MUPPET: Optimizing Performance in OpenMP via Mutation Testing

Dolores Miao[1], Ignacio Laguna[2], Giorgis Georgakoudis[2],
Konstantinos Parasyris[2], Cindy Rubio-González[1]
[1]University of California, Davis
Davis, CA, USA
{wjmiao,crubio}@ucdavis.edu
[2]Lawrence Livermore National Laboratory
Livermore, CA, USA
{ilaguna,georgakoudis1,parasyris1}@llnl.gov

Abstract

Performance optimization continues to be a challenge in modern HPC software. Existing performance optimization techniques, including profiling-based and auto-tuning techniques, fail to indicate program modifications at the source level thus preventing their portability across compilers. This paper describes Muppet, a new approach that identifies program modifications called *mutations* aimed at improving program performance. Muppet's mutations help developers reason about performance defects and missed opportunities to improve performance at the source code level. In contrast to compiler techniques that optimize code at intermediate representations (IR), Muppet uses the idea of source-level *mutation testing* to relax correctness constraints and automatically discover optimization opportunities that otherwise are not feasible using the IR. We demonstrate the Muppet's concept in the OpenMP programming model. Muppet generates a list of OpenMP mutations that alter the program parallelism in various ways, and is capable of running a variety of optimization algorithms such as Bayesian Optimization and delta debugging to find a subset of mutations which, when applied to the original program, cause the most speedup while maintaining program correctness. When Muppet is evaluated against a diverse set of benchmark programs and proxy applications, it is capable of finding sets of mutations in 70% of the evaluated programs that induce speedup.

CCS Concepts: • **Software and its engineering → Software performance**; • **Computing methodologies → Parallel programming languages**.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PMAM '24, March 3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0599-1/24/03.
https://doi.org/10.1145/3649169.3649246

Keywords: mutation testing, OpenMP, performance optimization, Bayesian optimization, delta-debugging algorithm, dynamic program analysis

ACM Reference Format:
Dolores Miao[1], Ignacio Laguna[2], Giorgis Georgakoudis[2],, Konstantinos Parasyris[2], Cindy Rubio-González[1]. 2024. MUPPET: Optimizing Performance in OpenMP via Mutation Testing. In *The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24), March 3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3649169.3649246

1 Introduction

Performance optimization continues to be a challenge in modern HPC software. The adoption of multi-core heterogeneous systems and the use of multi-process and multi-threaded programming models to fully utilize modern architectures are some of the factors that limit the ability of developers to solve performance issues; these issues can result in poor user experience, lower system throughput, limit scalability, and a waste of computational resources [5, 7, 53].

Problems with Existing Techniques. A large amount of work has been proposed to identify performance issues and a number of tools are used in the current HPC production environment to analyze applications' performance [3, 21, 32, 45]. However, the process of isolating performance problems and/or generating tests to identify them is still mostly a manual process.

Most performance optimization techniques focus on highlighting "hot spots" but ultimately rely on programmers to identify code modifications that fix a performance problem or improve overall performance. Other approaches are based on the concept of quantifying hardware or runtime system events [17, 35, 36], but do not explicitly inform the programmer how to modify the code to improve performance. Compiler optimizations improve performance usually at the intermediate representation (IR) level; however, reasoning about correctness at the IR level is much more difficult than at the source level. As a result, compiler optimizations can leave

optimization opportunities on the table. Moreover, IR-level optimizations are not portable across compilers.

We could potentially solve performance problems given accurate performance models for each available platform and application. If performance models are available, we could simply check if the application's behavior falls into the bounds of such models. However, such an ideal mechanism is hard to realize in practice as performance models are notoriously difficult to build accurately, given the complexity of the HPC software stack and underlying hardware. There are solutions to build performance models for specific aspects of the hardware and applications [12, 30, 51], but these models are usually not composable and as a result of little practical use in modeling an entire application and platform.

Our Contributions. We present an approach based on *mutation testing* [25] to identify source code changes, or *mutations*, that (1) improve performance, and (2) help developers reason about performance at the source-code level (in contrast to IR- or assembly-level like in existing methods). Since such an approach is based on source modifications, it is portable across compilers.

Mutation testing has been proposed to identify correctness faults [25], and assumes that a syntactic change (a mutant) along with an exploration campaign of multiple mutants can help discover programs' defects faster than traditional methods. While some previous work has applied mutation testing to solve performance defects [10], mutation testing for performance has not been applied on parallel code and/or HPC programs. We demonstrate our approach in the OpenMP programming model, which is widely used in HPC.

We implement our approach in the framework named Muppet (Mutation-Utilized Parallel Performance Enhancement Tester). First, Muppet generates a list of OpenMP mutations that could alter the program performance in various ways. A mutation is defined as a change in an existing OpenMP directive in the program that could change the performance of the code block that the directive targets. Muppet considers only mutations that are not likely to change the correctness of the code block. Next, Muppet considers different optimization algorithms, such as Bayesian optimization (BO) [34] and delta debugging [56], to find a subset of mutations that, when applied to the original program, cause the highest speedup. We implement Muppet in the clang/LLVM front-end and evaluate it in the NAS Parallel Benchmarks [31] and three proxy applications (LULESH [26], HPCG [14], CoMD [18]).

In summary, our contributions are:

- We present a source-level approach that uses mutation testing to optimize HPC code. Our approach considers four classes of source mutations and applies them in OpenMP directives. *To the best of our knowledge, we are the first to explore using mutation testing to optimize OpenMP code* (Section 3).
- We design and implement our idea in the Muppet framework via the clang/LLVM front-end. Our approach integrates Muppet with several optimization algorithms, such as BO and delta debugging. The output of Muppet is a set of source modifications, or mutations, that produce a maximum speedup among the explored mutations, without affecting correctness (Section 4).
- We evaluate Muppet on several benchmarks and proxy applications. We demonstrate that Muppet is capable of identifying mutations that improve performance in 70% of the evaluated programs, with the best speedup in average running time of 15.64% (Section 5).

2 Overview

In this section, we describe the philosophy of our approach, provide background information on mutation testing, and provide a simple mutation example in a matrix multiply kernel that improves performance.

2.1 Approach's Philosophy

Existing approaches to isolate performance issues are difficult to use in practice. A number of performance problems can be fixed by changes in the source code; however, existing methods do not directly point to developers' source modifications that fix such issues. Compilers optimize code at the IR level but such solutions are not portable across compilers and make it harder to reason about correctness than solutions based on source modifications.

We believe that tools and techniques for performance optimization should have the following features:

- **Fine granularity detection:** tools should pinpoint, with fine granularity, the location (code line) of performance issues or potential performance improvements.
- **Guided fixes:** the approach should help programmers understand and reason about performance defects—without a good understanding, it is hard to solve the problem or avoid it in the future.
- **Automatic recommendations:** the approach should automatically suggest code modifications that improve performance or fix a performance problem.

We designed Muppet using the above criteria to identify changes in OpenMP directives that improve performance.

2.2 Mutation Testing for Performance

2.2.1 Challenges.

The key idea of Muppet is to perform small changes in the code, called *mutations*, and use exploratory algorithms to search for cases where mutations improve performance or fix a performance problem. Mutation testing has been studied before to detect faulty programs by injecting small syntactical changes that expose correctness defects [25]. The idea of mutation testing is to generate sufficient data to expose real software defects in the code.

However, it is challenging to use traditional mutation testing in isolating performance defects because the syntactic changes could create faults, i.e., breaking the semantics of the program and producing incorrect programs.

2.2.2 Our Solution.

Inspired by the previous work on mutation testing, we propose a different approach: *to inject only mutations that are semantically correct and do not yield an incorrect program for the purpose of exposing performance defects or speedup opportunities.* Semantically correct mutants, or *equivalent mutants*, are considered problematic for traditional mutation testing because by definition, they cannot fail the test suite, so they should be avoided to increase the effectiveness of mutation testing. In contrast, our approach explores semantically correct mutations, or a weaker form of mutations that successfully pass correctness tests, to identify any mutations that increase performance, thus indicating performance defects.

2.3 Mutation Example

Here, we present a synthetic matrix-multiplication example, shown in Listing 1, that demonstrates MUPPET's capabilities—when we apply MUPPET, it can find a set of mutations that yields faster code execution.

Listing 1. Example code with a mutation found by MUPPET that improves performance.

```
1  #define ARRAY_SIZE (2048)
2  double A[ARRAY_SIZE][ARRAY_SIZE];
3  double B[ARRAY_SIZE][ARRAY_SIZE];
4  double C[ARRAY_SIZE][ARRAY_SIZE];
5
6  int main(void) {
7    // initialize array and timer setup omitted
8    float var = 2.3f;
9    #pragma omp parallel for shared(var)
10   // mutation adds an OpenMP directive
11   #pragma omp tile sizes(16,16,16)
12   for (int i = 0; i < ARRAY_SIZE; ++i)
13     for (int j = 0; j < ARRAY_SIZE; ++j)
14       for(int k = 0; k < ARRAY_SIZE; ++k) {
15         C[i][j] += var*A[i][k]*B[k][j];
16       }
17   // end processing omitted
18 }
```

Originally, the code has only the OpenMP `parallel for` directive to parallelize the loop. Then, MUPPET applies mutations to the existing OpenMP directives found in the code. Note that while MUPPET only considers semantically correct mutations (and are likely to produce a correct program), it relies on existing correctness checks of the program, as shown in Section 5.1.1 for the evaluated programs. When we run MUPPET on this example with delta debugging, after 20 tryouts, MUPPET reports a mutation that, when applied to the program, improves performance. With BO, it takes 66 tryouts to finish the optimization process; but the mutation was reported with 11 tryouts. The identified mutation is highlighted in the source code. In this simple example, the mutation is the addition of the OpenMP tile construct, which tiles one or more loops. In the end, MUPPET reports to the developer that adding this construct to the loop introduces a 18.84x speedup, from 7.116801 seconds to 0.377674 seconds.

3 Approach

3.1 Problem Statement

Given an OpenMP program P with running time T, MUPPET analyzes the program and generates a set of mutations, $M = \{m_1, m_2, ..., m_n\}$, which potentially could induce program speedup. We define the program running time for the original variant program as:

$$T = P(\emptyset)$$

We define the running time for a variant program as:

$$T' = P(M'), \text{where} \quad M' \subseteq M, \text{and} \quad accurate(P, M') = True.$$

We define the ideal minimum program running time as:

$$\begin{aligned} &T_{min} = P(M_{min}), \\ &\text{where} \quad M_{min} \subseteq M, \\ &\text{and} \quad accurate(P, M_{min}) = True, \\ &\text{and} \quad \forall M' \subseteq M, T' >= T_{min} \end{aligned}$$

The goal of MUPPET is find a subset of M, $M_{min'}$, with $T_{min'}$ as close to T_{min} as possible.

3.2 Tool Workflow.

The overall workflow of MUPPET is illustrated in Figure 1. The purposes of these modules are described below:

- *Mutation generator* analyses the program and finds a set of source code mutations, which can potentially be applied to change the OpenMP parallelism of the program.
- *Transformer* generates a program variant with a subset of mutations found in the *Mutation generator* module.
- *Tester* runs the mutated programs from *Transformer* and tests the performance speedup and correctness of the mutated variant.
- *Optimizer* applies a user-specified optimization algorithm to find the minimum of the function $T' = P(M')$.

Next, we delve into the details of these modules, following the order as they appear in Figure 1.

3.3 Mutation Generator

The Mutation Generator module traverses the abstract syntax tree (AST) of the program, looking for source code locations that potentially can be mutated so that program parallelism is changed. The time complexity of this step is $O(n)$ where n is the number of statement nodes on the AST.

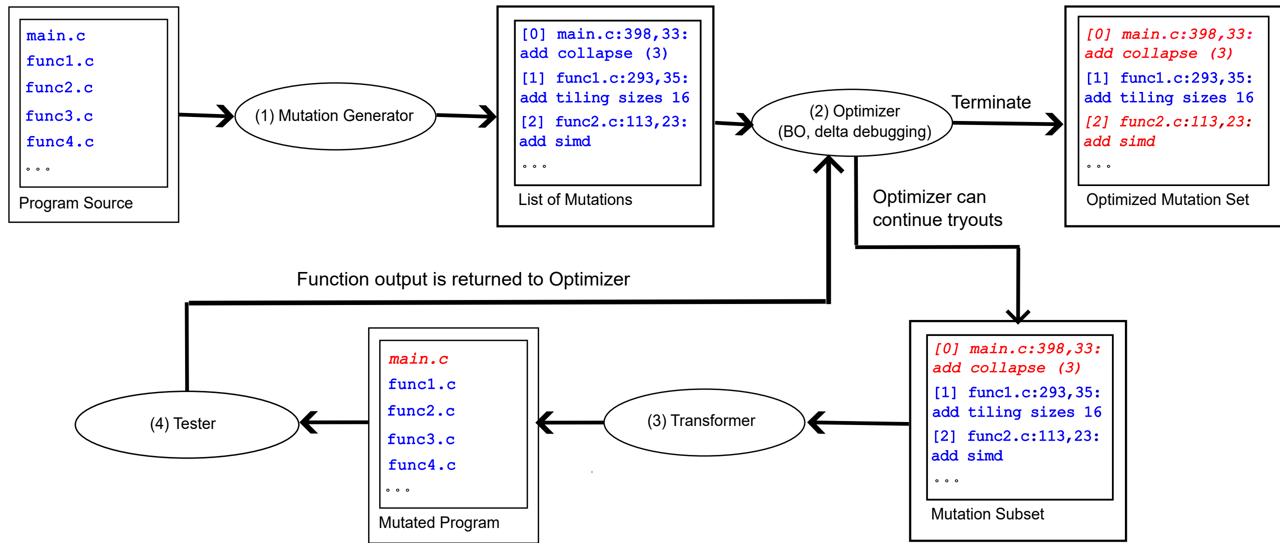

Figure 1. The workflow of Muppet. *Red texts in italic* indicates the mutation is applied, or the source file is changed.

The mutators in Muppet focus on mutating parallel/loop OpenMP constructs such as the `parallel` directive, `for` directive, or the `parallel for` directive. All of these directives specify a source code region to be executed in parallel, but the parallelism may not be high enough to utilize all available cores for the OpenMP program. It also looks for the beginning of `for` loops for SIMD mutations.

3.3.1 Mutation Classes. There are four types of mutations possible to apply to certain source code locations:

1. **Collapse Mutations** add a `collapse` clause to a multiple dimension `parallel for` loop. Collapse clauses may potentially improve parallelism by having more iterations, thus higher hardware thread usage, at the top level of the loop.
2. **SIMD Mutations** add a `simd` clause to an OpenMP parallelism-related directive such as a `parallel for` loop, or a `omp for` loop. SIMD clauses or directives hint at the compiler to check if there is a possibility to vectorize the loops and apply SIMD vectorization if possible.
3. **Tiling Mutations** add `tile` directives at the top of a multiple-dimension OpenMP loop. Tile directives split the loop space into smaller-sized "tiles", and each tile is ideally only accessed by one OpenMP thread. This design can potentially improve cache locality depending on how the data within memory is accessed within the loops, and thus may also introduce performance speedup. Due to the difficulty in determining loop size at compile time, Muppet only supports setting a fixed set of differently sized tiles as different mutations. For example, we can only set the tile size as a power of 8, 16, or 32. Given the limitations, users can still see from the optimization results whether using a smaller or larger-sized tile can have a higher speedup.
4. **Firstprivate Mutations** put read-only shared variables into a `firstprivate` clause for an OpenMP parallel region, in order to reduce dependency between parallel threads.

Once such language constructs (`parallel for`, `for`, etc.) are detected, the Mutation Generator module will then check the associated source code around the current language construct. If the source code around it satisfies certain statically defined criteria (see below), then unique information regarding the current mutation, such as source location, the way source code is modified (insert before, insert after, modify), and the mutation type, is added to the list of mutations. The algorithm for this process is shown in Algorithm 1.

3.3.2 Criteria Selection. The criteria for each type of mutation simply follows the syntax of OpenMP language specifications. These criteria can be customized for any new type of mutations added. Here are some examples: "collapse" mutations are identified by an OpenMP directive followed by a rectangular, nested loop, within which there is no jump statements such as `break`, `continue` or `return`; "simd" and "tiling" mutations are identified by a serial or parallel loop statement without OpenMP parallel constructs or jump statements inside; lastly, every variable inside an OpenMP parallel region is checked for eligibility to become `firstprivate` variables. Some of the OpenMP mutations that can be applied to in the previously shown *matmul* example are shown in Figure 2. The one that shows the highest speedup in *matmul* is the tiling mutation.

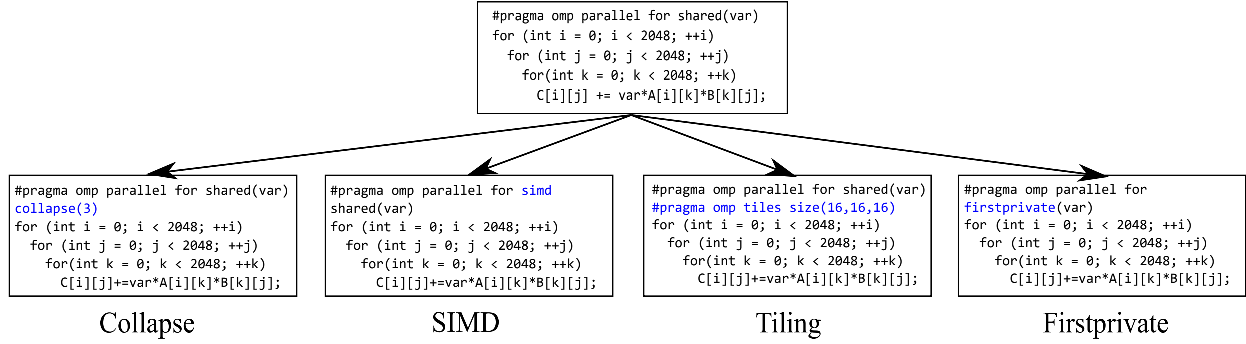

Figure 2. Classes of mutations in MUPPET.

Algorithm 1: The mutation generator algorithm.

```
1  Function GenerateMutations(StatementList):
2      M = ∅;
3      foreach Statement in StatementList do
4          if Statement is an OpenMP directive then
5              if can add collapse mutation then
6                  M = M ∪ CollapseMutation(Statement);
7              if can add SIMD mutation then
8                  M = M ∪ SIMDMutation(Statement);
9              if can add tiling mutation then
10                 M = M ∪ TilingMutation(Statement);
11             if can add firstprifate mutation then
12                 M = M ∪ FirstprivateMutation(Statement);
13         if Statement is a for loop then
14             if can add SIMD mutation then
15                 M = M ∪ SIMDMutation(Statement);
16     return M
```

3.4 Optimizer

Once a list of mutations is generated, it is exported to the Optimizer. This module runs an optimization algorithm specified by the end user to find the minimum point of $T' = P(M')$. During the optimization process, it finds specific points on the $T' = P(M')$ function by selecting/deselecting a subset of mutations, sending these mutations to the Transformer and Tester module, and receiving T' from the Transformer and Tester module once the mutated program has finished execution and running time statistics are collected.

MUPPET supports two optimization algorithms: Bayesian Optimization (BO) [34] and delta debugging [56]. The goal of these algorithms, albeit vastly different in implementation, is the same: find the subset of source mutations that would introduce maximum speedup. We selected BO because it is a common optimization algorithm that does not have the assumption of the function forms, which makes it an appropriate algorithm to use in MUPPET. Delta debugging, on the other hand, was originally developed as a software testing algorithm to isolate bugs inside a program, which is then adapted into finding speedup in program variants in previous work such as Precimonious [44] with regards to precision tuning. The inclusion in MUPPET of both algorithms shows how algorithms with vastly different original purposes can solve the same problem in different ways. MUPPET can also be extended to support other optimization algorithms such as genetic algorithm or simulated annealing.

For BO, since the input parameter of the function to be optimized, $T' = P(M')$, is a subset of mutations, which does not fit the function format of BO, we optimize $T' = P(Mb')$ instead where:

$$Mb' = \{mb_1, mb_2, ..., mb_n\}$$

$$mb_i = \begin{cases} 1, & \text{if} \quad m_i \in M' \\ 0, & \text{otherwise} \end{cases}$$

In this way, we convert the subset parameter into a list of binary parameters signaling whether a mutation is included in the subset so that BO can accept this list as input parameters for the function it optimizes.

As for delta debugging, we follow the LCCSEARCH algorithm in [44], where a change set in our adaptation of the algorithm is defined as the set of mutations that are applied to the original program, and the outputs are a minimal change set which causes speedup.

3.5 Transformer and Tester

The Transformer and Tester modules read the list of mutations from the Optimizer module, mutate the program into a variant, and run the variant to see if there is any speedup while maintaining the correctness of the program.

3.5.1 Compilation and Conflicts Checks. Even though there are already criteria placed in the Mutation Generator module for each mutation type to ensure that all mutations generated are syntactically correct, there are still situations where different mutations, when applied to the same programs at the same time, cause conflicts between them. If MUPPET lets these conflicts pass without checking during the transformer phase, it will cause a large number of mutated program variants that do not compile.

In order to save execution time, when the module transforms the program, it also statically checks and circumvents certain conflicts. These conflict checks can also be customized in the case where new types of mutations are implemented or new conflicts are discovered during testing. Currently, the conflict checks include: no tiling directives should be inside a SIMD region; and no SIMD directives or clauses should be inside a tile or collapse region.

4 Implementation Details

Muppet is implemented with a variety of programming languages and toolsets. The Mutation Generator and the Transformer modules are implemented via Clang plugins. Clang plugin system is one of several systems in the Clang compiler architecture that are capable of performing source-to-source code transformation, along with libtooling and libclang. Clang plugin is used so that our code transformation runs alongside the build environment of the evaluated programs, with the same kind of dependency checks. Muppet only requires minimal changes to the build scripts for it to work on new programs. This is described in 4.3.

The Optimizer and Tester modules, and the overarching framework managing the communication between modules, on the other hand, are implemented in Python. This is done to leverage the existence of a mature set of Python numerical optimization modules such as scikit-optimize [24].

4.1 Language Support

Muppet uses the modular approach; each of the three modules can be replaced in order to implement an analogous functionality. Currently Muppet targets C/C++ programs with OpenMP language constructs, though it is possible to target FORTRAN programs by rewriting the Mutation Generator and Transformer modules with a source-to-source FORTRAN compiler such as ROSE [40].

4.2 Customizing Muppet Runtime Parameters

Muppet supports BO and delta debugging in our implementation. BO is implemented with scikit-optimize, while delta debugging is implemented from scratch using the algorithm described in Precimonious [44], since it has no publicly available Python implementations.

Since running time for each program run may have variations that should not be counted as speedup, in order to suppress such variations, users can customize Muppet parameters to change how it measures running time. The *times* parameter specifies Muppet to run a number of repetitions for each variant, and collect running times for each run; the *shuffle* switch, only available for delta debugging, randomly shuffle the order of mutations so that delta debugging algorithm partitions these mutations differently each time (users can still specify the same random seed for the same shuffle result). Lastly, users can choose between using the minimum running time in all repetitions as program running time, or use the average running time.

4.3 Integrating New Programs with Muppet

For better management of programs in evaluation, Muppet calls a customized version of the FAROS build system [22]. Muppet calls a variety of functionalities offered in FAROS in order to analyze, transform, build and run the specified program. With FAROS, it is easy to add new programs to be mutated by simply adding new entries into the YAML config file.

4.3.1 Entries and Correctness.

An example entry for a locally stored simple matrix multiplication program is shown in Listing 2. It sets up commands for each step used in Muppet, such as building, calling plugins for mutations, running the program, extracting running time statistics from program output, and cleaning. The only required change to the *matmul* source code is (a) modify the build scripts (Makefile in this case) so that it accepts parameters for calling the Clang plugins; and (b) add correctness check code that parses program output in order to determine if the mutated program still runs correctly.

Listing 2. YAML config file for *matmul.*

```
1  matmul: fetch: 'cp -r ../../../extra/matmul .'
2    build_dir: 'matmul'
3    build: {
4      omp: ['make CC=clang++ OPT_LEVEL=3 OMP=1'],
5    }
6    call_plugin: {
7      analysis: ['make func_analysis OMP=1'],
8      mutate: ['make trans_mutations OMP=1'],
9    }
10   copy: ['matmul' ]
11   bin: 'matmul'
12   run: './matmul'
13   input: ''
14   measure: 'Work consumed (\d+\.\d+) seconds'
15   clean: 'rm -r *.*; cp ../../../../extra/
         matmul/*.* .'
```

5 Experimental Evaluation

This experimental evaluation answers the following research questions:

RQ1 Does Muppet discover source code mutations that induce speedup for OpenMP programs?

RQ2 What are the factors that may determine the efficacy of Muppet in finding these source code mutations?

5.1 Evaluation Setup

5.1.1 Benchmarks.

We use a set of 10 C/C++ OpenMP programs to evaluate Muppet. The programs include benchmark programs such as NPB-CPP [31] and HPCG [14], and proxy applications such as LULESH [26] and CoMD [18]. We

Table 1. Mutation speedup discovered by delta debugging and Bayesian Optimization.

(a) Delta debugging

Program	Original Min.	Best Min.	Speedup	Original Avg.	Best Avg.	Avg. Speedup	No. Possible Mutations (collapse/simd/firstprivate/tile)	No. Mutations in Best (collapse/simd/firstprivate/tile)
LULESH	11.095s	10.644s	4.23%	11.161s	10.741s	3.91%	0/95/0/222	0/7/0/8
HPCG	15.598s	14.071s	10.85%	15.654s	15.231s	2.78%	0/63/13/81	0/1/0/0
CoMD	2.296s	2.210s	3.90%	2.304s	2.232s	3.22%	0/78/13/132	0/25/4/7
FT.A	1.288s	1.244s	3.54%	1.305s	1.267s	3.06%	1/42/5/45	0/3/1/1
LU.A	4.933s	4.867s	1.35%	4.956s	4.890s	1.35%	3/100/6/186	0/1/0/1
MG.A	3.592s	3.124s	14.99%	3.603s	3.131s	15.07%	7/66/8/39	1/5/1/2
SP.A	31.534s	29.813s	5.77%	32.001s	30.948s	3.40%	64/267/3/396	0/6/0/4
BT.A	42.620s	42.344s	0.65%	42.686s	42.365s	0.75%	44/218/2/381	7/32/0/20
CG.B	22.432s	22.262s	0.76%	22.701s	22.376s	1.45%	0/18/11/27	0/1/1/1
EP.B	6.245s	6.238s	0.11%	6.251s	6.243s	0.13%	0/9/1/24	0/9/1/8

(b) Bayesian Optimization

Program	Original Min.	Best Min.	Speedup	Original Avg.	Best Avg.	Avg. Speedup	No. Possible Mutations (collapse/simd/firstprivate/tile)	No. Mutations in Best (collapse/simd/firstprivate/tile)
LULESH	11.080s	10.790s	2.69%	11.136s	10.881s	2.34%	0/95/0/222	0/47/0/62
HPCG	15.577s	13.599s	14.54%	15.693s	14.584s	7.61%	0/63/13/81	0/26/4/26
CoMD	2.300s	2.225s	3.36%	2.310s	2.254s	2.47%	0/78/13/132	0/37/10/37
FT.A	1.287s	1.261s	2.11%	1.295s	1.265s	2.37%	1/42/5/45	1/23/3/12
LU.A	4.906s	4.812s	1.94%	4.940s	4.872s	1.40%	3/100/6/186	2/46/2/54
MG.A	3.593s	3.132s	14.74%	3.624s	3.134s	15.64%	7/66/8/39	6/35/3/10
SP.A	30.819s	32.563s	-5.36%	31.330s	33.274s	-5.84%	64/267/3/396	28/138/2/118
BT.A	42.607s	43.370s	-1.76%	42.709s	43.437s	-1.68%	44/218/2/381	26/114/2/111
CG.B	22.504s	22.391s	0.51%	22.590s	22.466s	0.55%	0/18/11/27	0/9/6/9
EP.B	6.244s	6.237s	0.10%	6.247s	6.243s	0.07%	0/9/1/24	0/4/0/7

use these programs in order to evaluate the efficacy of Muppet in finding speedup in different programs, on a reference implementation or on manually optimized code.

On the benchmarks side, NPB-CPP is the C++ version of NAS Parallel Benchmarks ported to various programming frameworks on shared-memory architectures including OpenMP. We use 7 benchmark programs in varying problem sizes for evaluation: BT.A, CG.B, EP.B, FT.A, LU.A, MG.A, SP.A. HPCG is a benchmark program that performs multigrid preconditioned conjugate gradient iterations. We run it with a grid size of `96*96*96`. All benchmarks contain result verification routines in their source code, so we use them in order to determine program correctness.

On the proxy applications side, LULESH is a proxy application simulating the Shock Hydrodynamics Challenge Problem, while CoMD is a proxy application implementing classical molecular dynamics algorithms and workloads as used in materials science. Evaluating these programs may show the efficacy of Muppet in helping software developers in scientific computing optimize the parallel performance of their programs. LULESH is run with the parameter `-i 1500 -s 35`, and CoMD with `-e -i 1 -j 1 -k 1 -x 20 -y 20 -z 20`. We use the approach presented in [28] to determine the correctness of the program. For LULESH, we consider iteration count, final origin energy, and TotalAbsDiff as the output; for CoMD, we use the final energy as output.

5.1.2 Algorithm Parameters.

We use both BO and delta debugging in our experimental evaluation. Given the fact that program running time varies across the programs being evaluated, we put a tryout limit of 100 on both algorithms instead of using a total time limit. Our parameters for BO are $n_calls = 100$, $n_initial_points = 10$, and $noise = 0.01$.

5.1.3 Evaluation Environment.

We use a workstation computer with two 14-core Intel Xeon E5-2694v3 CPUs and 32GiB of RAM, running Ubuntu 22.04. We use Clang 16.0.6 with OpenMP 5.1 support as the compiler for both source-to-source code transformation, and for building and running the evaluated programs. Using OpenMP 5.1 enables us to build programs with `collapse` clauses as well.

We also ensure that performance variation is minimized between program runs. We avoid CPU context switching by limiting the programs to run on hardware threads on the second CPU by forcing the `taskset -c 14-27` command in FAROS. Hardware quiescing, as defined by [1], is also performed to reduce performance fluctuations, such as turning off both simultaneous multithreading and dynamic frequency scaling.

As for running time statistic collection, we run each mutated variant 5 times and use the minimum running time

as the program running time T. As a comparison, we also record the average running time for each tryout and evaluate if there is any possible discrepancy between average and minimum running time, but this statistic is not used as the fitness function output for optimization algorithms. We use the minimum running time for the fitness function because as stated in [1] it is best at rejecting noise introduced by the evaluation environment, since running time higher than the minimum must be due to such noise. However, we still calculate speedup for average running time to see how performance variability affects running time.

5.2 Evaluation Results

Even though we have taken various measures to reduce performance variability between each program run, it is still a factor that is not completely removed. Therefore, to determine if a program shows speedup when mutated, we use the 1% threshold. If amongst the 5 runs, the speedup between the minimum running time or between the average running time is lower than 1%, then the current subset of mutations is discarded.

The results of both algorithms can be found in Table 1. We compare the minimum running time for the mutated program against the minimum of the original (columns 2-4), and its average running time against the average running time of the original program (columns 5-7). Our evaluation shows that there are 7 out of 10 evaluated programs in which delta debugging can find a subset of mutations that, when applied, can cause speedup while maintaining the correctness of the program. The other 3 programs below the horizontal line in Table 1 show no speedup.

The speedup with regards to the minimum time ranges from 1.35% in LU to 14.99% in MG. Meanwhile, the speedup with regards to average time is generally about the same or smaller than the speedup with regards to the minimum time, especially so in HPCG where the speedup in average time is only 2.78% compared to the 10.85% speedup in minimum time. Such a drop in speedup is likely from increased inherent performance variability introduced by the sole mutation discovered. Running these programs more than 5 times, or further static program analysis, may be needed to determine a more robust speedup result. BO on the other hand can find mutation subsets that cause speedup in only 6 out of 10 programs, as it cannot find such a subset for SP. Furthermore, the speedup discovered is not greater than delta debugging except HPCG, which shows 7.61% in average running time speedup and 14.54% in minimum running time speedup.

We have also recorded the number of mutations that are applied in the subsets that cause the highest speedup with both algorithms (column 9 in Table 1), compared to the total number of possible mutations (column 8). The results in BO all have more mutations in the subsets, except EP which neither algorithm shows speedup. When we investigate all tryouts and their running times in programs such as SP, we deduce that a lot of mutations in these programs cause negative speedup, while only a few cause positive speedup. BO works worse in programs like these compared to delta debugging because it takes more tryouts than delta debugging to remove mutations with negative speedups from consideration. On the other hand, programs like HPCG likely have a few mutations that cause large speedup, but most others do not cause negative speedup. In these cases, BO works better than delta debugging and can find a subset of mutations that contain mutations with both large and small speedup.

6 Related Work

Mutation Testing. Mutation testing has been proposed to identify correctness defects [25]. The assumption in mutation testing is that a syntactic change (a mutant) can help discover programs' defects. Mutation testing, however, has not been applied deeply in HPC programs and on performance defects. Some attempts to build mutation testing for cloud systems have been reported [8]. Mutation operators (i.e., syntactic changes) have been proposed to reveal faults in small-size MPI programs [46]. With the increased use of LLVM, researchers are exploring the support of mutation testing in LLVM [11]. To the best of our knowledge, the only work that considers mutation testing for performance is [10]. However, this work does not consider parallelism and mutations in numerical (floating-point) code—these two aspects are critical to HPC applications. To the best of our knowledge, we are the first to explore using mutation testing for performance in OpenMP scientific codes.

General Auto-tuning. There is a significant corpus of past work on auto-tuning techniques. Typical examples include ATLAS [50], Active Harmony [48], FFTW [19], POET [54], CHILL [9], GEIST [49], OpenTuner [4], CLTune [38], Apollo [6, 52], and Dutta et al. [15, 16]. Their common theme is that they tune compile-time, such as tiling, or runtime parameters, such as the number of threads, presupposing a given source code representation of a program. Typical search algorithms for tuning they propose include random, grid, or Bayesian search, or various machine learning-based search models. By contrast, Muppet mutates the source code of the program, which exposes a large, combined set of both source code modifications as compile-time parameters and their possible configurations as runtime parameters to tune for. Furthermore, Muppet automates the generation of tuned source code variants without user intervention and it is the first to propose the delta debugging search algorithm for tuning. Integrating machine-learning techniques for fast searching in Muppet is an interesting future extension.

A number of papers research domain-specific tuning using code generation, alternate data layouts, or algorithmic parameters, such as [2, 13, 20, 27, 37] for linear algebra kernels and [33, 41, 42, 55] for stencils. Those approaches require users to express the programs in specialized domain-specific

languages amenable to tuning, which limits their generality. MUPPET tunes unaltered, user-provided, general OpenMP code to generate tuning source code variants and optimizing runtime parameters.

Auto-tuning OpenMP. Specifically on OpenMP, Adaptive OpenMP [23, 29], Sreenivasan et al. [47] propose OpenMP language extensions to support auto-tuning on OpenMP regions, such as scheduling policies of parallel loops, number of threads or teams. Those approaches require significant refactoring of the code and domain-specific knowledge from the programmer to successfully integrate tuning extensions and their possible configuration parameters in their OpenMP code. Instead, MUPPET treats source code modifications as a tunable parameter and independently explores the runtime configuration space.

Bliss [43] proposes probabilistic Bayesian optimization to tune hardware (core frequency, hyperthreading) and software execution parameters (OpenMP threads, algorithmic alternatives) for the whole application, specified by the user. Bliss does not modify the program's source code and tunes all regions in unison, by contrast, MUPPET both enables source code modifications and specializes tuning to each region, since mutations are region-specific.

Scalable Record-Replay [39] is a mechanism that extracts the LLVM IR of OpenMP GPU target region kernels to tune for each kernel in parallel the GPU launch bounds as compile-time parameters, by modifying the IR to re-compile, and the number of threads/teams as runtime parameters. Performing the kind of mutations in MUPPET on LLVM IR is challenging compared to source code, which motivates our choice of a source code mutation tool. Nevertheless, the idea of extracting OpenMP regions and tuning them independently is a possible extension to MUPPET to speed up search time.

7 Conclusion

We presented MUPPET, a novel application of mutation testing aimed at improving the performance of OpenMP programs. MUPPET uses different search algorithms to apply and compose program mutations to reduce application execution time. Because program transformations are performed at the source level, MUPPET's mutations are transferable across different OpenMP implementations and compilers. We demonstrate that MUPPET is capable of identifying mutations that improve performance in 70% of the evaluated programs achieving a maximum average speedup of 15.64%.

In the future, we plan to extend MUPPET to automatically update OpenMP code bases with the latest OpenMP features that improve performance while maintaining correctness. Currently, it is the responsibility of the code maintainer to manually update their code base to use newly available OpenMP features, which require significant manual efforts. The source code and data of MUPPET are publicly available at https://github.com/LLNL/MUPPET/.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-858593), the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0022182, and the National Science Foundation under award CCF-2119348.

References

[1] 2018. Performance Engineering of Software Systems. https://ocw.mit.edu/courses/6-172-performance-engineering-of-software-systems-fall-2018/
[2] Walid A. Abu-Sufah and Asma Abdel Karim. 2013. Auto-tuning of Sparse Matrix-Vector Multiplication on Graphics Processors. In *ISC (Lecture Notes in Computer Science)*, Vol. 7905. Springer, 151–164.
[3] Laksono Adhianto, S. Banerjee, Michael W. Fagan, et al. 2010. HPC-TOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.* 22, 6 (2010), 685–701.
[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, et al. 2014. OpenTuner: an extensible framework for program autotuning. In *PACT*. ACM, 303–316.
[5] Md. Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, et al. 2023. An Empirical Study of High Performance Computing (HPC) Performance Bugs. In *MSR*. IEEE, 194–206.
[6] David Beckingsale, Olga Pearce, Ignacio Laguna, et al. 2017. Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code. In *IPDPS*. IEEE Computer Society, 307–316.
[7] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, et al. 2013. Using automated performance modeling to find scalability bugs in complex codes. In *SC*. ACM, 45:1–45:12.
[8] Pablo C. Cañizares, Alberto Núñez, and Mercedes G. Merayo. 2018. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *J. Syst. Softw.* 143 (2018), 187–207.
[9] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Citeseer.
[10] Pedro Delgado-Pérez, Ana Belén Sánchez, Sergio Segura, et al. 2020. Performance mutation testing. *Software Testing, Verification and Reliability* (2020), e1728.
[11] Alex Denisov and Stanislav Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *ICST Workshops*. IEEE Computer Society, 25–31.
[12] Nan Ding and Samuel Williams. 2019. An Instruction Roofline Model for GPUs. In *PMBS@SC*. IEEE, 7–18.
[13] Jack J. Dongarra, Mark Gates, Jakub Kurzak, et al. 2018. Autotuning Numerical Dense Linear Algebra for Batched Computation With GPU Hardware Accelerators. *Proc. IEEE* 106, 11 (2018), 2040–2055.
[14] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *Int. J. High Perform. Comput. Appl.* 30, 1 (2016), 3–10.
[15] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, et al. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN. In *HPDC*. ACM, 45–57.
[16] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, et al. 2022. Pattern-based Autotuning of OpenMP Loops using Graph Neural Networks. In *AI4S*. IEEE, 26–31.
[17] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, et al. 2013. OMPT: An OpenMP tools application programming interface for performance analysis. In *International Workshop on OpenMP*. Springer, 171–185.

[18] The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). 2013. CoMD - Classical molecular dynamics proxy application. https://github.com/ECP-copa/CoMD.

[19] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[20] Mark Gates, Jakub Kurzak, Piotr Luszczek, et al. 2017. Autotuning Batch Cholesky Factorization in CUDA with Interleaved Layout of Matrices. In *IPDPS Workshops*. IEEE Computer Society, 1408–1417.

[21] Markus Geimer, Felix Wolf, Brian JN Wylie, et al. 2010. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 702–719.

[22] Giorgis Georgakoudis, Johannes Doerfert, Ignacio Laguna, et al. 2020. FAROS: A Framework to Analyze OpenMP Compilation Through Benchmarking and Compiler Optimization Analysis. In *IWOMP (Lecture Notes in Computer Science)*, Vol. 12295. Springer, 3–17.

[23] Giorgis Georgakoudis, Konstantinos Parasyris, Chunhua Liao, et al. 2023. Machine Learning-Driven Adaptive OpenMP For Portable Performance on Heterogeneous Systems. arXiv:cs.PL/2303.08873

[24] Tim Head, Manoj Kumar, Holger Nahrstaedt, et al. 2021. scikit-optimize/scikit-optimize.

[25] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[26] Ian Karlin, Abhinav Bhatele, Jeff Keasler, et al. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *IPDPS*. IEEE Computer Society, 919–932.

[27] Jakub Kurzak, Hartwig Anzt, Mark Gates, et al. 2016. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *IEEE Trans. Parallel Distributed Syst.* 27, 7 (2016), 2036–2048.

[28] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, et al. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *ISC (Lecture Notes in Computer Science)*, Vol. 11501. Springer, 227–246.

[29] Chunhua Liao, Daniel J Quinlan, Richard Vuduc, et al. 2009. Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 308–322.

[30] Yu Jung Lo, Samuel Williams, Brian Van Straalen, et al. 2014. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 129–148.

[31] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, et al. 2021. The NAS Parallel Benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Gener. Comput. Syst.* 125 (2021), 743–757.

[32] Diogo Marques, Helder Duarte, Aleksandar Ilic, et al. 2017. Performance Analysis with Cache-Aware Roofline Model in Intel Advisor. In *HPCS*. IEEE, 898–907.

[33] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, et al. 2020. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In *CGO*. ACM, 199–211.

[34] Jonas Mockus. 1994. Application of Bayesian approach to numerical methods of global and stochastic optimization. *J. Glob. Optim.* 4, 4 (1994), 347–365.

[35] Philip J Mucci, Shirley Browne, Christine Deane, et al. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710. Citeseer.

[36] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, et al. 2010. Statistical power modeling of GPU kernels using performance counters. In *International conference on green computing*. IEEE, 115–122.

[37] Rajib Nath, Stanimire Tomov, Jack Dongarra, et al. 2010. Autotuning dense linear algebra libraries on gpus and overview of the magma library. In *6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10)*.

[38] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A generic autotuner for OpenCL kernels. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 195–202.

[39] Konstantinos Parasyris, Giorgis Georgakoudis, Esteban Rangel, et al. 2023. Scalable Tuning of (OpenMP) GPU Applications via Kernel Record and Replay. In *SC*. ACM, 28:1–28:14.

[40] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[41] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, et al. 2018. Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations. *Proc. IEEE* 106, 11 (2018), 1902–1920. https://doi.org/10.1109/JPROC.2018.2862896

[42] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, et al. 2019. On Optimizing Complex Stencils on GPUs. In *IPDPS*. IEEE, 641–652.

[43] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, et al. 2021. Bliss: auto-tuning complex applications using a pool of diverse lightweight learning models. In *PLDI*. ACM, 1280–1295.

[44] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, et al. 2013. Precimonious: tuning assistant for floating-point precision. In *SC*, William Gropp and Satoshi Matsuoka (Eds.). ACM, 27.

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

[46] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. 2012. Mutation operators for concurrent programs in MPI. In *2012 13th Latin American Test Workshop (LATW)*. IEEE, 1–6.

[47] Vinu Sreenivasan, Rajath Javali, Mary Hall, et al. 2019. A framework for enabling OpenMP autotuning. In *International Workshop on OpenMP*. Springer, 50–60.

[48] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*. IEEE Computer Society Press, Washington, DC, USA, 1–11.

[49] Jayaraman J. Thiagarajan, Nikhil Jain, Rushil Anirudh, et al. 2018. Bootstrapping Parameter Space Exploration for Fast Tuning. In *ICS*. ACM, 385–395.

[50] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[51] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[52] Chad Wood, Giorgis Georgakoudis, David Beckingsale, et al. 2021. Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning. In *ISC (Lecture Notes in Computer Science)*, Vol. 12728. Springer, 453–472.

[53] Yi Yang, Ping Xiang, Mike Mantor, et al. 2012. Fixing performance bugs: An empirical study of open-source GPGPU programs. In *2012 41st International Conference on Parallel Processing*. IEEE, 329–339.

[54] Qing Yi, Keith Seymour, Haihang You, et al. 2007. POET: Parameterized optimizations for empirical tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[55] Xin You, Hailong Yang, Zhonghui Jiang, et al. 2021. DRStencil: Exploiting Data Reuse within Low-order Stencil on GPU. In *HPCC/DSS/SmartCity/DependSys*. IEEE, 63–70.

[56] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.