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%.

CSCS CONCEPTS
• Software and its engineering → Software performance;

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

1 INTRODUCTION
The inclusion of GPUs on today’s leadership-class computing platforms has increased the types of performance problems that can affect 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 \texttt{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 synchronization 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 insufficient 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 larger 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 program 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.
Autotuning techniques have been applied to identifying parameters. 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.

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 employs 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 overhead.

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 cudaMemcpy. 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 stacktrace 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
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, remediying the issue would require manual modification of a binary object. While FFM can deliver an estimate of benefit for remediying 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.

Figure 1: Overview of the stages of the FFM model with Extended FFM components listed in green.
When a problem is discovered, identifying the correct remedy to address the problem but also describing what needs to change to remedy the problem 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 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]. Memory transfer operations, such as `cuMemcpyDtoH` and `cuMemcpyHtoD`, 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 2: An illustrative example of an unnecessary synchronization in cuFFT when used in compatibility mode with Qbox](image)

![Figure 3: Information required to remedy a problematic synchronization caused by a memory transfer](image)
cuMemcpyHtoD_v2 called at...
Offset 131568 in libcufftww.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
inlibcufftww), 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 syn-
chronizations caused by cudaMemcpy operations is one of the most
common mistakes detected in GPU programs [36, 37]. Two fac-
tors play a key role in the overuse of cudaMemcpy operations: the
application structure can hide where these operations take place
and fixing problematic behavior requires modifying other oper-
ations in the program. Figure 5 is an excerpt from the applica-
tion cuIBM [20] that shows how identifying the location of cud-
afree operations can be difficult in modern programs. The function
cusp::system::detail::generic::multiply<int, double> 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 indirectation and template function calls
that a cudaMalloc operation is performed. Only when the temporary_array
object is being destroyed, and after calling another several layers of
destructors, is the cudaMemcpy operation performed. An extra level of
complexity is added when the compiler optimizes this code by re-
moving some of these layers of indirectation, increasing the difficulty
in locating the problem.

Unlike other unnecessary synchronization operations that FFM
detects, cudaMemcpy is unique in that there is no asynchronous ver-
sion of the operation available. Fixing a synchronization issue at
cuMemcpy is limited to removing or moving the operation. However,
fixing a cudaMemcpy operation cannot be done without also address-
ing 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 cud-
afree operations with unnecessary synchronizations and identify
the corresponding cudaMalloc operations. The first step is to link
cudaMalloc operations with the cudaMemcpy 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 cudaMemcpy operation
was unnecessary. We use the existing data that Diogenes collects to
determine if the synchronization is unnecessary at the cudaMemcpy
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, the type of problem identified (unnec-
necessary 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. De-
tecting 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 1081 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 management 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 compatible 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 occurred at a unique execution stack during program execution. If the same execution stack is responsible for multiple occurrences 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 benefit obtained if these transformations were made permanent.

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 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 (lidDrivenCavityRe5000).

In QBox, FFM identified remedies for 79 memory transfer and 1 explicit synchronization issue. The memory transfer remedies targeted implicitly synchronous cudaMemcpyHtoD and cudaMemcpyDtoH function calls that occurred 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.

<table>
<thead>
<tr>
<th>Application</th>
<th>Source Size</th>
<th>FFM Perscribed Remedies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(lines of code)</td>
<td>Memory</td>
</tr>
<tr>
<td>cumf_als [33]</td>
<td>5K</td>
<td>22</td>
</tr>
<tr>
<td>cuIBM [20]</td>
<td>10K</td>
<td>209</td>
</tr>
<tr>
<td>QBox [14]</td>
<td>100K</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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 rerunning 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 operations 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-existence 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 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.

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 \( GPUMean \)), a pinned page memory operation (\( PinnedMalloc \) or \( PinnedFree \)),

![Figure 8: Illustrative example of the annotated graph used to model application execution during autocorrection](image)

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

or a synchronous transfer operation (\( SyncTransCPUtoGPU \) and \( SyncTransGPUCPU \)). 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 \( cudaMemcpyAsync \)). The instrumentation modification 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 \( cudaMemcpyAsync \) and \( cudaMemcpyAsync \)). The wrapping instrumentation captures the parameters and return values of the wrapped call but does not alter the behavior of the
writes to stack addresses during autocorrection. If the operation is problematic, the execution phase identifies and corrects problematic operations. TransIntercept, PinnedWrap, GPUMemIntercept, and PostSync-chronization 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.

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 PostSync-chronization 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.
We tested the effectiveness of the autocorrection on the applications cuBM, 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 cuBM, 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 cuBM 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

5.4 Experiments: Autocorrection

We tested the effectiveness of the autocorrection on the applications cuBM, 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 cuBM, 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 cuBM 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
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_al 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 cudaMemcpy and 85K calls to cudaMemcpy. These operations were primarily the cudaMemcpy/cudaCopy 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 cudaMemcpy 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%.

cubM shows an extreme example of unnecessary synchronizations caused by cudaMemcpy operations. Remedies were applied to 339 problematic synchronizations occurring at cudaMemcpy operations. The application of the remedies resulted in the interception of 45 million calls to cudaMemcpy and 45 million calls to cudaMemcpy, 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 took place at various memory transfer operations (such as cudaMemcpyHtoD). The result was the elimination of 32 million synchronization operations, reducing execution time by 9.9%.

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_al) 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 cudaMemcpy). 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 functionality 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


