caused by implicitly synchronous `cudaMemcpy` calls. In these instances, the CPU memory used in the transfer was already pinned and the problem was caused by not using the asynchronous version of the call. The explicit synchronization issues were primarily `cudaDeviceSynchronization` calls that were unnecessary. There were two memory transfer issues identified by Diogenes 1.0 detailed in Chapter 5.4.1 for which Diogenes 2.0 could not create remedies due to the inability to construct a complete data flow graph for the CPU memory used in these transfers. We believe the cause of this problem is memory that is not being freed before application exit. Dut to limitations of the construction of Diogenes, we cannot capture tracing data after the program calls exit. If memory is not freed by the time the application exits, Diogenes is not able to capture the location of the free operation since it never took place. We require complete information on where all memory addresses used in the transfer are allocated and freed to ensure that the remedy we generate is accurate.

The remedies identified for cuIBM and QBox addressed similar problematic behavior as those of cumf_als. For cuIBM, Diogenes 2.0 identified remedies for 539 memory management, 31 memory transfer, and 168 explicit synchronization issues. The major difference in the problems identified in cuIBM from those identified in cumf_als was the transfer remedies identified were primarily unnecessary synchronization caused by not using pinned memory with asynchronous memory transfer requests. Our experiments with cuIBM were run using the lid-driven cavity with Reynolds number 5000 dataset supplied in the public source code repository for cuIBM (*lidDrivenCavityRe5000*).

In QBox, Diogenes 2.0 identified remedies for 79 memory transfer and 1 explicit synchronization issue. The memory transfer remedies targeted implicitly synchronous `cuMemcpyHtoD` and `cuMemcpyDtoH` function calls that occured in the Nvidia `cufft` library that needed to both be converted to an asynchronous form and to use pinned memory in the transfer. Our experiments with QBox were run using the Gold 16 data set generated by the tool contained in the QBox distribution.

7 AUTOMATIC CORRECTION OF PROBLEMATIC OPERATIONS

Fixing problems identified by FFM can require a significant restructuring of application code or the modification of closed-source binaries. Developers are left with the tough choice of leaving these issues unresolved or potentially spending significant effort refactoring their code. If they choose to address the issues, the benefit they get may not have been worth the effort they place into fixing the problem. The high cost of developer time to fix problematic operations in combination with the potential risk of limited performance benefit results in the choice being made to not address these issues.

FFM provides an estimate of potential benefit to a developer to lessen the risk. While FFM is able to provide accurate estimates of benefit of fixing problematic operations, significant risk for the developer still remains. The developer needs to devise a plan on how to fix the issue, determine if that plan would be efficient enough to obtain the benefit FFM predicted, and then do the work to apply the fixes.

We created autocorrection techniques to lessen the risk to developers. Autocorrection works by identifying a class of fixable problematic operations in a program, selecting transformations that can be applied to correct the problems, and applying these transformations to the application binary to reveal the actual benefit that could be obtained. Autocorrection supplies the developer with transformations that can be used as starting point for the creation of a permanent solution. Unlike an estimate of benefit, this starting point is an actual benefit obtained if these transformations were made permanent to the program, reducing the risk to the developer that their time will be wasted. While the transformations implemented by autocorrection can be used as permanent solutions in some instances, application specific fixes may exist that a developer can identify that can

offer greater benefits. We are also still bound by the limitation of FFM in that it can only give assurances on transformation safety for the program inputs for which it has seen. Thus a developer still must ensure that these transformations are valid for other inputs not exercised by FFM, either by manual analysis or by reruning FFM with these inputs.

Autocorrection is performed using binary code modification that takes place at application startup. We focus on correcting the problematic synchronization operations caused by memory management and memory transfers issues. We focus on these issues since they are the most likely to require large structural changes to the application to resolve. The transformation we apply for memory management issues is to use a memory pool that limits the number of memory allocation and free operations that are passed to the GPU driver. This transformation eliminates a large percentage of overhead from excessive memory allocation and free opertions while also allowing us to selectively enable/disable synchronization behavior when it is required.

Our memory transfer transformations convert the call to its asynchronous form and then selectively apply a synchronization operation when required. Supporting the asynchronous call requires that we ensure that any CPU memory address passed to the transfer request falls on CUDA pinned page. If the CPU address is not on a pinned page, a temporary pinned page is allocated that will be used for the transfer.

Autocorrection takes place in two phases: the setup phase at application start-up to identify and apply the transformations to remedy the problem and the execution phase where the transformation applies the remedy when the operation is invoked by the application. In this chapter, we describe the model of application execution created to identify problematic operations and how these phases work together to remedy these common synchronization problems.

7.1 Model of Application Execution

To identify problematic operations in the program and identify the corrective measure to apply, we extend the model of application execution used defined in Chapter 5. This model is used by the setup and execution phases to apply corrections to the application. Recall that we model application execution as a graph $G = (N, V)$, where N is the set of operations performed by a processor and V is the set of edges.

We expand the definition of the existing attribute $NType$ and add the new attributes $CallStack$, $MemPtr$, and $Size$ to a node N. $NType$ is expanded to denote eight new operation types: a CPU memory operation ($CPUMalloc$ or $CPUFree$), a GPU memory operation ($GPUMalloc$ or $GPUFree$), a pinned page memory operation ($PinnedMalloc$ or $PinnedFree$), or a synchronous transfer operation ($SyncTransCPUtoGPU$ and $SyncTransGPUToCPU$). The $CallStack$ attribute denotes the current execution stack on the processor at the beginning of the operation. The $MemPtr$ and $Size$ attributes vary based the $NType$ of the node. For GPU memory operations, $MemPtr$ is the memory address created or freed. For transfer operations, $MemPtr$ is the CPU memory address used by the transfer.

7.2 Setup Phase of Autocorrection

The autocorrection process begins by using the model of application execution created in stages 1 through 4 of FFM to identify unnecessary synchronization operations. For use in autocorrection, we slightly alter FFM's model of application execution to generate call stacks for each unnecessary synchronization operation. The call stacks will be saved for use by the execution stage to determine what corrections are safe to apply at an instance of a synchronization. We also use the setup phase to insert

```
1  // uSync - Set of unnecessary sync ops stacks
2  uSync = []
3  // rSync - Set of necessary sync ops stacks
4  rSync = []
5
6  def AutocorrectSetup(Graph):
7    InterceptTransOps(TransIntercept)
8    InterceptGPUMemOps(GPUMemIntercept)
9    WrapPinnedMemoryOps(PinnedWrap)
10   PostCallSyncNotify(PostSynchronization)
11   for Node in Graph.N:
12     if (Node.NType == SyncTransCPUtoGPU or
13         Node.NType == SyncTransGPUToCPU or
14         Node.NType == GPUFree):
15         if (Node.Problem == Sync and !IsTransFromStack(Node)):
16           uSync = uSync U Node.CallStack
17         else:
18           rSync = rSync U Node.CallStack
19   uSync = uSync - rSync
20   WriteToFile(uSync)
```

Functions *TransIntercept*, *GPUMemIntercept*, *PinnedWrap*, and *PostSynchronization* are defined in Figure 7.2

Functions *InterceptTransOps*, *InterceptGPUMemOps*, *WrapPinnedMemoryOps*, *PostCallSyncNotify*, and *WriteToFile* are defined in Figure 7.3

Figure 7.1: Setup phase used to identify unnecessary synchronization operations and insert function wrappers to support Autocorrection

instrumentation into the application around the functions required to support autocorrection.

Figure 7.1 shows the algorithm used during the setup phase. In function `AutocorrectSetup` on line 7 and 8, we insert instrumentation to intercept calls to common transfer operations (such as `cuMemcpy` and `cuMemcpyAsync`) and GPU memory operations (such as `cuMemAlloc` and `cuMemFree`). The interception instrumentation modifies the code to redirect the calls made by the application to apply the appropriate remedy

at execution time. On line 9, we insert instrumentation to wrap common CUDA pinned page memory operations (such as `cuMemAllocHost` and `cuMemFreeHost`). The wrapping instrumentation captures the paramerters and return values of the wrapped call but does not alter the behavior of the call performed. The captured information is used to help identify the appropriate remedy at execution time. On line 10, we insert instrumenation at the exit of the internal GPU driver synchronization function to be notified when a synchronization operation has completed. At synchronization exit, we execute code to finalize the remedies applied during interception of problematic operations. The functions `TransIntercept`, `PinnedWrap`, `GPUMemIntercept`, and `PostSynchronization` are defined in the execution stage.

On lines 11-18, we iterate through the nodes in the graph to identify the synchronization operations that are problematic. We look at every transfer and memory free operation performed by the application, recording the call stack of the operation and whether the operation is performing a required synchronization. The necessity of the synchronization operation was determined by the original FFM analysis. A synchronization is deemed necessary by FFM if data protected by the synchronization is accessed. Data protected by the synchronization includes the CPU memory regions used in all pending memory transfers and the CPU memory regions of all memory pages shared between the CPU/GPU. A memory access by the CPU to the memory regions of protected data marks the synchronization as required. The call stacks later are used by the execution phase to determine if a synchronization operation should be skipped. On line 15, `IsTransFromStack` checks if the node is performing a GPU to CPU transfer with a destination on the CPU stack. We force a synchronization if a transfer has a destination of a CPU stack due to potential dangers that can occur with delaying writes to stack addresses during autocorrection. If the operation is performing an unnecessary synchronization, we add the call

stack to the set `uSync`. Likewise for required synchronization operations, we add the call stack to the set `rSync`. After we have iterated through the graph, we remove any call stack that appears in `rSync` from `uSync` to ensure that only unnecessary synchronization operations are remedied. On line 20, we write the call stacks out to a file to be read by the execution phase.

7.3 Execution Phase of Autocorrection

The execution phase identifies and corrects problematic operations that occur during program execution. We intercept potentially problematic operations that the application performs, such as memory allocation and free routines, and use the information collected during the setup phase to determine what corrective measure should be applied. For all synchronous operations intercepted, we change the default behavior to be asynchronous. We then use the data from the setup phase to identify if the intercepted call requires a synchronization. If a synchronization is required, we invoke an explicit synchronization operation before returning control back to the application.

Figure 7.2 shows the algorithm used during execution to correct synchronization problems in the program and apply general fixes to problematic behavior. Memory management operations are intercepted and modified to apply the appropriate remedy by the function `GPUMemIntercept` on line 31. Similarly, `TransIntercept` on line 13 intercepts and modifies memory transfer operations to apply remedies. The wrapper function `PinnedWrap` on line 6 supports the interceptor function `TransIntercept` by providing data on pinned memory allocation operations. The function exit wrapper `PostSynchronization` on line 26 notifies the execution phase of a synchronization and performs post synchronization tasks.

`GPUMemIntercept` (line 31) intercepts memory allocation and free op-

```
1  uSync = ReadFromFile()
2  Ordered.Map pinnedSet = ∅
3  // DelayedCopies - List of (TransPtr, TempPinnedPtr) pairs
4  DelayedCopies = []
5
6  def PinnedWrap(Node):
7    CallOriginalFunction(Node)
8    if Node.NType == PinnedMalloc:
9      pinnedSet[Node.MemPtr] = Node.size
10   else:
11     pinnedSet = pinnedSet - Node.MemPtr
12
13 def TransIntercept(Node):
14   TempPinnedPtr = Node.MemPtr;
15   if (pinnedSet ∩ Node.MemPtr == ∅):
16     TempPinnedPtr = GetPinnedMemFromPool(Node.TransSize)
17     if (Node.NType == SyncTransCPUtoGPU):
18       memcpy(TempPinnedPtr, Node.MemPtr)
19     else if (Node.NType == SyncTransGPUToCPU):
20       DelayedCopies = DelayedCopies ∪
21                       (Node.MemPtr, TempPinnedPtr)
22   AsyncTransfer(TempPinnedPtr, Node)
23   if (uSync ∩ CallStack == ∅):
24     PerformSynchronization()
25
26 def PostSynchronization():
27   for pair in DelayedCopies:
28     memcpy(pair.MemPtr, pair.TempPinnedPtr)
29   DelayedCopies = []
30
31 def GPUMemIntercept(Node)
32   if Node.NType == GPUMalloc:
33     return GetGPUMemFromPool(Node.size)
34   else:
35     ReturnGPUMemToPool(Node.MemPtr)
36     if (uSync ∩ CallStack == ∅)
37       PerformSynchronization()
```

Functions *ReadFromFile, IsPinnedPage, GetPinnedMemFromPool, PerformSynchronization, AsyncTransfer, GetGPUMemFromPool, ReturnGPUMemToPool* and *CallOriginalFunction* are defined in Figure 7.3

Figure 7.2: Execution Phase of Autocorrection

ReadFromFile() - reads call stacks from file provided by the setup phase.

IsPinnedPage(Map, Ptr) - returns true if Ptr is contained in Map

GetPinnedMemFromPool(size) - returns a temp pinned page from a memory pool (reclaimed when no longer used)

PerformSynchronization() - performs an explicit CPU/GPU synchronization

AsyncTransfer(CPUMemAddress, Node) - Performs an async transfer using the original parameters in node, replacing the CPU address used in the transfer with CPUMemAddress

GetGPUMemFromPool(size) - Get a GPU memory allocation with a specified size from a memory pool

ReturnGPUMemToPool(MemPtr) - Return memory address to memory pool

CallOriginalFunction(Node) - Calls the original function that was requested by the application

InterceptTransOps(TransIntercept) - Intercepts a set of known transfer ops

InterceptGPUMemOps(GPUMemIntercept) - Intercepts a set of known GPU memory ops

WrapPinnedMemoryOps(PinnedWrap) - Wraps a set of known pinned memory ops

PostCallSyncNotify(PostSynchronization) - Inserts a call to Function at end of sync

WriteToFile([Stacks]) - Writes the set of stacks performing unnecessary sync to a file.

IsTransFromStack(MemPtr) - Returns true if CPU stack address is used in a GPU to CPU transfer at Node

Figure 7.3: Auxiliary functions for the setup and execution phases of autocorrection

erations, redirecting these operations to use a memory pool. By using a memory pool, we limit the number of calls to `cuMemFree` made to the driver, reducing the number of synchronization operations that take place. When an allocation request is intercepted, we redirect the call to allocate memory using the memory pool (line 33). `cuMemAlloc` is called only if the memory pool does not have enough allocated memory to satisfy the request. When a free request is intercepted, we return the memory region to the memory pool (line 35). Since we may not call `cuMemFree`, and thus may not perform the implicit synchronization, we must check to see if the intercepted call requires a synchronization. We compare the call stack that initiated the request to the call stacks that were identified as unnecessary in the setup phase (line 36). If there is no match, we perform an explicit synchronization.

`TransIntercept` (line 13) intercepts and modifies memory transfer operations to remove unnecessary synchronization operations. We convert the synchronous memory copy operation to its asynchronous form, applying a synchronization only when it is required. Converting the call its asynchronous form requires that we first identify if the transfer is going to or from a CUDA managed pinned page. We compare the CPU memory pointer used in the transfer to a set of pinned pages allocated by the program (line 14). The set of allocated pinned pages (`pinnedSet`) is captured by the wrapper `PinnedWrap` on line 6. `PinnedWrap` inserts allocated memory ranges into a set (line 8) and removes those that are freed (line 10).

If the intercepted transfer request is not going to or from a pinned page CPU memory address, we must modify the transfer to use a temporary pinned page. The temporary pinned page stages the data being transferred to or from the GPU, allowing for the transfer to become asynchronous and accelerating the rate data is transferred. For transfers of data going to the GPU, the data to be transferred from the CPU is copied to this temporary

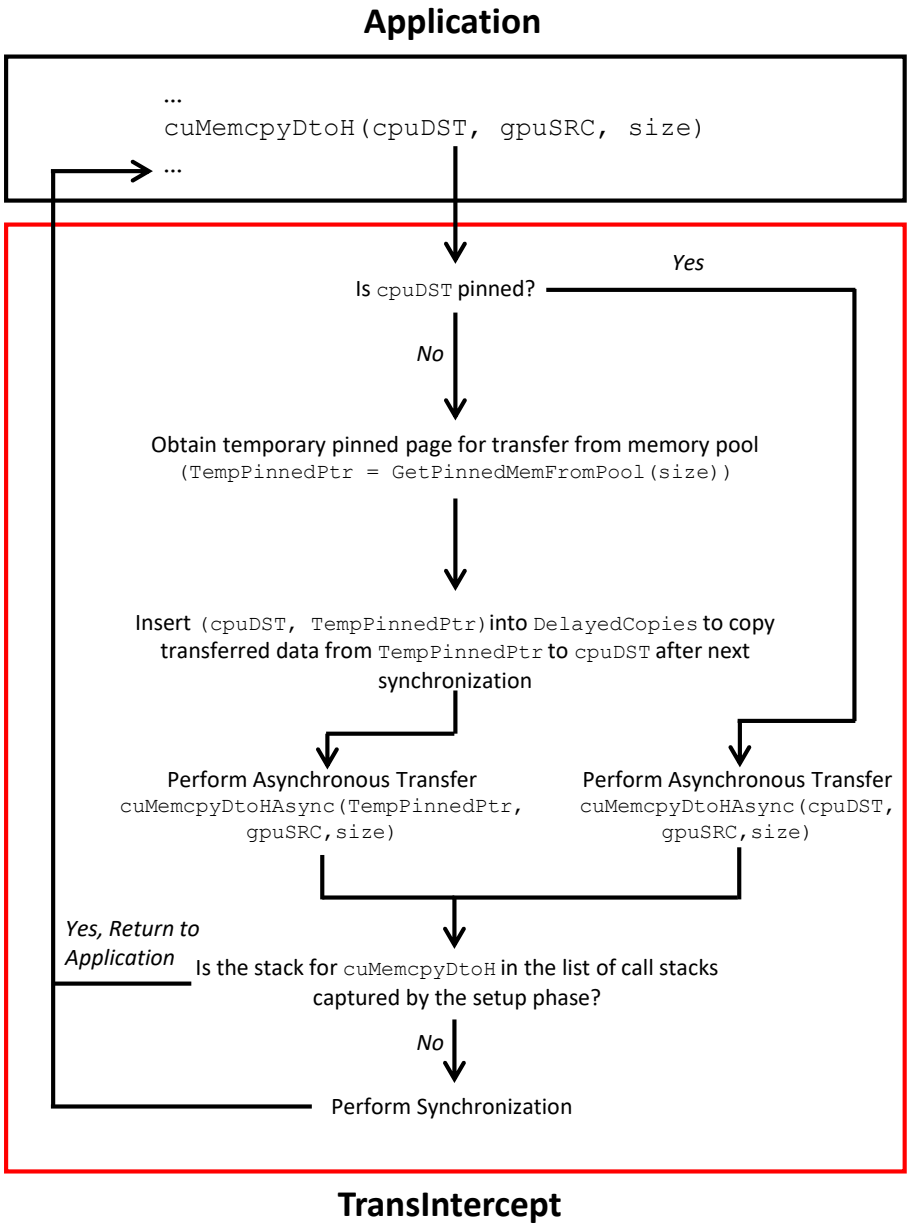

Figure 7.4: An ilustration of the processing steps performed by `TransIntercept` for intercepted `cuMemcpyDtoH` operations

page and the transfer request is modified to use the pinned page (lines 16-18). If instead the transfer is from the GPU to the CPU, we modify the transfer request to use the pinned page. However, the data transferred from the GPU is expected by the program to be at the CPU memory address used in the original transfer request. We ensure this behavior by delaying the copy from the temporary pinned page to the original CPU destination memory address to occur at the completion of the transfer at the next synchronization (line 20). The function `PostSynchronization` is called when a synchronization completes and the copy is performed (line 26-29). While the additional copy does add overhead to the operation, both the allocation of a pinned page and the copy operation would be performed by CUDA driver if we did not perform this ourselves. On line 22, we initiate the modified asynchronous transfer. On line 23-24, we determine if a synchronization must be performed. If the current execution stack is not contained in the unnecessary synchronization set collected during the setup phase, a synchronization is performed. In Figure 7.4, we show an illustration of the processing steps taken by `TransIntercept` for intercepted `cuMemcpyDtoH` operations.

7.4 Experiments with Autocorrection

We tested the effectiveness of the autocorrection on the applications cuIBM, cumf_als, and Qbox. All experiments were conducted using the same input datasets used and described in remedy identification experiments in Section 6.4. Table 7.1 summarizes the benefit obtained using autocorrection in these experiments. For each application, we list its unmodified execution time, the percentage of execution time saved in the original study of FFM by manually correcting a subset of problems in the program, and the percentage of execution time saved by using the autocorrection method. The use of autocorrection reduced execution time by 43.3% for

cuIBM, 33.2% for cumf_als, and 9.9% for Qbox. When compared to the results obtained using the original implementation of FFM, we obtained an additional 25.7% reduction in execution time using autocorrection for cuIBM and a 24.9% reduction for cumf_als. Qbox itself was not manually corrected as part of the original work on Diogenes. However, a reduction in execution time of 85% was achieved in Chapter 3 by modifying a few hundred lines of the FFT component of Qbox to use the native `cufft` interface. These changes required refactoring the code to use a library with different abstraction and manually managing the GPU memory and synchronization. The automatic corrections applied to Qbox do not require these modifications.

The major cause of the performance difference seen between manual correction using FFM and autocorrection is the larger number of problems that are actually corrected. The original FFM experiments focused on fixing only the top few problems with the largest potential performance benefit. This choice was made to mimic the typical behavior of a performance tool user who only typically fix the most problematic operations. This leaves, in some cases, hundreds of smaller issues that are viewed as not having large enough benefit to justify fixing them by themselves but can result in large aggregate benefit if they were all corrected. Autocorrection allows for these potential large gains available from fixing smaller issues to be exposed without having to perform the tedious repair of hundreds of smaller issues.

Table 7.2 summarizes the number of problems automatically remedied and the resulting number of synchronization operations that were eliminated by the applied remedies. We categorize the remedies using the same criteria as the remedy identification experiments: memory management problems and unnecessary synchronous memory transfer problems. Each remedy represents the correction of a problem of a specified type at a unique execution stack. *We verified the output of each application to ensure*

App Name	Original Exec Time (seconds)	Savings With Original FFM by Manual Correction (% of exec)	Savings With Autocorrection (% of exec)
cumf_als	1169	8.3%	33.2%
cuIBM	1909	17.6%	43.3%
Qbox	2243	No Manual Correction	9.9%

Table 7.1: Summary of the performance benfits obtained using autocorrection compared to the original implementation of FFM

that the remedies did not result in incorrect behavior.

Our experiments for cumf_als resulted in remedies being applied to 22 memory management and 3 memory transfer issues. The 22 remedies applied to memory management issues intercepted approximately 85K calls to `cudaMalloc` and 85K calls to `cudaFree`. These operations were primarily the `cudaMalloc/cudaFree` pairs inside of the main execution loop of the program that we described in the remedy identification phase. The result of this remedy was the elimination of 85K synchronization operations that took place unnecessarily at `cudaFree` operations. The 3 remedies applied to memory transfer issues removed an additional 45K synchronization operations. Total benefit obtained was a reduction in execution time by 33.2%.

cuIBM shows an extreme example of unnecessary synchronization caused by `cudaFree` operations. Remedies were applied to 539 problematic synchronization operations occuring at `cudaFree`. The application of the remedies resulted in the interception of 45 million calls to `cudaMalloc` and 45 million calls to `cudaFree`, removing 45 million synchronization operations. Remedies applied to the 31 memory transfer problems resulted in 32 synchronization operations being removed. Total benefit obtained was a reduction in cuIBM's execution time by 43.3% for the *lidDrivenCavityRe5000* input dataset.

	Memory Management		Memory Transfer	
Application Name	Problems Remedied	Sync Ops Removed	Problems Remedied	Sync Ops Removed
cumf_als	22	85,005	3	45,005
cuIBM	539	45,290,724	31	32
Qbox	0	0	79	32,048,836

Table 7.2: Synchronization operations removed using autocorrection

We also saw a reduction in cuIBM's execution time with other input datasets using the same autocorrection generated to resolve the problems seen with `lidDrivenCavityRe5000`. We obtained a 58% reduction in execution time with `cylinderRe3000`, a 48% reduction with `flyingSnakeRe2000 AoA30`, and a 43% reduction with `lidDrivenCaveityRe3200`. These results suggest that the problems identified and the autocorrection generated for cuIBM with the `lidDrivenCavityRe5000` input dataset address problems that are not unique to a specific dataset and would benefit application performance more broadly if corrected.

Qbox shows an extreme example of unnecessary synchronization caused by memory transfer operations. Remedies were applied to 79 problematic synchronization operations that occured at various memory transfer operations (such as `cuMemcpyHtoD`). The result was the elimination of 32 million synchronization operations, reducing execution time by 9.9%.

7.5 Limiations of Diogenes 2.0 and Next Steps

Similar to Diogenes 1.0, the overhead of running Diogenes 2.0 is significantly higher than that of other performance tools. The overhead of running the entire extended FFM model was between 7x (for cumf_als) to 45x (for Qbox) of execution time. While the cost of FFM is high in terms of time, the targeted feedback that a tool user receives and the performance

benefits that can be obtained can save programmer time.

While cumf_als and cuIBM showed a significant reduction in application execution time beyond what was obtained using manual correction, Qbox fell short of the 85% reduction in execution time achieved in Chapter 3. We believe the primary reason for this shortfall was the inability to automatically remove duplicate transfers. In Qbox, there are millions of transfers that duplicate either full or partial data of prior transfers. The conversion in Chapter 3 used a new interface that allowed us to directly control when data was transferred between the CPU and GPU, giving us the ability to minimize the occurrence of duplicate transfers. A method to automatically correct duplicate data transfers would likely see the automatic correction approach close in on the benefits seen with the manual method.

8 CONCLUSION

Our goal for this research has been to create performance tool techniques that can help developers adapt their applications to more effectively exploit the additional parallelism afforded by many-core architectures. The guiding principle of our work was to deliver actionable feedback to developers on the problems present in their applications, exposing previously hidden performance opportunities that would offer substantial performance benefits if corrected. The result was the development of the feed-forward measurement model capable of guiding developers to problematic operations in their program and delivering actionable feedback in the form of an estimate of expected benefit if the problem were corrected. We expanded this work to automatically identify how to remedy problematic operations and a method for automatically applying these remedies to the program. In this chapter, we summarize the technical contributions and future research directions of our work.

8.1 Contributions

In this dissertation, we presented five main contributions:

Exposing hidden performance problems in GPU applications: We identified four performance issues that impact high performance scientific applications that use GPUs for computation: unobvious missed parallelization opportunities, duplicate data transfers, synchronization issues, and JIT compilation. The manual correction of the problems we discovered resulted in 18%-87% reduction in program execution time. What links the issues together is the lack of performance tools and techniques to detect their presence. We have developed

techniques that can detect when and where these issues are present within applications.

The feed-forward measurement model: We introduce a multi-stage, multi-run performance measurement and analysis approach called the feed-forward measurement model (FFM). FFM automates the identification of unnecessary synchronization and memory transfer operations in GPU programs. We focused on these problems because they were the most common and most performance detrimental in the applications we studied. 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 measurement collection frameworks for data collection, instead FFM uses binary instrumentation to directly capture and time events such as synchronization operations. 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.

The performance model of FFM: We created a performance model that uses the data collected by FFM to provide an estimate of expected benefit that could be obtained if a problematic operation were fixed. We model application execution to determine the effect that (re)moving a problematic operation will have on other operations in the program and overall application execution time. The performance 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 real world applications. Diogenes was able to provide accurate feedback (around 77% combined accuracy across the applications that we studied) 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%.

Automatic remedy identification: We extended FFM model to identify if the synchronization problem is a component of a larger construct that exhibits a problem that can span over many operations and automates the identification of remedies that can correct the issue. We focused on identifying the cause of four of the most common types of synchronization problems we have seen in the real-world applications: 1) memory transfer issues, 2) memory management issues, 3) unnecessary operations, and 4) misplaced operations. Diogenes 2.0 implements the extended FFM model to automatically identify remedies to problematic synchronization operations. Using this implements, we were able automatically identify remedies for several hundred synchronization issues across three applications.

Automatic correction of problematic synchronization operations: We further extended the FFM model to automatically correct synchronization problems. We focused our efforts on automatically correcting synchronization problems caused by memory transfer and memory management issues. These larger constructs exhibit a problem that span over many operations, making them more difficult to fix manually. We implemented this extension in Diogenes 2.0 and were able to automatically reduce application execution time by 7% to 43% in the applications that we have studied.

8.2 Future Directions

There are several opportunities for future expansion of the work and ideas we have presented here.

Expanding FFM to new problem types: FFM's ability to adjust instrumentation based on program behavior and spread data collection over multiple runs opens the possibility for creating new tools that deliver actionable feedback on other problems. One such potential problem is missed parallelization opportunities. One profiling run of the program would be used to collect the detailed data needed to detect memory access patterns favorable to parallelization. A second profiling run would be used to collect measurements sensitive to CPU overhead such as the GPU activity taking place during the loops execution. There would still be challenges that need to be overcome, such as how to estimate the GPU execution time of a CPU loop, but FFMs framework would jump start the development of this tool.

Reducing Execution Overhead of FFM/Diogenes: One of the key limitations of FFM is the overhead we observed. A majority of this overhead is caused by load/store analysis performed to detect unnecessary synchronization operations. Creating new techniques to collect this data or finding methods to limit its use would expand the applications that can be run with FFM. One path that warrants exploration is limiting the use of load/store analysis so that it is performed for only limited number of occurrences of a synchronization operation with a unique stack trace. What we have seen in practice is that a synchronization operation that is necessary is often necessary every time it occurs in the program. Turning on load/store analysis only when a synchronization operation with a unique stack trace is seen could substantially reduce overhead. The limitation of this approach is that it opens the possibility for incorrectly marking a synchronization operation as being unnecessary. However, this a trade off that a user might be willing to accept if it allows them to run the tool on programs they otherwise would not be able to.

The recent introduction of a new hardware primitive on Intel x86-64

processors called Memory Protection Keys [42] (MPK) is another promising option for reducing the cost of load/store analysis phase. MPK allows for memory regions to be protected using unprivileged CPU instructions that can be called by the application directly without involving the operating system kernel. Using MPK, we could protect the memory regions containing GPU-computed data and wait for hardware to signal that an instruction is trying to access that data. This would allow us to capture the data we need while leveraging the extremely low overhead of MPK, shown in a study to be less than 1% of application execution time [71]. The limitation of this approach is that only Intel server processors currently support this feature. Until other vendors such as AMD also support this feature, using MPK would limit its applicability.

Extraction of FFM components for use by other performance tools: During the course of our work on FFM, several techniques were created that could be used by other performance tools to improve their data collection and analysis. Two techniques that we feel would have significant impact on existing tools are the direct measurement of synchronization operations and the performance model. To our knowledge, FFMs technique for identifying and measuring synchronization operations is the only alternative method for measuring synchronization delay outside of vendor-supplied performance measurement collection frameworks. This technique can supply a more detailed and accurate picture of the synchronization operations than those supplied by the vendor and would immediately impact techniques such as blame analysis that can make use of this information. We are in the process of extracting this measurement capability into a stand-alone measurement tool that other tools and techniques can utilize to collect this information. We believe the performance model could be used by other performance tools to provide better informa-

tion on the expected benefit of fixing a performance problem.

Expanding FFM to new platforms: FFM was developed and its implementation Diogenes tested on PowerPC platforms with Nvidia GPUs. We focused our development efforts on this combination of CPU and GPU architecture due to its prominence in today's leadership-class high performance computing platforms. However, the upcoming release of exascale high performance computing platforms sees a significant shift in both CPU and GPU architectures. x86-64 based CPUs paired with Intel and AMD GPUs are slated to be used in at least two of the upcoming major exascale machines. Given the increase in architecture diversity of exascale machines, architecture dependencies contained within FFM and Diogenes will need to be addressed. We believe that only minor changes to FFM and Diogenes will be required to support these platforms but the lack of information on the GPUs to be used in these machines leaves us uncertain on exactly what will need to be changed.

REFERENCES

[1] Adhianto, L., S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. In *Concurrency and Computation: Practice and Experience*, 22(6), April 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:6.

[2] Advanced Micro Devices. *AMD uProf User Guide*. 3.2 edition, 2019.

[3] Advanced Micro Devices. ROCm, a New Era in Open GPU Computing, 2020. URL `https://rocm.github.io/packages.html`.

[4] Anderson, J. A., C. D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. In *Journal of Computational Physics*, 227(10), May 2008. ISSN 0021-9991.

[5] Anderson, T. E., and E. D. Lazowska. Quartz: A Tool for Tuning Parallel Program Performance. In *The Proceedings of the 1990 Conference on Measurement and Modeling of Computer Systems*, (SIGMETRICS '90), Boulder, Colorado, May 1990. ISBN 0-89791-359-0. doi: 10.1145/98457.98518.

[6] Ansel, J., C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *the Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (PLDI '09), Dublin, Ireland, June 2009. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542481.

[7] Ansel, J., Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and Compiler Support for Auto-tuning

Variable-accuracy Algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*, April 2011. doi: 10.1109/CGO.2011.5764677.

[8] Ansel, J., S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *in the Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques*, (PACT '14), August 2014. doi: 10.1145/2628071.2628092.

[9] Ardalani, N., C. Lestourgeon, K. Sankaralingam, and X. Zhu. Cross-architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance. In *the Proceedings of the 48th International Symposium on Microarchitecture*, (MICRO '15), Waikiki, Hawaii, December 2015. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830780.

[10] Atkins, M., and R. Subramaniam. PC Software Performance Tuning. In *Computer*, 29(8), August 1996. ISSN 1558-0814. doi: 10.1109/2.532045.

[11] Ayers, A., R. Schooler, and R. Gottlieb. Aggressive Inlining. In *the Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, (PLDI '97), Las Vegas, NV, June 1997. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258928.

[12] Baldini, I., S. J. Fink, and E. Altman. Predicting GPU Performance from CPU Runs Using Machine Learning. In *the Proceedings of the 26th IEEE International Symposium on Computer Architecture and High Performance Computing*, (SBAC-PAD '14), October 2014. doi: 10.1109/SBAC-PAD.2014.30.

[13] Behzad, B., H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. Taming Parallel I/O Complexity with Auto-

tuning. In *the Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, (SC'13), Denver, CO, November 2013. doi: 10.1145/2503210.2503278.

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

[15] Carrington, L., M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An Idiom-finding Tool for Increasing Productivity of Accelerators. In *the Proceedings of the 25th International Conference on Supercomputing*, (ICS '11), Tucson, Arizona, June 2011. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995928.

[16] Chabbi, M., K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective sampling-driven performance tools for gpu-accelerated supercomputers. In *the Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage and Analysis*, (SC '13), Denver, CO, November 2013. Association for Computing Machinery. ISBN 9781450323789. doi: 10.1145/2503210.2503299.

[17] Chang, P. P., S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. In *Software: Practice and Experience*, 21(12), December 1991. doi: 10.1002/spe.4380211204. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211204`.

[18] Chang, P. P., S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. In *Software: Practice and Experience*, 22(5), May 1992. doi: 10.1002/spe.4380220502.

[19] Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *the Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, (IISWC '09), Austin, TX, October 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: 10.1109/IISWC.2009.5306797.

[20] Chen, D., N. Vachharajani, R. Hundt, S. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming Hardware Event Samples for FDO Compilation. In *the Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, (CGO '10), Toronto, Canada, April 2010. Association for Computing Machinery. ISBN 9781605586359. doi: 10.1145/1772954.1772963.

[21] Chen, D., T. Moseley, and D. X. Li. AutoFDO: Automatic Feedback-directed Optimization for Warehouse-scale Applications. In *the Proceedings of the 2016 IEEE/ACM International Symposium on Code Generation and Optimization*, (CGO '16), Barcelona, Spain, March 2016.

[22] Chen, L., O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *the Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing*, (IPDPS '10), Atlanta, GA, April 2010. doi: 10.1109/IPDPS.2010.5470413.

[23] Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. In *Computing Research Repository*, abs/1410.0759, October 2014.

[24] Cohen, W. E. Tuning Programs with OProfile. In *WIDE OPEN MAGAZINE*, 1(1), January 2004.

[25] Cohn, R., and P. G. Lowney. Feedback Directed Optimization in Compaq's Compilation Tools for Alpha. In *in The 2nd ACM Workshop on Feedback-Directed Optimization*, (FDO '99), Haifa, Israel, November 1999.

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

[27] Ding, Y., J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe. Autotuning Algorithmic Choice for Input Sensitivity. In *SIGPLAN Not.*, 50(6), June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737969.

[28] Draeger, E. W., X. Andrade, J. A. Gunnels, A. Bhatele, A. Schleife, and A. A. Correa. Massively parallel first-principles simulation of electron dynamics in materials. In *the Proceedings of the 2016 International Parallel and Distributed Processing Symposium*, (IPDPS '16), Chicago, IL, May 2016.

[29] Ester, Martin, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *the Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, (KDD '96), August 1996.

[30] Fan, Z., F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *the Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, (SC '04), Pittsburgh, PA, November 2004. ISBN 0-7695-2153-3. doi: 10.1109/SC.2004.26. URL `https://doi.org/10.1109/SC.2004.26`.

[31] Frigo, M., and S. G. Johnson. The design and implementation of FFTW3. In *the Proceedings of the IEEE*, 93(2), February 2005.

[32] Fujii, Y., T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data Transfer Matters for GPU Computing. In *the 2013 International Conference on Parallel and Distributed Systems*, (ICPADS '13), Seoul, South Korea, December 2013.

[33] Galvez, R., and G. V. Anders. Accelerating the Solution of Families of Shifted Linear Systems with CUDA. Technical Report 1102.2143, arXiv, February 2011.

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

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

[36] Google. Using Cloud TPU Tools, 2020. URL `https://cloud.google.com/tpu/docs/cloud-tpu-tools`.

[37] Graham, S. L., P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *SIGPLAN Not.*, 17(6), June 1982. ISSN 0362-1340. doi: 10.1145/872726.806987.

[38] Guarna, V. A., D. Gannon, D. Jablonowski, A. D. Malony, and Y. Gaur. Faust: An Integrated Environment for Parallel Programming. In *IEEE Software*, 6(4), July 1989. ISSN 1937-4194. doi: 10.1109/52.31649.

[39] Gygi, F. Architecture of Qbox: A Scalable First-principles Molecular Dynamics Code. In *IBM Journal of Research and Development*, 52(1), January 2008. ISSN 0018-8646.

[40] Harper, F. M., and J. A. Konstan. The MovieLens Datasets: History and Context. In *ACM Transactions on Interactive Intelligent Systems*, 5 (4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872.

[41] Homescu, A., S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided Automated Software Diversity. In *the Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, (CGO '13), February 2013. doi: 10.1109/CGO.2013.6494997.

[42] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325384-071US. October 2016.

[43] Joubert, W., R. Archibald, M. Berrill, W. M. Brown, M. Eisenbach, R. Grout, J. Larkin, J. Levesque, B. Messer, M. Norman, B. Philip, R. Sankaran, A. Tharrington, and J. Turner. Accelerated application development: The ORNL Titan experience. In *Computers and Electrical Engineering*, 46, August 2015. ISSN 0045-7906. doi: http://dx.doi.org/10.1016/j.compeleceng.2015.04.008.

[44] Keutzer, K., B. L. Massingill, T. G. Mattson, and B. A. Sanders. A Design Pattern Language for Engineering (Parallel) Software: Merging the PLPP and OPL Projects. In *the Proceedings of the 2010 Workshop on Parallel Programming Patterns*, (ParaPLoP '10), Carefree, Arizona, March 2010. ISBN 978-1-4503-0127-5. doi: 10.1145/1953611.1953620.

[45] Kiss, A., J. Jasz, G. Lehotai, and T. Gyimothy. Interprocedural Static Slicing of Binary Executables. In *the Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, (SCAM '03), Amsterdam, NL, September 2003. doi: 10.1109/SCAM.2003.1238038.

[46] Knüpfer, A., H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis

tool-set. In *Tools for High Performance Computing*, Berlin, Heidelberg, July 2008. Springer Berlin Heidelberg. ISBN 978-3-540-68564-7.

[47] Knüpfer, A., 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. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *the Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*, Berlin, Heidelberg, September 2011. ISBN 978-3-642-31476-6. doi: 10.1007/978-3-642-31476-6_7.

[48] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. In *Communications of the ACM 21, 7 (July 1978), 558-565. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.*, July 1978.

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

[50] Layton, S., A. Krishnan, and L. A. Barba. *cuIBM Source Code Repository*. Commit 0b63f86 edition, 2019. URL `https://github.com/barbagroup/cuIBM`.

[51] Li, D. X., R. Ashok, and R. Hundt. Lightweight Feedback-Directed Cross-Module Optimization. In *the Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, (CGO '10), Toronto, Ontario, April 2010. Association for Com-

puting Machinery. ISBN 9781605586359. doi: 10.1145/1772954.1772964. URL https://doi.org/10.1145/1772954.1772964.

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

[53] Malony, A. D., S. Biersdorff, W. Spear, and S. Mayanglambam. An Experimental Approach to Performance Measurement of Heterogeneous Parallel Applications Using CUDA. In *the Proceedings of the 24th ACM International Conference on Supercomputing*, (ICS '10), Tsukuba, Ibaraki, Japan, June 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810105.

[54] Malony, A. D., S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *the Processing of the 2011 International Conference on Parallel Processing*, (ICPP '11), Taipei City, Taiwan, September 2011. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.71.

[55] McFarling, S. Program Optimization for Instruction Caches. In *the Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS III), Boston, MA, April 1989. ACM. ISBN 0-89791-300-0. doi: 10.1145/70082.68200.

[56] Mellor-Crummey, J., R. Fowler, and D. Whalley. Tools for Application-Oriented Performance Tuning. In *the Proceedings of the 15th International Conference on Supercomputing*, (ICS '01), Sorrento, Italy, June 2001. ISBN 1-58113-410-X. doi: 10.1145/377792.377826.

[57] Miller, B. P., M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. In *IEEE Computer*, 28(11), November 1995. ISSN 1558-0814. doi: 10.1109/2.471178.

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

[59] Morajko, A., P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: Monitoring, Analysis and Tuning Environment for parallel/distributed applications. In *Concurrency and Computation: Practice and Experience*, 19(11), October 2007. doi: 10.1002/cpe.1126.

[60] Nickolls, J., and W. J. Dally. The GPU Computing Era. In *IEEE Micro*, 30(2), March 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.41.

[61] Nugteren, C., and H. Corporaal. A Modular and Parameterisable Classification of Algorithm. Technical Report ESR-2011-02, Eindhoven University of Technology, Eindhoven, Netherlands, February 2011.

[62] Nvidia. *The CUBLAS Library*. 8.0 edition, 2016.

[63] Nvidia. *CUDA Compiler Driver NVCC - Reference Guide*. 8.0 edition, 2016.

[64] Nvidia. *The Cuda FFT Library*. 9.2 edition, 2018.

[65] Nvidia. *The CUDA Profiling Tools Interface*. 9.2 edition, 2018.

[66] Nvidia. *The Nvidia CUDA Profiler Users Guide*. 9.2 edition, 2018.

[67] Nvidia. *The Thrust Quick Start Guide*. 9.2 edition, 2018.

[68] Olschanowsky, C., A. Snavely, M. R. Meswani, and L. Carrington. PIR: PMaC's Idiom Recognizer. In *the Proceedings of the 39th International Conference on Parallel Processing Workshops*, (ICPP '10), San Diego, CA, September 2010. doi: 10.1109/ICPPW.2010.36.

[69] Ottoni, G., and B. Maher. Optimizing Function Placement for Large-scale Data-center Applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, February 2017. doi: 10.1109/CGO.2017.7863743.

[70] Panchenko, M., R. Auler, B. Nell, and G. Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *the Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, (CGO '19), Washington, DC, USA, February 2019. IEEE Press. ISBN 978-1-7281-1436-1.

[71] Park, Soyeon, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference*, (USENIX 19), Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/park-soyeon`.

[72] Petersen, P. Intel® parallel studio. In *the Encyclopedia of Parallel Computing*. November 2011.

[73] Pettis, K, and R. C. Hansen. Profile Guided Code Positioning. In *the Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, (PLDI '90), White Plains, NY, June 1990. ACM. ISBN 0-89791-364-7. doi: 10.1145/93542.93550.

[74] Pillet, V., J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *the Proceedings of the 18th*

Technical Meeting on Transputer and Occam Developments), (WoTUG '18), Manchester, England, April 1995.

[75] Plimpton, S. Fast Parallel Algorithms for Short-Range Molecular Dynamics. In *Journal of Computational Physics*, 117(1), March 1995. ISSN 0021-9991. doi: 10.1006/jcph.1995.1039.

[76] Poliakoff, D., and M. LeGendre. Gotcha: An Function-Wrapping Interface for HPC Tools. In *the Proceedings of the International Workshop on Visual Performance Analysis*, (VPA '19), April 2019. ISBN 978-3-030-17872-7.

[77] Project, Paradyn. *Dyninst: Putting the Performance in High Performance Computing*. URL `http://www.dyninst.org`.

[78] Quinlan, S., and S. Dorward. Venti: A New Approach to Archival Storage. In *the Proceedings of the 2002 Conference on File and Storage Technologies*, (FAST '02), Monterey, CA, January 2002. ISBN 1-880446-03-0.

[79] Samples, D. A, and P. N Hilfinger. Code Reorganization for Instruction Caches. Technical Report UCB/CSD-88-447, University of California at Berkeley, October 1988.

[80] Schmitt, F., R. Dietrich, and G. Juckeland. Scalable Critical Path Analysis for Hybrid MPI-CUDA Applications. In *the Proceedings of the 2014 IEEE International Parallel Distributed Processing Symposium Workshops*, (IPDPS '14), Phoenix, AZ, May 2014. doi: 10.1109/IPDPSW.2014.103.

[81] Shen, D., S. L. Song, A. Li, and X. Liu. CUDAAdvisor: LLVM-based Runtime Profiling for Modern GPUs. In *the Proceedings of the 2018 International Symposium on Code Generation and Optimization*, (CGO

2018), Vienna, Austria, February 2018. ISBN 978-1-4503-5617-6. doi: 10.1145/3168831.

[82] Shende, S., and A. D. Malony. The TAU Parallel Performance System. In *The International Journal of High Performance Computing Applications*, Vol 20(Num 2), May 2006.

[83] SoftBank Group. *Allinea Forge User Guide*. 7.0 edition, 2016.

[84] Tan, W., S. Chang, L. Fong, C. Li, Z. Wang, and L. Cao. Matrix Factorization on GPUs with Memory Optimization and Approximate Computing. In *the Proceedings of the 47th International Conference on Parallel Processing*, (ICPP '18), Eugene, OR, USA, 2018. ACM. ISBN 978-1-4503-6510-9. doi: 10.1145/3225058.3225096.

[85] Tiwari, A., C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A Scalable Auto-tuning Framework for Compiler Optimization. In *the Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, (IPDPS '09), May 2009. doi: 10.1109/IPDPS.2009.5161054.

[86] Waldspurger, C. A. Memory Resource Management in VMware ESX Server. In *the 5th symposium on Operating Systems Design and Implementation*, (OSDI '02), Boston, MA, December 2002. doi: 10.1145/844128.844146.

[87] Wang, Y., and D. Kaeli. Profile-guided I/O Partitioning. In *the Proceedings of the 17th Annual International Conference on Supercomputing*, (ICS '03), San Francisco, CA, June 2003. ACM. ISBN 1-58113-733-8. doi: 10.1145/782814.782850.

[88] Weiser, M. Program Slicing. In *the Proceedings of the 5th International Conference on Software Engineering*, (ICSE '81), San Diego, CA, March 1981. ISBN 0-89791-146-6.

[89] Welton, B., and B. P. Miller. The Anatomy of Mr. Scan: A Dissection of Performance of an Extreme Scale GPU-based Clustering Algorithm. In *the Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, (ScalA '14), November 2014. ISBN 978-1-4799-7562-4. doi: 10.1109/ScalA.2014.10.

[90] Welton, B., and B. P. Miller. Data Reduction and Partitioning in an Extreme Scale GPU-Based Clustering Algorithm. In *the 2nd Workshop on Data Reduction for Big Scientific Data*, (DRBSD 2), November 2017.

[91] Welton, B., and B. P. Miller. Exposing Hidden Performance Opportunities in High Performance GPU Applications. In *the Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, (CCGRID '18), Washington, D.C., May 2018. doi: 10.1109/CCGRID.2018.00045.

[92] Welton, B., and B. P. Miller. Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool. In *the Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage, and Analysis*, (SC '19), Denver, CO, November 2019. doi: 10.1109/CCGRID.2018.00045.

[93] Welton, B., and B. P. Miller. Identifying and (Automatically) Remedying Performance Problems in CPU/GPU Applications. In *Under Submission*, 2020.

[94] Welton, B., E. Samanas, and B. P. Miller. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *the Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage and Analysis*, (SC '13), Denver, Colorado, November 2013. ACM. ISBN 978-1-4503-2378-9. doi: 10.1145/2503210.2503262.

[95] Westerdahl, M. xxhash - xxh64. `http://www.xxhash.com/`.

[96] Wienke, S., P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-world Applications. In *the 18th International Conference on Parallel Processing*, (EuroPar '12), August 2012. ISBN 978-3-642-32819-0. doi: 10.1007/978-3-642-32820-6_85.

[97] Williams, S., L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In *the Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, (SC '07), November 2007. doi: 10.1145/1362622.1362674.

[98] Wu, C E., A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems. In *the Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, (SC '00). IEEE Computer Society, November 2000.

[99] Yang, C.-Q., and B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *the Proceedings of the 8th International Conference on Distributed Computing Systems*, (ICDCS '88), San Jose, CA, June 1988. doi: 10.1109/DCS.1988.12538.

[100] Yang, U. AMG: Algebraic Multigrid Benchmark. `https://github.com/LLNL/AMG`, 2018.

[101] Yixun L., E. Z. Zhang, and X. Shen. A Cross-input Adaptive Framework for GPU Program Optimizations. In *the Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*, (IPDPS '09), May 2009. doi: 10.1109/IPDPS.2009.5160988.