Identifying and (Automatically) Remedying Performance Problems in CPU/GPU Applications

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

ABSTRACT

GPU accelerators have become common on today's leadership-class computing platforms. Effective exploitation of the additional parallelism offered by GPUs is fraught with challenges. A key performance challenge faced by developers is how to limit the time consumed by synchronizations between the CPU and GPU. We introduce the extended feed-forward measurement (FFM) performance tool that provides an automated detection of synchronization problems, identifies if the synchronization problem is a component of a larger construct that exhibits a problem beyond an individual synchronization operation, identifies remedies that can correct the issue, and in some cases automatically applies remedies to problems exhibited by larger constructs. The extended FFM performance tool identifies three causes of unnecessary synchronizations: a problem caused by a single operation, a problem caused by memory management issues, and a problem caused by a memory transfer. The extended FFM model prescribes remedies for each construct and can automatically apply remedies for memory management and memory transfer cause problems. We created an implementation of the extended FFM performance tool and employed it to identify and automatically correct problems in three real-world scientific applications, resulting in an automatically obtained reduction in execution time between 9% and 43%.

CCS CONCEPTS

• **Software and its engineering** → *Software performance*;

KEYWORDS

GPUs, Performance Tools, Performance Analysis, Feed-Forward Measurement, Autocorrection

ACM Reference Format:
Benjamin Welton and Barton P. Miller. 2020. Identifying and (Automatically) Remedying Performance Problems in CPU/GPU Applications. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3392717.3392759

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

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392759

1 INTRODUCTION

The inclusion of GPUs on today's leadership-class computing platforms has increased the types of performance problems that can effect an application. One challenge is handling synchronization between the CPU/GPU. Synchronization operations reduce CPU/GPU overlap and can have a large impact on application performance. Knowing the high cost of synchronization operations, developers try to ensure that synchronization is performed only when necessary. Despite these efforts, problematic synchronization can consume up to 50% of execution time in real world applications [36, 37]. Developers need support to both identify the places in their code responsible for unnecessary synchronization delays and to develop fixes to remediate these delays.

Previous research on performance tools developed a technique called *Feed Forward Measurement* [37] (FFM), showing how to identify these places in code and provided reliable estimates of the benefit that could be obtained if they were remedied. While FFM was able to identify problems missed by other approaches, there were still gaps in its analysis that resulted in performance opportunities being left unexposed.

While FFM could say that a synchronization was problematic, it could not identify if the synchronization is a component of a larger construct that exhibit a problem that can span over 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.

Once a problem is identified by FFM, it was left to the developer to determine what remedy to employ to fix the problem. Potential remedies include removing the synchronization, moving the synchronization to a more advantageous location, or leaving the synchronization in place if the benefit of changing the behavior is too low to be worth the effort. If a larger problem is present, a single remedy might exist that could fix the larger problem. The lack of guidance on what remedy to select can result in the developer selecting the wrong remedy, lowering performance benefit or impacting program correctness.

When a developer 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.

To address these problems we have developed the extended FFM model capable of identifying larger constructs within an application that exhibit a problem, identifying the remedy that should be employed to fix the problem, and automatically applying the remedy to problems exhibited by the larger constructs. We focused our efforts on giving FFM the capability to detect two larger problems that are common in applications today: unnecessary synchronizations caused by memory management issues spanning multiple operations and by memory transfers. These problems are by far the most common and costly larger constructs exhibiting a synchronization problem present in the real world applications we have encountered.

We identify the presence of these larger constructs by creating a graph tracking allocation/free operations along with their usage in various GPU operations. The graph is annotated using memory tracing data collected by an extended version of FFM to identify the memory operations that are associated with an unnecessary synchronization. The annotated graph is used to determine which operations are responsible for the synchronization, allowing us to determine if it is component of a larger problem.

For a synchronization problem that FFM identifies, whether it is a part of a larger construct or not, we automatically identify the remedy that should be employed to fix the problem. There are four corrective actions that can be prescribed to remedy a problem: 1) remove the operation performing the synchronization, 2) move the operation performing the synchronization, 3) fix a memory management problem, and 4) fix a synchronous memory transfer. We automatically apply remedies for larger problems on the binary level. We focus our automatic application efforts on larger problems due to their increased difficulty in correcting manually. Fixing the issue on the binary level allows for problematic operations to be corrected even if they take place in a closed source component. In real world applications, we have found that autocorrection can reduce application execution time by up to 43%.

In Section 2, we describe the techniques used by existing tools and how they are insufficent to identify larger problems. We introduce the overall structure of FFM and its existing functionality in Section 3 and in Section 4 we introduce the extended FFM model. In Section 4.1, we detail the techniques used to identify largers constructs and how remedies are developed. In Section 5, we discuss our techniques for automatically applying remedies to application binaries. In Sections 4.2 and 5.4 we describe our experiments with these techniques on real world applications. In Section 6 we describe other general improvements made to FFM.

2 RELATED WORK

The techniques employed by FFM are inspired by previous research on performance tools, profile guided optimizers, and autotuners. FFM leverages contributions in these areas to create a new profiling and analysis structure that reduces (or eliminates) the limitations faced by existing techniques. We describe the techniques commonly used in each area to identify (and correct) performance issues, the limitations on the types of performance problems identifiable by each technique, and how FFM's approach differs.

2.1 Performance Measurement Tools

Profiling and tracing based performance tools [1, 6, 10, 13, 18, 21–23, 25, 27, 32] describe resource consumption at points in the program. The assumption is that points in the program with the highest resource consumption correlate to problems that, if fixed, would result in the largest performance benefit. While resource consumption can help a user identify problems, there are limits on the help it can provide.

Early work on critical path analysis showed [16] that resource consumption is not always a good predictor of the obtainable benefit. When a point of high resource consumption is identified, the user must perform a detailed manual analysis of the operation to determine if it is problematic and what remedy to employ. FFM's approach differs by identifying the problem and providing the user with an estimate of the performance benefit of fixing a problem. The differences between FFM and previous performance tools are described in our original paper [37].

Resource consumption reporting at single points in a program can hide larger problems where the problem spreads resource consumption across many points. These larger problems will appear as a collection of smaller single point issues even though a single remedy exists that would result in a reduction in resource consumption of all of the smaller points. The result is that the user expends effort trying to identify and fix a problem that may have limited performance benefit while missing those that would have a larger benefit. FFM addresses this problem by grouping together points where a single remedy can be applied that will fix problems at multiple points and automatically applying a fix to common synchronization problems.

2.2 Profile Guided Optimization

Profile guided optimization (PGO) is a compiler code optimization technique that uses performance data collected from a run of a program to optimize the application. PGO's insert instrumentation into the application (typically during compilation), run the instrumented program with a representative input data set, and use the collected profile data from the representative run to guide a recompilation of the program to eliminate specific performance problems. PGO techniques targeted problems such as reducing cache miss rates [8, 24, 28, 29], loop restructuring [9], and identifying functions where inlining would improve performance [4, 7, 9]. 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 [17] and I/O performance [35].

PGO techniques rely on compiler assistance to both insert instrumentation into the application and to correct any issues identified. The tying of an optimization technique to a compiler limits it only to applications that have source code available. New hardware counter sampling based techniques [8, 28] have been developed that limit the compiler assistance required to collect profiling data. However, sample based PGO data collection approaches still rely on compiler assistance to correct any issues that are identified with the data they collect. FFM operates purely on the binary without requiring using a specific compiler or access to the source code used to compile the binary.

PGOs focus on identifying and correcting 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 operations, such as CPU/GPU synchronizations that cross library boundaries, requires understanding of non-synchronization GPU behavior to identify that a problem is present and the correct transformation to apply.

2.3 Autotuning

Autotuning is the process of identifying optimal 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 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 identifying parameters such as the best program inputs [5, 38, 39], compiler settings [2, 34], and algorithm configurations [3, 11] 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 can be 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.

3 THE FEED FORWARD PERFORMANCE MODEL

The Feed Forward Measurement Model (FFM) was originally detailed in the work on Diogenes [37]. FFM was created to address the actionability of feedback provided by performance tools. The focus of FFM was to identify the presence of problems and giving the user feedback in the form of an estimate of the potential performance benefit if the problem were fixed. FFM employes a multi-stage/multi-run instrumentation approach to performance analysis where data is collected over multiple executions of the program. With each execution, the instrumentation and data collected is adjusted based on application behavior. This approach allows the applications behavior to guide FFM to potentially problematic CPU/GPU interactions. When a potential problem is identified, FFM uses finer-grained instrumentation to identify the type of problem present. Spreading the collection of fine grained detail over multiple runs allows for the use of high overhead instrumentation without hiding problems sensitive to overheard.

Diogenes uses FFM to identify problematic synchronization and memory transfer events between the CPU and GPU. A synchronization was deemed problematic if the data it was protecting was never accessed or if the synchronization could be moved to a different location to improve performance. FFM records the locations of data being transferred to and from the GPU. After a synchronization takes place, it uses memory tracing to identify the instructions that access these locations. A memory transfer was deemed problematic if it contained data that had already been transferred. FFM uses content-based data deduplication to identify when a transfer contains data that has already been sent to or from the GPU.

FFM consists of five stages. The five stages are broken down into four data collection stages and an analysis stage to identify problematic operations. Figure 1 shows a visual representation of the stages of FFM including the modifications made to support these new enhancements. Data is collected using binary instrumentation of CPU code to collect performance data on synchronizations and memory transfer events, allowing the capture of events that are missed by vendor-supplied performance data collection frameworks and library interposition methods. The use of binary instrumentation allows FFM to maintain compatibility with applications written in a wide range of programming languages and parallelization frameworks. The five stages of FFM are:

Stage 1 - *Baseline Measurement*: Identifies the functions performing GPU synchronizations and measures application execution time. We identify functions performing GPU synchronizations by instrumenting the internal driver function that performs the synchronization. This internal driver function is called by public API calls that need to perform a synchronization such as `cuMemcpy`. Instrumenting this function allows FFM to directly capture when a synchronization takes place without the reliance on vendor supplied tools, resulting in a more accurate picture of the synchronizations taking place in the program. This list is the starting point of the FFM model and dictates where more detailed information will be collected in stages 2 and 3.

Stage 2 - *Detailed Tracing*: Trace function calls performing synchronization and memory transfers. For each operation, we record the amount of time spent in the operation along with a stacktrace. We trace the function calls performing synchronizations discovered in stage 1 and a predefined set of memory transfer operations. The trace data is used by stage 5.

Stage 3 - *Memory Tracing and Data Hashing*: Collects the data needed to determine if an operation is problematic. Two different data collection approaches are employed based on the type of operation. For synchronization operations, a memory tracing approach is used to identify locations where data protected by the synchronization is accessed. We store the location of the first CPU instruction accessing data that was computed by the GPU along with a stack trace of the synchronization operation performed before the access. For memory transfers, we use a content-based data deduplication approach to identify duplicate transfers.

Stage 4 - *Sync-Use Analysis*: Collect timing information to determine if the synchronization was misplaced. The time between the instruction that first accesses data computed by the GPU and the synchronization is collected. The timing information is used in stage 5 to calculated the expected benefit for misplaced synchronizations.

Stage 5 - *Modeling*: Uses the information collected in stages 1 through 4 to determine if an operation is problematic and what the potential benefit might be from correcting the operation. For synchronization operations, a simple data flow analysis is used to determine the necessity of a synchronization operation. For data transfers, content-based data deduplication is used to detect problematic transfers. We use a new performance model to determine what the effect on application execution would be if these problems

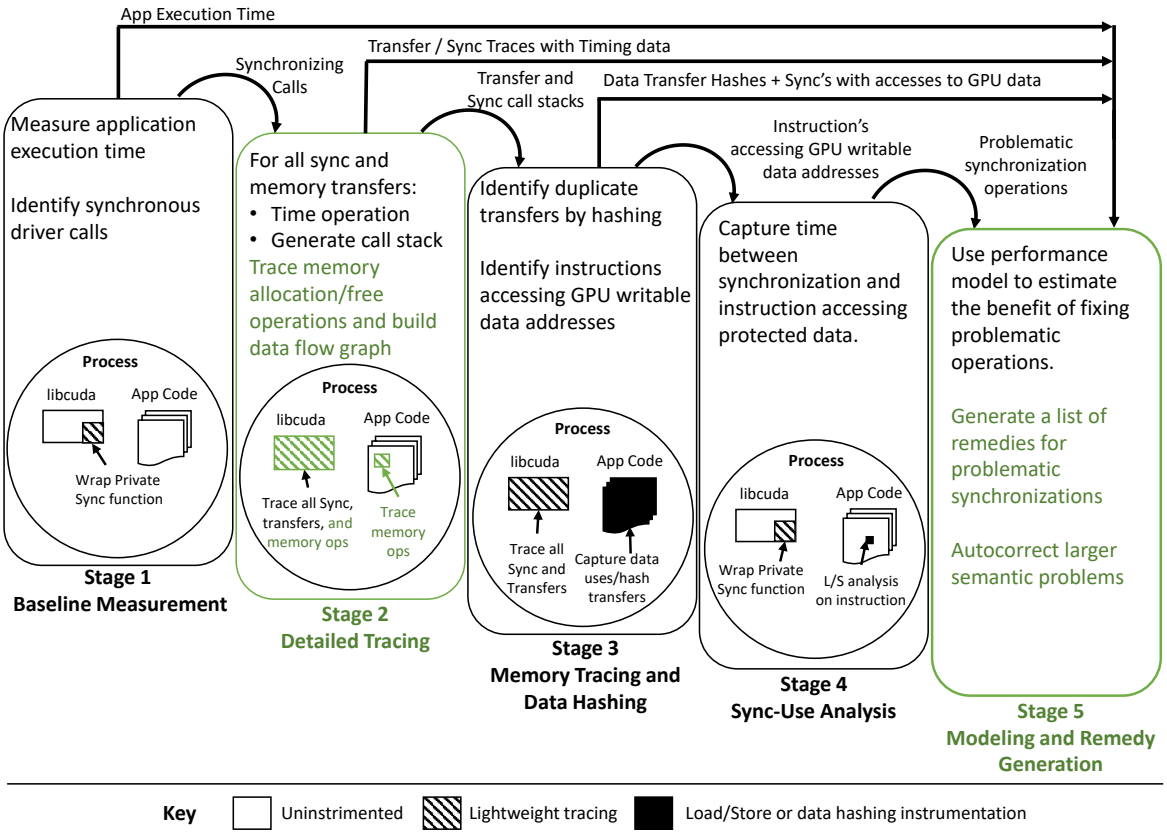

Figure 1: Overview of the stages of the FFM model with Extended FFM components listed in green.

were corrected. Generating an estimate of potential benefit can determine on which problems the user should focus on.

4 THE EXTENDED FEED FORWARD PERFORMANCE MODEL

While FFM automatically identified the presence of a problem, manual analysis was still required to determine how to remedy the issue. Identifying the remedy may not be obvious and may require an expert understanding of GPU programming to identify what was wrong. A problematic operation may also be larger construct that requires understanding of non-synchronization GPU behavior to identify and resolve. Fixing problems exhibited by larger constructs can result in greater performance benefits.

Once the remedy was identified, the user had to manually apply the remedy to correct the problematic behavior. Implementing a remedy can be time consuming if it requires major modifications to the program. In some cases, if the problem appears in a binary for which source code is not available, remedying the issue would require manual modification of a binary object. While FFM can deliver an estimate of benefit for remedying issues that it identifies, a developer is much less likely to correct issues if the time needed to remedy the issue is high and they are given only an estimate.

We introduce the extended FFM model with enhanced capabilities to identify larger constructs that caused a problematic synchronization to occur, identify the remedy to problematic operations, and automatically correct a subset of problems. We extend stages 2 and 5 of FFM to support these new capabilities. The extensions to the existing FFM stages are:

Stage 2 - *Detailed Tracing*: We enhanced the detailed tracing stage to include the collection of memory allocation and free operations so that a dataflow graph could be constructed. The dataflow graph is used by stage 5 to help detect the presence of larger problems and identify the remedy for a problematic operation. We discuss the construction of the dataflow graph and how it is used to identify larger problems and to identify remedies in Section 4.1.

Stage 5 - *Modeling and Remedy Generation*: We enhanced the modeling stage to generate a list of remedies for the problems discovered by the model. We use the memory graph constructed in stage 2 along with the list of problematic operations identified by the model to determine what remedy to apply. We discuss our technique to identifying the remedy in Section 4.1. When a remedy is identified that we can automatically apply to the application, we automatically correct the problem in the application using a generic solution. We describe the automatic correction process in Section 5.

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

Figure 2: An illustrative example of an unnecessary synchronization in cuFFT when used in compatibility mode with Qbox

4.1 Automatic Remedy Identification

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 section, we describe the challenges of identifying the correct remedy for each problem type and how we automate the identification of the cause of the problem.

4.1.1 Memory Transfer Issues. The unnecessary use of synchronous memory transfer operations is a problem that we previously found to be common in large applications [36, 37], such as within cuFFT [26] when used with Qbox [14], cuIBM [20], and cumf_als [33]. One such instance, originally presented in an exploration of GPU performance problems [36], can be seen in Figure 2 with the synchronizations performed in Nvidia's cuFFT [26] 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 synchronizations can be unnecessary depending on how the application uses `fftw_execute_dft`. The molecular dynamics application Qbox [14], when linked against cuFFT in compatibility mode, has instances where both implicit synchronizations in `fftw_execute_dft` are unnecessary.

The obvious remedy for an unnecessary synchronization performed at a transfer is to convert these call sites to their asynchronous form, such as converting `cuMemcpyDtoH` to `cuMemcpyDtoHAsync`. However, performing only this conversion may result in the synchronous behavior remaining. If the CPU memory used in the transfer was pinned, such as by allocating the CPU memory with `cuMemAllocHost`, the transfer speed itself could be reduced by as much as 50%. 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 call to use pinned pages. 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 3. 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 operations 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 are 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.

We modified stage 2 of FFM 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 [30] 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 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 allocated by `malloc`. If the address was allocated via `malloc`, we record the location of 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 when pinned memory is already being used for a transfer.

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 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 [31] 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

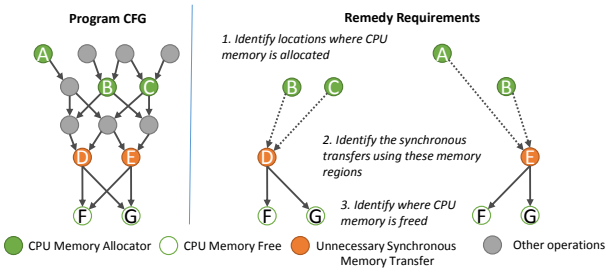

Figure 3: Information required to remedy a problematic synchronization caused by a memory transfer

```
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 4: Example Diogenes remedy output for unnecessary synchronizations caused by memory transfers in Qbox

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 4. Figure 4 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.

4.1.2 Memory Management Issues. Frequent unnecessary synchronizations caused by `cudaFree` operations is one of the most common mistakes detected in GPU programs [36, 37]. 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 5 is an excerpt from the application cuIBM [20] 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 [12] 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.

```
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 5: Example of an unnecessary synchronization as a result of memory management issue in cuIBM

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.

Providing actionable feedback requires that we identify the `cudaFree` operations with unnecessary synchronizations and identify the corresponding `cudaMalloc` operations. The first step is to link `cudaMalloc` operations with the `cudaFree` operations that freed the memory. Similar to identifying remedies in memory transfer issues, we modified stage 2 of FFM to construct a simple dynamic data flow graph. We use GOTCHA to interpose on `cudaMalloc` and `cudaFree` operations, and compare the memory locations allocated by `cudaMalloc` operations to the memory freed by `cudaFree`. If a match is found, we record the location of this pair of operations. Stage 5 was modified to generate a remedy that reports these pairs to the developer if the synchronization at the `cudaFree` operation was unnecessary. We use the existing data that Diogenes collects to determine if the synchronization is unnecessary at the `cudaFree` callsite. If it is, the pair is reported to the developer. Figure 6 shows an excerpt from the output of Diogenes for the program cumf_als [33]. The output contains the stack with line numbers at which `cudaFree` was called at, 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` callsite.

4.1.3 Unnecessary and Misplaced Explicit Synchronizations. Detecting unnecessary explicit synchronization operations, such as `cuCtxSynchronize`, was part of the original FFM design. While FFM could 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).

4.2 Experiments: Remedy Identification

We tested the effectiveness of the remedy identification extension to FFM on three real world applications: cumf_als [33] (git revision a5d918a), a GPU-based large matrix factorization library that uses the alternating least square (ALS) method developed at IBM and University of Illinois Urbana-Champaign; cuIBM [20] (git revision 0b63f86), a 2D Navier-Stokes solver using the immersed boundary method developed at Boston University; and Qbox [14], (version r140b) a molecular dynamics application developed at U.C. Davis that uses nvidia's cufft [26] (version 8.0) in compatibility mode.

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. Each application was compiled with the GNU Compiler Collection

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

Figure 6: Example Diogenes remedy output for unnecessary synchronizations caused by memory mangement issues in cumf_als

(GCC) version 4.9 and linked against CUDA version 9.2 (cuIBM and cumf_als). Qbox differed in that the FFTW compatable library used was cuFFT from CUDA version 8.0. The experiments were conducted on a single node using a single GPU with system GPU driver version of 418.87.

Table 7 summarizes the remedies prescribed by FFM 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 remedying these issues in Section 5.4.

Our experiments for cumf_als were run using the GroupLens MovieLens [15] 10M data set run with an iteration count of 5000. FFM 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.

The remedies identified for cuIBM and QBox addressed similar problematic behavior as those of cumf_als. For cuIBM, FFM identified remedies for 539 memory management, 31 memory transfer, and 168 explicit synchronization issues. The major difference in the problems identified in cuIBM to those identified in cumf_als

Application Name	Source Size (lines of code)	FFM Perscribed Remedies: Memory Mgmt Problems	Transfer Sync Problems	Explicit Sync Problems
cumf_als [33]	5K	22	3	10
cuIBM [20]	36K	539	31	168
QBox [14]	100K	0	79	1

Figure 7: Number of remedies perscribed for synchronization problems identified by FFM

was the transfer remedies identified were primarily targeted 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 (*lidDrivenCavity/Re5000*).

In QBox, FFM 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 automatically generated by the tool contained in the QBox distribution.

5 AUTOCORRECTION 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 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.

FFM was originally created to provide an estimate of potential benefit to a developer to lessen the risk. While FFM was able to provide accurate estimates of benefit of fixing problematic operations, significant risk for the developer still remained. The developer needed 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.

The autocorrection technique was created to lessen the risk to developers by identifying a class of fixable problematic operations in their 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 of memory management and memory transfers operations. 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 portion of overhead from excessive memory allocation and free opertions while also allowing us to selectively enable/disable synchronization behavior when it is required. Without this transformation, we would have no ability to control synchronization behavior due to non-existance of asynchronous GPU memory free routines.

The memory transfer transformations that we apply convert the call to its asynchronous form 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: 1) a setup phase at application startup to identify and apply the transformations to remedy the problem and 2) the execution phase where the transformation dynamically decides what remedy to employ based on the operation invoked by the application. In this section, we describe the model of application execution created to identify problematic operations and how these phases work together to remedy these common synchronization problems.

5.1 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 in our original implementation of FFM. This model is used by the setup and execution phases to apply corrections to the application. 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. The set of operations $N = \{C, G\}$, where C is the set of CPU operations in the graph and G is the set of GPU operations in the graph. An edge describes the Lamport happens-before [19] ordering between operations. On the same processor, an outbound edge from a node n_x to a node n_y denotes the operation peformed 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.

Each node in N represents an operation that takes place in the application and has attributes ($NType$, $Problem$, $CallStack$, $MemPtr$, $Size$) associated with it. $NType$ denotes the type of operation performed by the node: a CPU memory operation ($CPUMalloc$ or $CPUFree$), a GPU memory operation ($GPUMalloc$ or $GPUFree$), a pinned page memory operation ($PinnedMalloc$ or $PinnedFree$),

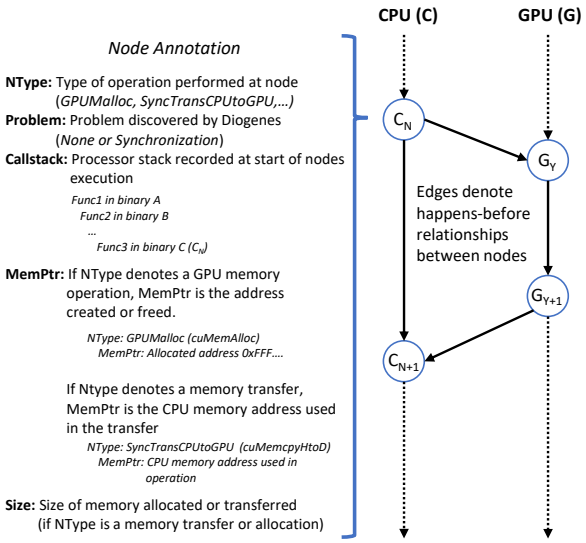

Figure 8: Illustrative example of the annotated graph used to model application execution during autocorrection

or a synchronous transfer operation ($SyncTransCPUtoGPU$ and $SyncTransGPUToCPU$). The $Problem$ attribute denotes the problematic operation identified in stages 3 and 4 of the FFM model ($None$ or $Sync$). 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. Figure 8 shows a visual representation of the graph constructed and the annotation applied to each node.

5.2 Setup Phase of Autocorrection

The autocorrection process begins by using the model of application execution created by the original implementation 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 instrumentation into the application around the functions required to support autocorrection.

Figure 9 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 `cudaMemcpy` and `cudaMemcpyAsync`) and GPU memory operations (such as `cudaMalloc` and `cudaFree`). 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 `cudaHostAlloc` and `cudaHostFree`). 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. `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 synchronizations, 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 synchronizations are remedied. On line 20, we write the call stacks out to a file to be read by the execution phase.

```
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
16           !IsTransFromStack(Node)):
17         uSync = uSync U Node.CallStack
18       else:
19         rSync = rSync U Node.CallStack
20   uSync = uSync - rSync
21   WriteToFile(uSync)
```

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
Functions *TransIntercept*, *GPUMemIntercept*, *PinnedWrap*, and *PostSynchronization* are defined in Figure 10

Figure 9: Setup phase used to identify unnecessary synchronizations and insert function wrappers to support Autocorrection

5.3 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 10 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 32. Similarly, `TransIntercept` on line 14 intercepts and modifies memory transfer operations to apply remedies. The wrapper function `PinnedWrap` on line 7 supports the interceptor function `TransIntercept` by providing data on pinned memory allocation operations. The function exit wrapper `PostSynchronization` on line 27 notifies the execution phase of a synchronization and performs post synchronization tasks.

`GPUMemIntercept` (line 32) intercepts memory allocation and free operations, redirecting these operations to use a memory pool. By using a memory pool, we limit the number of calls to `cudaFree` 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 34). `cudaMalloc` 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 36). Since we may not call `cudaFree`, 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 37). If there is no match, we perform an explicit synchronization.

`TransIntercept` (line 14) intercepts and modifies memory transfer operations to remove unnecessary synchronizations. We convert the synchronous memory copy operation to its asynchronous form, applying a synchronization only when it is required. Converting the call to 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 15). The set of allocated pinned pages (`pinnedSet`) is captured by the wrapper `PinnedWrap` on line 7. `PinnedWrap` inserts allocated memory ranges into a set (line 10) and removes those that are freed (line 12).

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 page and the transfer request is modified to use the pinned page (lines 17-19). 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 21). The function `PostSynchronization` is called when a synchronization completes and the copy is performed (line 27-30). 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 23, we initiate the modified asynchronous transfer. On line 24-25, 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.

5.4 Experiments: 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 (Section 4.2). Table 1 summarizes the benefit obtained using autocorrection. 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 FFM. However, a reduction in execution time of 85% [36] was achieved 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

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

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

Figure 10: Execution Phase of Autocorrection

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

different abstraction and manually managing the GPU memory and synchronization.

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 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 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 synchronizations 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 synchronizations caused by `cudaFree` operations. Remedies were applied to 539 problematic synchronizations occuring at `cudaFree` operations. The application of the remedies resulted in the interception of 45 million calls to `cudaMalloc` and 45 million calls to `cudaFree`, removing 45 million synchronizations. Remedies applied to the 31 memory transfer problems resulted in 32 synchronizations being removed. Total benefit obtained was a reduction in execution time by 43.3%. Qbox shows an extreme example of unnecessary synchronizations caused by memory transfer operations. Remedies were applied to 79 problematic synchronizations 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%.

	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 2: Synchronization operations removed using autocorrection

The overhead of running FFM 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.

6 IMPLEMENTATION IMPROVEMENTS TO FFM

Unlike most other performance tools, FFM directly instruments the user space GPU driver to trace synchronizations. Direct instrumentation gives us the ability to detect synchronizations that are missed by other performance tools reliant on vendor supplied tracing methods. A major drawback is that this technique requires the identification of an unmarked internal GPU driver function responsible for performing the synchronization. The identification of this function was a manual process and needed to be performed on every update to the device driver. The requirement for manual identification limited the applicability of FFM to only a few drivers.

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 (such as `cuCtxSynchronize` and `cuMemcpy`). 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 in, 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 2 to 3 functions, that do not return when a synchronization is performed. We select the deepest function on the stack as the synchronization function. We are currently working on extracting this funcitonality out of FFM so that other tools can benefit from more accurate synchronization information.

7 CONCLUSION

We have presented the extended feed-forward measurement model that automates the detection of synchronizations problems, identifies if the synchronization problem is a component of a larger construct, identifies remedies that can correct the issue, and automatically applies remedies to problems exhibited by larger constructs. We developed a prototype implementation of the extended FFM model, employing it on three real world applications. Using this implementation, we were able to automatically identify remedies for several hundred synchronization issues across the three

applications. The autocorrection technique was able to correct the problems exhibited by larger constructs identified in FFM to reduce application execution time between 9% and 43%. While the implementation focused on the identification and correction of synchronization issues on applications running on Nvidia GPUs, the general technique of the extended FFM model should be easily modified to apply to other GPU accelerators as they become more prevalent in the high performance computing space.

8 ACKNOWLEDGEMENTS

This work is supported in part by Department of Energy grant DE-AC05-00OR22725 under Oak Ridge National Lab contracts 4000151982 and 4000164398; National Science Foundation Cyber Infrastructure grant ACI-1449918; Lawrence Livermore National Lab grant B617863; and a grant from Cray Inc. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

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

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 303–315. https://doi.org/10.1145/2628071.2628092

[3] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 85–96. https://doi.org/10.1109/CGO.2011.5764677

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

[5] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir. 2013. Taming parallel I/O complexity with auto-tuning. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503278

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

[7] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience* 22, 5 (1992), 349–369. https://doi.org/10.1002/spe.4380220502 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380220502

[8] D. Chen, T. Moseley, and D. X. Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 12–23.

[9] Robert Cohn and P. Geoffrey Lowney. 1999. Feedback directed optimization in CompaqâĂŹs compilation tools for Alpha. In *In 2nd ACM Workshop on Feedback-Directed Optimization (FDO*.

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

[11] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. *SIGPLAN Not.* 50, 6 (June 2015), 379–390. https://doi.org/10.1145/2813885.2737969

[12] Richard Galvez and Greg van Anders. 2011. Accelerating the solution of families of shifted linear systems with CUDA. (2011). arXiv:hep-lat/1102.2143

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

[14] F. Gygi. 2008. Architecture of Qbox: A Scalable First-principles Molecular Dynamics Code. *IBM Journal of Research and Development* 52, 1 (January 2008), 8. http://dl.acm.org/citation.cfm?id=1375990.1376003

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

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

[17] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO.2013.6494997

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

[19] Leslie Lamport. 1978. Time, Clocks and the Ordering of Events in a Distributed System. *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), 558–565.

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

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

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

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

[24] S. McFarling. 1989. Program Optimization for Instruction Caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*. ACM, New York, NY, USA, 183–191. https://doi.org/10.1145/70082.68200

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

[26] Nvidia. 2018. *The Cuda FFT Library* (9.2 ed.).

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

[28] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 2–14. http://dl.acm.org/citation.cfm?id=3314872.3314876

[29] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 16–27. https://doi.org/10.1145/93542.93550

[30] David Poliakoff and Matt LeGendre. 2019. Gotcha: An Function-Wrapping Interface for HPC Tools. In *Programming and Performance Visualization Tools*, Abhinav Bhatele, David Boehme, Joshua A. Levine, Allen D. Malony, and Martin Schulz (Eds.). 185–197.

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

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

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

[34] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161054

[35] Yijian Wang and David Kaeli. 2003. Profile-guided I/O Partitioning. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 252–260. https://doi.org/10.1145/782814.782850

[36] B. Welton and B. P. Miller. 2018. Exposing Hidden Performance Opportunities in High Performance GPU Applications. In *18th IEEE/ACM International Symposium*

on Cluster, Cloud and Grid Computing (CCGRID '18). Washington, D.C., 301–310. https://doi.org/10.1109/CCGRID.2018.00045

[37] B. Welton and B. P. Miller. 2019. Diogenes: Looking For An Honest CPU/GPU Performance Measurement Tool. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19)*. Denver, CO, 301–310. https://doi.org/10.1109/CCGRID.2018.00045

[38] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1145/1362622.1362674

[39] Yixun Liu, E. Z. Zhang, and X. Shen. 2009. A cross-input adaptive framework for GPU program optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–10. https://doi.org/10.1109/IPDPS.2009.5160988