Understanding the Power of Evolutionary Computation for GPU Code Optimization

Jhe-Yu Liou[1] Muaaz Awan[2] Steven Hofmeyr[2] Stephanie Forrest[1] Carole-Jean Wu[1]

[1]*Arizona State University*
[2]*Lawrence Berkeley National Laboratory*

Abstract

Achieving high performance for GPU codes requires developers to have significant knowledge in parallel programming and GPU architectures, and in-depth understanding of the application. This combination makes it challenging to find performance optimizations for GPU-based applications, especially in scientific computing. This paper shows that significant speedups can be achieved on two quite different scientific workloads using the tool, GEVO, to improve performance over human-optimized GPU code. GEVO uses evolutionary computation to find code edits that improve the runtime of a multiple sequence alignment kernel and a SARS-CoV-2 simulation by 28.9% and 29% respectively. Further, when GEVO begins with an early, unoptimized version of the sequence alignment program, it finds an impressive 30 times speedup—a performance improvement similar to that of the hand-tuned version. This work presents an in-depth analysis of the discovered optimizations, revealing that the primary sources of improvement vary across applications; that most of the optimizations generalize across GPU architectures; and that several of the most important optimizations involve significant code interdependencies. The results showcase the potential of automated program optimization tools to help reduce the optimization burden for scientific computing developers and enhance performance portability for domain-specific accelerators.

1. Introduction

Graphics Processing Units (GPUs) are widely used to accelerate parallel applications in domains such as statistical modeling, machine learning, molecular simulations and bioinformatics, just to name a few. Although the tooling and programming language support for GPUs have matured, GPU programs remain challenging to optimize. Most GPU programs consist of parallel tasks, which compilers can optimize only to a certain extent, and issues such as thread mapping, communication, and synchronization are typically left for programmers to exploit manually. Consequently, applications often require hand-tuning to take full advantage of the GPU's computational power, which is time-consuming and requires significant expertise about parallel programming, underlying GPU architectures, and domain knowledge about the applications of interest.

To tackle the aforementioned challenges, prior works, such as [10, 43], have explored automated compilation optimization methods to reduce the programming and performance optimization burden on application programmers. One such approach uses evolutionary computation (EC) to optimize GPU programs represented in the LLVM [21] intermediate representation (LLVM-IR) [27]. The strength of this approach is its ability to explore optimization opportunities that don't preserve exact program semantics. An earlier study demonstrated that an EC-based approach achieved run-time improvements on a wide variety of general-purpose, but mostly unoptimized GPU programs by an average of 51%, performing especially well for error-tolerant applications. Despite these results, questions remain about *what optimizations such a method can find*, *how well it performs on hand-tuned production applications*, *how the optimizations are discovered*, and *how the method can be integrated into a production-level GPU application development*.

In this paper, we address these research questions by deploying GEVO [27] and analyzing the performance optimization opportunities on two important bioinformatics applications: gene sequence alignment and a SARS-CoV-2 infection simulation[1]. Aligning sequences of DNA, RNA or proteins is a fundamental operation in computational biology and underpins the success of many bioinformatics and medical applications [42]. The SARS-CoV-2 model (called *SIMCoV*) simulates how virus interacts with the patient's immune system while spreading in a human lung and causing tissue damage. Accelerating the performance of the SARS-CoV-2 simulation is crucial for understanding the many complexities of COVID-19.

Both sequence alignment and the SIMCoV simulations are highly computation-intensive. For example, in the first six months of 2021, over 6.7 million CPU hours were used for genome assembly on National Energy Research Scientific Computing Cluster (NERSC)'s Cori Supercomputer, with roughly 40% of the time spent in the sequence alignment kernel. Because of its importance, significant effort has been spent developing and manually optimizing ADEPT [2], a state-of-the-art GPU accelerated sequence alignment library. Similarly, it would take more than two weeks for SIMCoV to fully simulate a single infection, even for a single two-

1. The application source code and the performance optimization opportunities identified and presented in this paper are available at https://doi.org/10.6084/m9.figshare.21136768

dimensional slice of human lung tissue, on a modern, consumer-level CPU.

We deploy GEVO on two versions of ADEPT, downloaded from its public open-source code repository. ADEPT-V0 is the version of the code before hand-tuning, whereas ADEPT-V1 represents the hand-optimized version. We show that the performance of ADEPT-V0 can be improved by 30 times on state-of-the-art GPUs—a level of performance that is similar to the hand-tuned version. On the hand-tuned version (ADEPT-V1), an additional 28.9% speedup is achieved with GEVO-discovered optimizations. Similarly, GEVO finds optimizations providing 29% performance improvement for the SIMCoV simulation code running on the P100 GPU.

While GEVO does not enforce program semantics guarantee, we demonstrate that the benefits of automated program optimization tools are multi-dimensional, by using a tailored instrumentation of the program source code to localize the discovered optimizations and through a detailed performance analysis. Our results showcase the potential of automated program optimization tools to reduce the optimization burden for application developers, allowing them to focus on algorithms rather than details of hardware features and architecture specifics which are often black box or proprietary, and we show how such tools can actively influence the development of GPU application codes.

An important contribution of this work is its in-depth analysis of the discovered performance improvements, which can shed light on under-studied phenomena by slightly relaxing strict adherence to existing program semantics. Our analysis shows that a key source of the impressive performance improvements are multiple interdependent code modifications, known as *epistasis* in evolutionary biology. To gain insight into how the search process assembles these interdependent code modifications, we recapitulate and analyze the search history from an informative run. We also convert the discovered code LLVM-IR modifications back to the source code to characterize their contributions. To our knowledge, this is the first such study to reveal the importance of interdependencies in GPU code, which has implications for automated compiler optimization in general.

The contributions of the paper are summarized as follows:

- While EC methods have been shown in prior work [27] to improve the performance of naive GPU programs, we demonstrate that these methods can compete directly with human experts, outperforming even hand-tuned GPU programs (Section 4).
- We conduct a detailed study and code analysis to characterize performance improvements and to explain how the optimizations were discovered and achieved. Compared to earlier EC-based work on software, which typically uses one or two mutations to repair small bugs or otherwise improve software, we find optimizations that involve many more mutations. In some cases, a single GEVO optimized program contains nearly 1400 separate mutations (edits), of which up to 17 contribute significantly to the optimization. We define a multi-step process to identify relevant interdependent clusters and report how they were discovered (Section 5).
- We demonstrate the benefits of using EC methods in different program development stages, identifying performance hot-spots and strengthening a programmer's understanding of system performance improvement opportunities. The lessons learned can suggest further algorithmic improvements or manual adjustment of suggested optimizations, removing unwanted side effects if any.

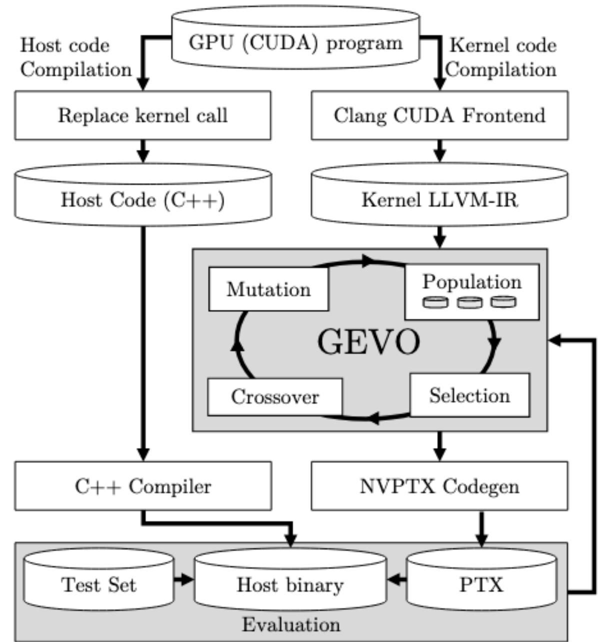


Figure 1: The GPU program compilation flow with GEVO interposed to dynamically modify and evaluate variants of the kernel code (gray blocks).

By focusing on two computation-intensive workloads, our analysis reveals the importance of manipulating interdependencies to find performance enhancements at the LLVM-IR level, highlighting why stochastic methods like EC are particularly suitable for accelerating execution time performance of domain-specific computations beyond what is achievable by algorithm and hardware domain experts.

2. Background

This section briefly describes GEVO, provides relevant background on ADEPT and SIMCoV, and gives details about their corresponding GPU-based implementations.

2.1. Evolutionary Search for GPU Code Optimizations

There is considerable interest in methods that automatically tune code after traditional compiler passes. Our work uses EC because it generalizes to large code sizes

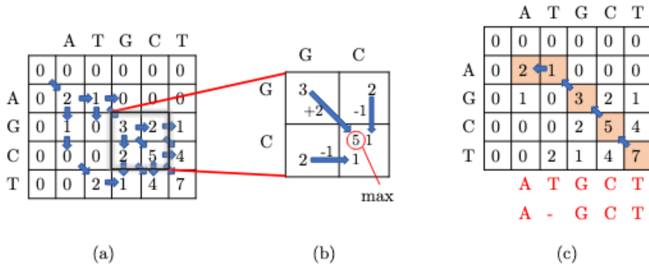


Figure 2: Example of the Smith-Waterman algorithm aligning two sequences, ATGCT and AGCT. (a) The forward pass calculates the scoring matrix with arrows showing how the scores are derived. (b) A single score calculation from the three neighboring cells. (c) The reverse pass from the calculated scoring matrix determines the alignment, with the final alignment result shown in the red text under the matrix.

and can be applied generically to many software problems, including automated bug repair [23, 55], energy reduction [6, 45], and run-time optimization [18, 54]. Many tools have been developed over the past decade for evolving program text [17, 23, 30, 50, 52, 55], and the vast majority of them operate on source code. In a nutshell, these methods start with a single program, generate an initial population of program variants using random mutation operators, validate each variant by running it on multiple test cases, evaluate the valid variants according to a fitness metric, and use this information to select the best individuals, which are then subjected to further mutation and recombined with one another to produce novel variants. This process is iterated until a time-out is reached or an acceptable solution is discovered. Mutation operators are readily implemented in source code or assembly, but the single static assignment discipline of LLVM-IR complicates their implementation considerably. The only mature EC tool that operates on LLVM-IR is GEVO (Gpu EVOlution) [27], which we adapted for the present work.

GEVO takes as input a GPU program, user-defined test cases, and a fitness function to be optimized, which in our case is runtime. Kernels that run on the GPU are first separated and compiled into LLVM-IR by the Clang compiler. GEVO takes these kernels as input, applies mutation and crossover to produce new kernel variants, and translates the implementations into PTX files. The mutations can either operate on an instruction (copy, delete, move, replace, or swap) or replace the operands between instructions. The host code running on the CPU is modified to load the generated PTX file into the GPU. GEVO then evaluates the kernel variant according to the fitness function. This process is illustrated in Figure 1.

2.2. Sequence Alignment

ADEPT implements Smith-Waterman, a widely used sequence alignment algorithm based on dynamic programming which guarantees an optimal *local* alignment between two given sequences [51].

2.2.1. Smith-Waterman Algorithm. Given two sequences $A = (a_1, a_2, ..., a_n)$, $B = (b_1, b_2, ..., b_m)$ to be aligned, a scoring matrix H is calculated with size $(n+1) \times (m+1)$, where n and m are the length of A and B (Figure 2(a)). The cell H_{ij} in the scoring matrix H represents the highest alignment score with sequences ending in the pair of a_i and b_j.

The cell score H_{ij} is calculated by maximizing over the values from three directions of prior alignments $(H_{i-1,j-1}, H_{i,j-1}, H_{i-1,j})$ (Figure 2(b)). The diagonal direction considers the similarity score s of the current pair a_i, b_j in the sequences, awarding the cell score (+2) if the paired a_i, b_j is matched and penalizing it (-2) otherwise. The vertical or horizontal direction introduces a gap in the current location of one sequence or another. Gap insertion penalizes the cell score with a smaller penalty (-1) than a sequence pair mismatch. How the score is awarded or penalized is arbitrarily determined and can be changed based on particular scenarios.

After the scoring matrix is obtained by iterating the cell score calculation from top left to bottom right, the optimal alignment is generated by tracing back from the highest score in the matrix H, traversing along the highest score in the region in the reverse direction from how the matrix was calculated, until score zero is reached (Figure 2(c)).

2.2.2. GPU-accelerated Smith-Waterman Algorithm. ADEPT parallelizes Smith-Waterman by offloading the computation of each column of the scoring matrix into one thread. As Figure 3 shows, the computation in each cell also depends on the scores of neighboring cells. Thus, the threads must be delayed, following the order of column index so the dependant values are ready to be shared from other threads.

In the GPU CUDA programming model, developers can exchange thread data through global/host memory, GPU device memory, shared memory, or private-thread register [39]. The first two memory types have no restriction on which threads can exchange data, but data stored in the shared memory and private thread register are visible only within a thread block and a warp, respectively. Despite much faster data access latency, private registers are unfriendly to programmers because they involve low-level, intrinsic instructions. To reduce data movement latency, ADEPT optimizations exploit both shared memory and private registers for data exchange.

2.3. Coronavirus Simulation Model

Moses et al. developed a computationally intensive, spatially explicit model (SIMCoV) to study why infection trajectories vary so widely across different patients, even those with identical co-morbidities. [31]. SIMCoV simulates both the spread of virus (SARS-CoV-2) through the complex physical structure of the lung and important aspects of the immune response, modeling the dynamics of four important elements: epithelial cells, virions, inflammatory signals, and T cells. Given a simulation space, e.g., for simplicity, consider a grid that represents a two-dimensional slice of lung tissue,

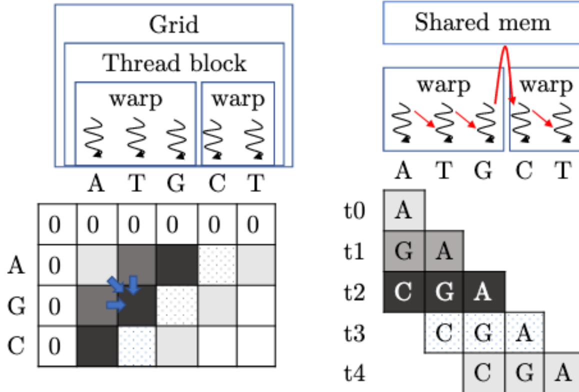


Figure 3: Illustration of the GPU-accelerated Smith-Waterman algorithm. The kernel runtime performance can be improved depending on the data communication patterns: spatial (bottom, left) vs. temporal (bottom, right).

the model is initialized with an epithelial cell at each grid point, and a set of infection sites. On each iteration, the model simulates four tasks for each grid point:

- Circulating T cells *extravasate* from the vascular system into the epithelial tissue with a probability determined by the presence of inflammatory signal.
- If the grid point contains a T cell, the T cell moves randomly to an adjacent location.
- Each epithelial cell's state is updated to one of: healthy, infected, apoptotic (in the process of dying), dead. Virions cause healthy cells to become infected, and infected cells eventually die. T Cells trigger cell death by binding to cells, preventing the further production of the virus.
- Virus and inflammatory signals diffuse from established sites of infection to neighboring grid points.

2.3.1. GPU-accelerated SIMCoV. SIMCoV's GPU implementation parallelizes its multi-core CPU implementation to construct GPU kernels by assigning each grid point's calculation to a thread. Over 90% of the GPU kernel runtime is spent moving T cells and spreading virus and inflammatory signals.

2.3.2. Stochastic Nature of the SIMCoV Simulation. Many components of the model simulation are stochastic, e.g., T cell generation and movement. This mimics biology but also poses validation challenges for GEVO, which must determine the correctness of any code modification. Fixing the random seed removes most of the stochasticity, but not all. For example, the simulation does not allow two T cells to move into the same grid point, which can cause a race condition. When such race conditions occur, the outcome is determined by the implementation of the GPU thread scheduler. This is an architecture-dependent approach and not transparent to application developers.

3. Experimental Setup

3.1. Compilation Preprocessing

First, we compile both the ADEPT and SIMCoV GPU kernels from CUDA into LLVM-IR using the Clang compiler. To enable code correspondence between the CUDA source and the GEVO-transformed codes, we instrumented the Clang compiler to enable source code debugging information and modified GEVO's mutation operator to encode the source code location information. Next, we modified both ADEPT's and SIMCoV's host code to invoke the GPU kernel from an external PTX file—the final product of the newly-mutated LLVM-IR that is executable by the CUDA binary. The host code is compiled using NVIDIA's nvcc compiler [34]. Figure 1 illustrates the compilation process.

3.2. Application Code

To study GEVO's effectiveness at different code development stages, we considered two versions of ADEPT:

- **ADEPT-V0** is the original parallel implementation (423 lines of code from one CUDA kernel, 1097 LLVM-IR instructions)
- **ADEPT-V1** is a manually-optimized version by an expert in both the application domain and GPU (623 lines of code from two CUDA kernels, 1707 LLVM-IR instructions).

ADEPT-V1 contains NVIDIA hardware-specific intrinsics, which use both shared memory and private registers for data exchanges (Section 2.2). ADEPT-V1 executes approximately 20-30 times faster than ADEPT-V0 across the GPUs used in this paper.

For SIMCoV, the only available GPU code to us was an initial GPU port from its multi-core CPU implementation, similar to ADEPT-V0, with 1197 line of code from 8 GPU kernels, translating to 1712 LLVM-IR instructions.

3.3. Validating Code Transformations

For ADEPT, We used the 30,000 pairs of DNA gene sequences in the ADEPT repository for fitness evaluation. In addition, we held out 4.6 million pairs of sequences to validate the final optimized ADEPT code. Although GEVO can trade off error tolerance against performance objectives, gene sequence alignment often requires strict accuracy so we require 100% accuracy for our ADEPT validation.

SIMCoV does not have a formal testing dataset for verification. Therefore, we controlled the simulation environment by fixing the initial random seed so that the simulation progress, including virus spread, epithelial cell state, and number of T cells is as similar as possible across runs. We use the simulation output generated from the unmodified SIMCoV as ground truth. To manage the remaining nondeterminism, we introduce the concepts of per-value mean and per-value variance to measure how close the output is to ground truth.

To evaluate fitness of a SIMCoV variant, we run the simulation on a small, 100x100 grid for 2500 simulation

TABLE 1: Architectural characteristics of the GPUs

GPU	P100	1080Ti	V100
Architecture Family	Pascal	Pascal	Volta
CUDA cores	3584	3584	5120
Core Frequency	1386 MHz	1999 MHz	1530 MHz
Memory Size	16GB HBM	11GB GDDR5X	16GB HBM2

steps, which is generally insufficient for the simulation to reach a steady state. Similar to ADEPT's held-out tests, we further validate the final GEVO optimized SIMCoV program after the run by both running the same 100x100 grid size for 10,000 simulation steps and by simulating a much larger, 2500x2500, grid. We were unable to run our optimized SIMCoV on a 10,000x10,000 grid, as the original paper did, due to the size limit of the GPU memory.

3.4. System Hardware

We evaluated and analyzed performance improvement using three generations of NVIDIA GPUs: P100 [37], 1080Ti GPU [36], and V100 [38], summarized in Table 1. We disabled the GPU Boost Technology [35] to maintain constant GPU operating frequency for the experiments. The machine with P100 GPU has a 20-core CPU with 256GB memory. For V100 GPU, we used NERSC's Cori Supercomputer's GPU partition [33]. In most cases, these runs used one V100 GPU with 10 CPU cores and 16GB memory.

3.5. GEVO Specification

Kernel execution time is the fitness target, averaged across all test cases. Individuals that fail one or more test cases are not part of the calculation. We set the population size to 256, retained the four best individuals into the next generation (elitism), applied crossover with 80% probability for each individual, and used a mutation probability of 30% per individual per generation. Different search budgets are given to GEVO for ADEPT (7 days) and SIMCoV (2 days), which roughly translates into 300 and 130 generations respectively.

4. Performance Evaluation Results

Summary: Figures 4 and 5 present the performance improvements for ADEPT-V0, ADEPT-V1, and SIMCoV on three generations of the GPUs. Execution time improved for ADEPT-V0 by 32.8X, 32X, and 18.36X on the P100, 1080ti, and V100 GPUs, reducing the kernel runtime from 2,362 ms to 72 ms, from 1442 ms to 45 ms, and from 918 ms to 50 ms, respectively. For the hand-tuned, well-optimized version, ADEPT-V1, GEVO finds an optimization that achieves 1.28X, 1.31X, and 1.17X performance improvement on the P100, 1080ti, and V100 GPUs. For SIMCoV, the performance improvement is 1.29X, 1.42X, and 1.16X on the P100, 1080ti, and V100 GPUs, respectively.

Because GEVO implements a stochastic search, we next ask how much variation there is across the experimental

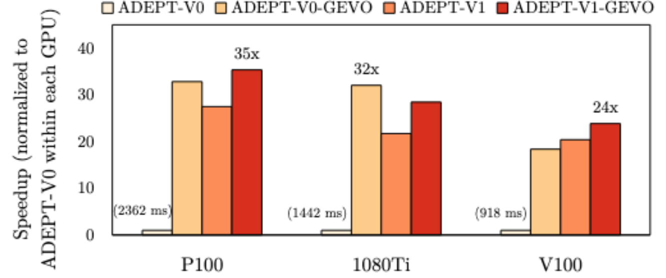


Figure 4: The performance results of ADEPT on the three generations of the GPUs.

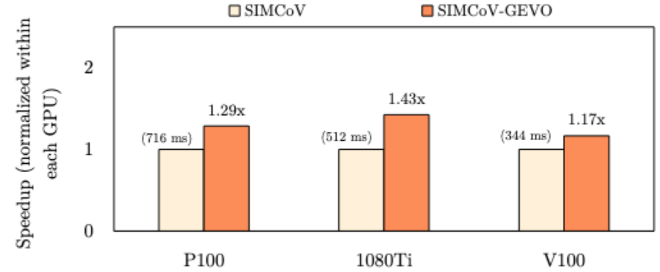


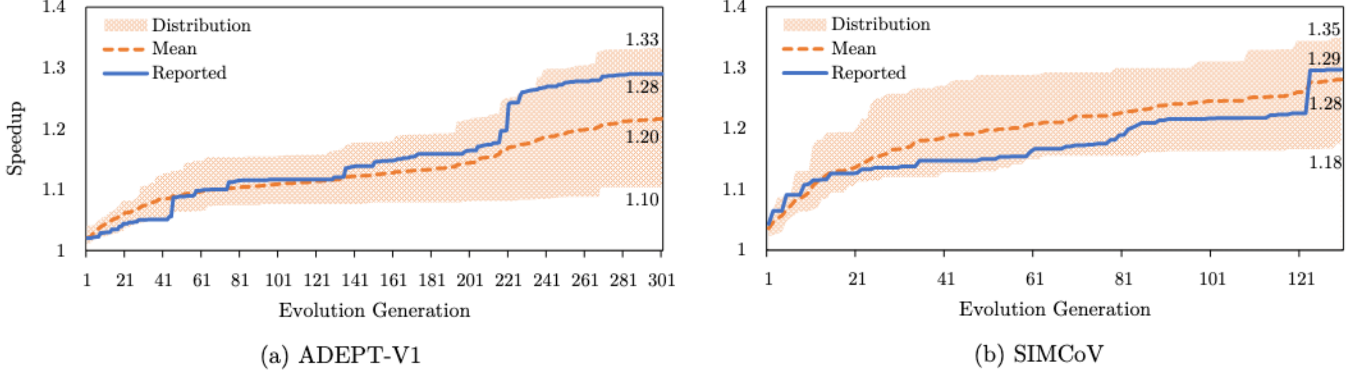
Figure 5: The performance results of SIMCoV on the three generations of the GPUs.

runs. Because the experiments are computationally expensive, we focused our analysis on on the P100 GPU and conducted ten independent runs for each configuration (Figure 6). For ADEPT-V1, compared with the initial run (1.29X improvement indicated by the solid blue line in Figure 6(a)), the highest speedup is 1.33X while the lowest is 1.1X. The mean is 1.20X and the variance is ±0.08. Figure 6(b) shows that, for SIMCoV, the highest speedup is 1.35X and the lowest is 1.18X, with a mean of 1.28X and variance of ±0.06. These results convey the value of running GEVO multiple times to discover the best possible optimization. The sources of the performance improvement for the ADEPT and SIMCov GPU codes are distinct, which we analyze and present in detail in Section 6.

Generality: To assess the portability of the discovered optimizations, we ran ADEPT-V0) (GEVO optimized for the P100) on the V100 GPU and compared its performance to ADEPT-V) GEVO optimized for the V100. The former achieves 99% of the performance gain of the latter and similarly for the other 1080 Ti GPUs, suggesting that many of the optimizations generalize across the three GPUs which feature distinct compute and memory architectures. We observed similar performance portability with SIMCoV. However, with ADEPT-V1, the same analysis showed that a small subset of the optimized code from the P100 GPU cannot run directly on the V100 GPU, suggesting that some performance optimizations are GPU architecture-dependent.

5. Understanding the Optimizations

To study the GEVO-discovered optimizations, we define a multi-step process, which first eliminates edits that contribute less than 1% performance improvement (*weak*



(a) ADEPT-V1 (b) SIMCoV

Figure 6: Distribution of performance improvements across ten GEVO runs for (a) ADEPT-V1 and (b) SIMCoV on P100. The shaded area encloses the historical path for all runs, while the dashed line indicates the average.

mutations), then separate out mutations (edits) that are independent, i.e. those that achieve greater than 1% fitness improvement independent of the other edits in the set. We can then conclude that the remaining mutations are interdependent (epistatic), but we do not know if the entire set is mutually interdependent, or if there are subsets. To find the subsets, we conduct an exhaustive search of all possible combinations of the epistatic edits, which is feasible because the total number of epistatic edits is small. (For example, the edit number is reduced to 12 from 1394 on ADEPT-V1) The following subsections describe each step in detail, primarily using ADEPT-V1 and SIMCoV on P100 as examples.

5.1. Edit Minimization

Overall, the best performing code variants from ADEPT-V1 and SIMCoV on a P100 GPU contained a total of 1394 and 384 mutations, respectively. It is surprising that the code is robust against so many mutations while preserving required functionality. To focus on the performance-critical changes, and to avoid side effects, we removed weak edits from consideration (Algorithm 1).

Algorithm 1 Identify weak edits

Parameter: Edit set $S = \{e_1, ..., e_n\}$
Function $f(S)$: measure the fitness (performance) of the program with edit set S applied

1: $weaks \leftarrow \emptyset$
2: **for each** $e_i \in S$ **do**
3: **if** $\frac{f(S - weaks) - f(S - weaks - e_i)}{f(S - weaks - e_i)} < 1\%$ **then**
4: $weaks \leftarrow weaks + e_i$

We systematically measured the performance difference between the program variant with and without each target mutation, in the context of all the remaining mutations. Any individual edit may not have an immediate impact on kernel execution time, but it could enable other higher-performing program mutants, serving as a kind of stepping stone. Our systematic reduction identifies these false-negative cases for weak edits. It is also possible that when weak edits are removed from consideration, we avoid a situation where multiple weak edits can potentially lead to an identical program variant. For example, suppose edits e_1 and e_2 are both stepping stones leading to e_3. In this case, e_1 and e_2 are redundant, and one of the two can be safely removed from the edit set without performance impact. We measure the 1% performance threshold using the *nvprof* profiling tool. This process reduces the number of code edits in our set from 1394 to 17 for ADEPT-V1 with minimal reduction of performance (0.9%), corresponding to performance improvement of 28% instead of 28.9%.

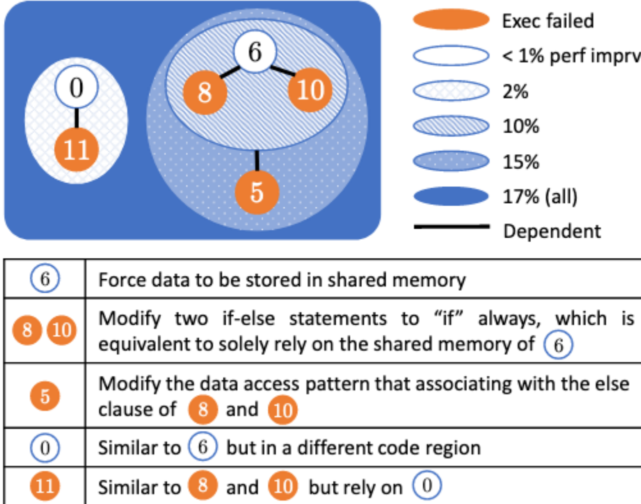
5.2. Edit Interactions

Algorithm 2 Separate independent and epistatic edits.

Parameter: Edit set $S = \{e_1, ..., e_n\}$
Function $f(S)$: measure the fitness (performance) of the program with edit set S applied

1: $Indep \leftarrow \emptyset$
2: **for each** $e_i \in S$ **do**
3: **if** $f(e_i)$ or $f(S - Indep - e_i)$ fails **then**
4: **continue**
5: $PerfIncr \leftarrow \frac{f(\emptyset) - f(e_i)}{f(\emptyset)}$
6: $PerfDecr \leftarrow \frac{f(S - Indep - e_i) - f(S - Indep)}{f(S - Indep - e_i)}$
7: **if** $PerfIncr \simeq PerfDecr$ **then**
8: $Indep \leftarrow Indep + e_i$
9: $Epistasis \leftarrow S - Indep$

Next, we describe how to identify interactions (epistasis) among edits, producing a set of independent edits and a set of epistatic edits (Algorithm 2). The algorithm first identifies the set of independent edits, and whatever remains after the procedure is considered to be epistatic. An independent edit must individually be both applicable and removable from the edit set (lines 4 and 5 of Algorithm 2) without causing an error. If it passes this check, we next evaluate how performance changes with and without the edit applied, first to the empty set of edits (i.e. to the original program) and then in the context of the remaining edit set (lines 6 to



Edit	Behavior
6	Force data to be stored in shared memory
8 10	Modify two if-else statements to "if" always, which is equivalent to solely rely on the shared memory of 6
5	Modify the data access pattern that associating with the else clause of 8 and 10
0	Similar to 6 but in a different code region
11	Similar to 8 and 10 but rely on 0

Figure 7: The edit relation graph and corresponding performance improvements for GEVO-optimized ADEPT-V1 on P100 GPU. Each node is an individual edit labeled with its index (the bottom table briefs each edit's behavior). The different backgrounds show the performance improvement for the different edit combinations, where the orange color indicates execution failure when applying certain edits individually, like edit 8.

9 of Algorithm 2). If the run-time from the above two tests agrees, the edit is identified as independent. In our running example, this algorithm divided the 17 significant edits from Section 5.1 into 5 independent and 12 epistatic edits. The two sets contribute 7% and 17% performance improvement to ADEPT-V1, respectively. Interestingly, we did not find performance-impactful epistatic edits for ADEPT-V0 or SIMCoV.

5.3. Epistatic Edit Set Analysis

While prior work in EC for software improvement rarely discovers epistasis (e.g., in bug repair it is usually one or two mutations), epistasis is common in biology [4]. We analyze the epistatic set for ADEPT-V1. This set consists of twelve edits. We show a dependency graph (Figure 7) for the most important epistatic clusters—determined by evaluating every subset of the epistatic set. The numbers in circles represent the edit index, and the black lines indicate a dependency relation.

There are two independent epistatic subgroups. One subgroup (edits 5, 6, 8, and 10) is the most significant, contributing 88.2% of the overall 17% performance improvement. Edits 8 and 10 both depend on the success of edit 6. The program mutants with either edit 8 or edit 10 individually fail the verification step. Edit 5 also fails individually and requires all three remaining edits (6, 8, and 10), to function properly. We consider this most significant cluster of edits in detail. Figure 8 shows when the edits were discovered and how the discovery affected fitness. As expected, edit 6 with no dependencies was discovered first, followed by edit

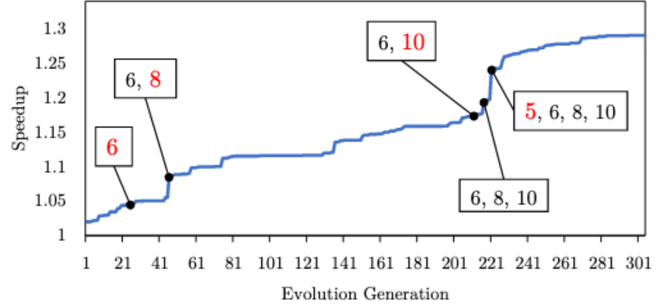


Figure 8: The discovery sequence for edits in the epistasis set (edits 5, 6, 8, and 10) across 303 generations. These are the same edits to ADEPT-V1 on P100 GPU shown in Figure 7. The group of edits in each box indicates in which generation this group was found, and edits colored red indicate the first time that edit was discovered.

8 in the 47th generation, edit 10 in the 213th generation, and edit 5 in the 221st generation.

The performance variation from run to run (figure 6(a)), was affected by the completeness of the discovered epistatic subgroups. For example, in the best run, GEVO further expanded the epistatic subgroup (e0, e11) to a 4-edit cluster similar to the subgroup (e5, e6, e8, e10). In the lowest performing run, GEVO discovered (e6, e10) but missed e8 and e5.

6. Functional Analysis of the Optimizations

This section explores the functional impact of the key mutations from Section 5. We do so by tracing each relevant code edit in the LLVM-IR level back to its corresponding CUDA source code. Although requiring significant manual effort, this is an important step in understanding the performance optimization opportunities that EC can uncover.

6.1. Rearrange Usage of Sub-Memory Systems on GPU

The epistatic edits identified in Section 5.2 alter how ADEPT-V1 uses the GPU's shared memory and private registers. By doing so, 15% performance improvement is achieved on the P100. These edits are applicable on the V100 as well, achieving similar performance improvement. Recall that, in Section 2.2, ADEPT-V1 uses both private registers and shared memory to exchange data. Its implementation is shown in Figure 9 with GEVO mutations indicated in red. These edits essentially eliminate the use of private registers and rely only on shared memory.

The *else* clauses at lines 19 and 28 are for the thread that meets the conditions to share data through private registers using the *shfl_sync* function. Due to a limitation of the GPU architecture, GPU threads that cannot exchange data through private registers communicate through shared memory. The effect of edits 8 (line 17) and 10 (line 26) is to drop the use of private registers. It is achieved by replacing the corresponding *if* condition with the existing boolean expression from line 14. If the boolean expression

```
1   ...
2   // if (laneId == 31)
3   if (landId == 0) { // edit 5
4     sh_prev_E[warpId] = _prev_E;
5     sh_prev_prev_H[warpId] = _prev_prev_H;}
6
7   // if(diag >= maxSize)
8   if (tID < minSize) { // edit 6
9     local_prev_E[tID] = _prev_E;
10    local_prev_prev_H[tID] = _prev_prev_H; }
11
12  __syncthreads();
13
14  if (is_valid[tID] && tID < minSize) {
15    ...
16    // if(diag >= maxSize) {
17    if (is_valid[tID])  // edit 8
18      eVal = local_prev_E[tID-1] + extendGap;
19    else {
20      if (warpId != 0 && landId == 0)
21        eVal = sh_prev_E[warpId-1];
22      else // private register
23        eVal = __shfl_sync(...); }
24
25    // if(diag >= maxSize) {
26    if (is_valid[tID])  // edit 10
27      final_H = local_prev_prev_H[tID-1];
28    else {
29      if (warpId != 0 && landId == 0)
30        final_H = sh_prev_prev_H[warpId-1];
31      else // private register
32        final_H = __shfl_sync(...);
33    } ...
```

Figure 9: Simplified code snippet from ADEPT-V1 for how data is exchanged using both private registers and shared memory. In edits 5, 6, 8, and 10 (red text, lines 3, 8, 17, and 26), GEVO eliminates private registers and uses shared memory instead.

in line 14 is true, both lines 17 and 26 are evaluated as true. This effectively causes every relevant GPU thread in the code snippet to write/read the data to/from the shared memory regardless of any other condition. However, edits 8 and 10 cannot be applied alone without edit 6 that implicitly enables every thread writing its data to the shared memory named *local_prev_XX*. After applying the three aforementioned edits, the shared memory named *sh_prev_XX* is not required, leading to edit 5. At this stage, a human developer would likely remove the entire *if* clause at lines 3 since the shared memory within the if clause is no longer referred to. Instead of removing the shared memory, edit 5 is introduced that only changes which thread will access the shared memory. This modification achieves the same performance improvement as if the affected code snippet were removed. We suspect that by changing the memory access pattern, as edit 5 does, the GPU can schedule the memory access differently to hide the memory latency of this particular access [24].

Accessing private registers on GPUs is much faster than the shared memory. So then, how do edits that leverage shared memory achieve performance advantage? This might be related to branch divergence. Recall from Section 2.2 and Figure 3, while some threads in a warp can use private registers for data sharing, there is often one thread, usually the first thread in the warp, that must communicate through shared memory. Combining with the GPU lock-step execution model, i.e., every thread in the same warp executes the same instruction at the same time, the aforementioned behavior guarantees branch divergence in the if-else region between lines 17-23 and 26-32. This essentially forces every thread in the same warp to run through both if and else regions, and whichever thread uses private registers has to wait for the slowest thread that accesses the shared memory to finish. As a result, the advantage of the fast access latency using the private registers is lost.

6.2. Remove Warp-Level Synchronization

The CUDA programming guide suggests that, before exchanging data through the private register, programmers should invoke a query function, such as *activemask* or *ballot_sync*, in order to return a mask indicating which threads are still alive in the warp. In particular, after the NVIDIA Volta GPU architecture (V100 GPU in our evaluation environment), *ballot_sync* should be used as the query function inside any conditional branch where branch divergence can happen. The reasoning is that the Volta architecture allows GPUs to subdivide a warp into subgroups to be scheduled independently, and *ballot_sync* implicitly forces the GPU to synchronize threads in the same warp.

Perhaps to be conservative, the developers of ADEPT used both *activemask* and *ballot_sync* before accessing the private registers in a conditional branch. An independent edit shows that removing *ballot_sync* yields 4% performance improvement on the V100 GPU but not on the P100 GPU. This supports the idea that *ballot_sync* performs warp-level synchronization on the Volta GPU architecture but not on the older GPU architectures. This edit is interesting because it violates the CUDA programming guide [40]. Yet, the edit passes all the verification tests. However, due to the proprietary design of the Volta GPU warp scheduler, we cannot conclude in which situations it is safe to remove warp-level synchronization.

6.3. Remove Unnecessary Memory Initialization and Synchronization Procedures

For ADEPT-V0, GEVO removed a small code region consisting of *memset* and *syncthread* functions for shared memory initialization and synchronization. This change improved the kernel performance by more than thirty-fold. In this case, it appears that we can completely ignore shared memory initialization, even on the algorithm level, because other edits were not engaged to compensate for the behavior change. In fact, the human expert also removed this code region in ADEPT-V1. Even if the initialization is required, the way it was implemented is vastly inefficient. The original code asks all the GPU threads to perform memory initialization on the same memory region. Combined with synchronization, GPU threads block each other to initialize

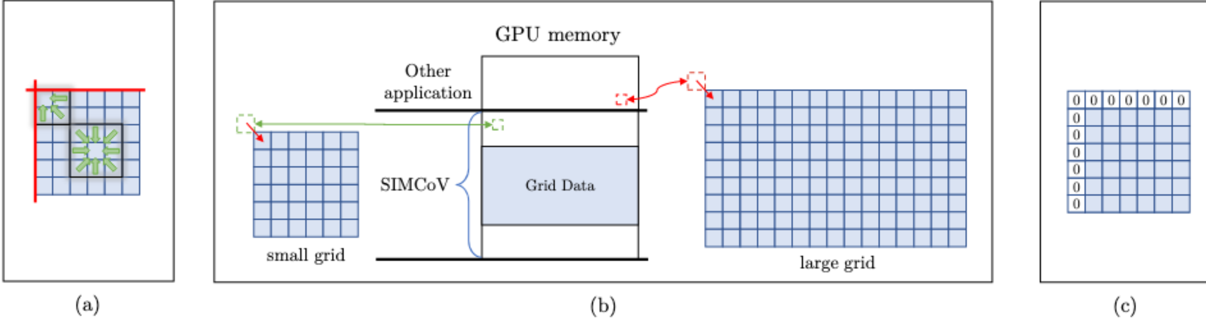


Figure 10: (a) illustrates that boundary check is a necessary step in the SIMCoV code. (b) illustrates how the boundary check removal is acceptable in a small grid but would fail for a large grid, which can be resolved by (c) padding the grid borders with extra grid points of 0 manually.

the same memory region over and over again, creating a significant performance bottleneck. The common practice is to initialize the memory through the CUDA API outside the kernel or through the in-kernel code using only one active thread. For application developers, the ability to quickly identify promising performance hot-spots that are challenging to discover using conventional tools is valuable, and this example highlights how GEVO supports this task.

6.4. Boundary Check Removal and Grid Padding

In SIMCoV, GEVO removed multiple conditional branches, which disabled a grid boundary check. Its purpose is to prevent errors when accumulating inflammatory signals from the neighboring grid points (the fourth task in Section 2.3). As Figure 10(a) shows, the boundary check prevents the edge grid points from attempting to accumulate values from points outside of the grid (illegal memory accesses). The performance analysis presented in this section addresses the following questions: **(1)** The boundary check optimization alone achieves 20% performance improvement. How does a simple boundary removal achieve such disproportional execution time improvement? **(2)** How can out-of-bound memory access not break the program's behavior?

To answer the first question, we examined the kernel with the modified code region. Surprisingly, a significant portion (31%) of the kernel instructions were performing logic operations related to the boundary comparison, although, as shown in Figure 10(a), the vast majority of the grid points are not located on the boundary. Removing the boundary check, however, is only legitimate if there is a compensating code modification to prevent illegal accesses outside the boundary. This example demonstrates how the GEVO approach can inform application developers. By actively searching through the code for performance optimization opportunities, the search can expose promising performance hot-spot regions that may be overlooked otherwise.

We answer the second question using validation test sets. That is, by running the SIMCoV simulation at a larger grid size: 2500x2500. Even though the SIMCoV code passes the initial test using a smaller simulation area, the boundary check optimization triggers a segmentation fault on this larger held-out test (Figure 10(b)). It is not surprising that larger held-out tests are needed during the optimization search process to detect such out-of-bound memory accesses, and this is a routine part of our evaluation strategy. After probing the code and the boundary check optimization more deeply, we observed that, by simply padding the grid borders with extra points of value 0 (Figure 10(c)), the application can achieve a 14% performance improvement with a negligible increase in the memory requirement.

6.5. Remaining Edits

We attempted to analyze all the edits, but there are some that we were unable to decipher. For example, one edit duplicates a memory write operation to a region that no subsequent code ever accesses. Such an operation seems redundant and could slow down program runtime. Surprisingly, it improves the kernel performance by 1% when run on the P100 GPU.

7. Discussion

Based on the mutational edit analysis described in Section 6, many relevant mutations are related to the GPU architecture. This implies that, although the GPU programming model has matured in the past decade or two, it is still difficult to master, especially for hardware-related programming language features. Scientific applications, such as those we consider here, are often written by domain experts who are not necessarily trained as software developers. In these circumstances, an approach such as GEVO is an interesting and promising choice for GPU code optimization [25–28]. We contacted the original developers of both ADEPT and SIMCoV, presented the discovered optimizations, and asked for their opinions and feedback. The developers were surprised that EC could synthesize code modifications with such large performance improvements. The main developer of ADEPT told us, "*If I was aware such an automatic optimization tool existed, it might have saved a couple of months of effort, especially for optimizing toward a specific GPU architecture!*" And, from

the developer of SIMCoV, "*When I looked at the optimizations found for SIMCoV, I saw how I could change my algorithm to improve its performance at scale. On CPUs, SIMCoV requires many cores to run useful simulations in a reasonable time. The CPU implementation bogs down when the simulated lung contains many agents, but the GPU version always loops over the full space so it does not suffer in this scenario.*"

Our results and the developer feedback from both ADEPT and SIMCoV illustrate two scenarios in the software development cycle where EC-based optimization can help: rapid prototyping in the early development stage and advanced fine-tuning in the final development stage. In the prototyping stage, the developer can quickly implement a workable but less-optimized version of the software and let EC perform code optimization searches, identify potentially-interesting performance critical regions, and address those inefficiencies. In the late development stage, EC can be deployed after hand-tuning by experts to search for additional optimizations.

A feature of our approach is that it does not require programmer domain knowledge for optimization. We acknowledge that EC-driven optimization does not necessarily preserve exact program semantics, which is both a strength and a limitation. It is a strength because small changes in semantics can lead to large runtime reduction, often without sacrificing functionality. It is a limitation because test suites are used to evaluate fitness and verify program behavior. With domain knowledge, developers can reason about the discovered optimizations, and either adopt them for better program performance, use them to improve the test suite, or use the insights to inspire related code enhancements, e.g., by introducing zero padding (Section 6.4). The results reported here for ADEPT did not require us to augment the test suite, an advantage of working with a deterministic program with an extensive test suite. However, if there are mutations that improve performance but do not make sense to programmers, like the one that introduced an additional memory write into an unused code location (Section 6.5), it may make sense for the programmer to eliminate the edit or design new tests.

GPUs are complex hardware with an equally complex programming environment. This is one reason why automated code optimization can be effective. Performant code can easily fail to provide expected performance, sending developers on a lengthy performance debugging journey. There is no golden rule for finding optimal performance on GPUs. For instance, higher concurrency does not guarantee higher performance, because in some cases using larger shared memory per block while minimizing occupancy may yield better throughput. Similarly, as demonstrated in the case of ADEPT, using a faster method of inter-thread communication (register to register transfer) does not imply the best performance. EC can automate this search for counter-intuitive optimizations while exploring hundreds of times more code modifications than a human developer can reasonably consider. We expect that the results achieved for ADEPT may generalize to other bioinformatics kernels and programs that use dynamic programming.

The final optimized sequence alignment program contains a large number of interacting edits, which is vastly more than what was reported by any earlier EC work for software (one or two edits is much more typical). This could arise from several factors: basic properties of the LLVM-IR representation and mutation operators, properties of GPU architectures, opportunities presented by the particular algorithms, or the implementation choices made by the developer—an avenue for future work. In particular, more effective epistasis is discovered in ADEPT-V1 than in ADEPT-V0. The developer-optimized codes in ADEPT-V1 might provide more paths for epistasis to surface since those optimized codes seem to be more resilient to our mutation operators. More generally, high-level languages are designed to help programmers express algorithms in a modular away that minimizes interactions between different parts of the code. So, it would not be surprising if their very structure works against epistasis. At the same time, the search space defined for a lower-level program representation like LLVM-IR is much larger than it is for source code, which would intuitively make search problems more challenging. How these factors balance out, and how to measure them remains an open question.

Regardless of their source, the fact that we found optimizations with such a high number of interacting edits shows how automated methods can discover complicated modifications to the target program, but it also presents challenges for deeper analysis. The approach used in the last step of our analysis involved exhaustively iterating through all the edit combinations. This will not scale well beyond the roughly twenty edits we considered.

8. Related Work

Beyond traditional compilers, the domain of automatic code optimization has three main branches: program synthesis [1, 3, 8, 15, 29], superoptimization [11, 43, 44, 48], and evolutionary computation [17, 54]. One key difference among the branches is the validation method. Program synthesis and superoptimization typically use a SAT/SMT solver [32] to check the logical equivalence of program rewrites, while EC relies on testing sets to encode the intended program specification. The trade-off is that the SAT/SMT solver guarantees program semantics but does not scale well, while test-based methods give up strict semantic equivalence but are more scalable. As a result, most earlier work in this domain applies only to programs of a limited length, usually under 200 lines of codes.

Deep learning methods have recently been used to analyze programs as well, including neural-network based logical reasoning [13, 41] and SAT solvers [47, 49] but also for superoptimization [9]. However, for optimizing parallel codes like GPU programs, EC may be more viable because logically reasoning about thread communications in a SAT solver requires deducing the entire parallel programming model in a logical form which is time-consuming and challenging.

EC is a popular approach for improving computer programs, e.g., to automatically repair bugs [12, 14, 22, 23, 53]. Surprisingly, prior analysis [46] showed that 20% to 40% of randomly generated program mutations (edits) have no observable functional effect (even when limited to only regions of the code that are actively tested), which suggested the possibility of using EC to optimize non-functional properties of software. As a result, EC has also been adopted to optimize software properties such as performance [54] and energy cost [5–7, 45].

Earlier EC work targeting GPU programs dates back to Sitthi-Amorn's work [50], which began with a basic lighting algorithm and used EC to gradually modify the shader program into a form that resembles an advanced algorithm proposed by domain experts. Later, Langdon et al. applied EC to a series of CUDA programs, ranging from compression methods [18] to RNA and DNA analysis [19, 20]. Specifically, BarraCUDA [16], a DNA sequence alignment program, was one of the target programs in the DNA analysis study [20]. However, their approach is different and less general than the one we used here. For example, the above works searched for parameter configurations outside the CUDA kernel such as the number of threads per thread block. The work manually parsed and transformed the CUDA kernel code into a custom-designed, line-based Backus Normal Form grammar as the code representation, where EC was applied. The performance improvements were attributed almost entirely to parameter tuning rather than modifying the kernel code. Orthogonal to the prior work, our approach finds performance optimization opportunities by transforming the implementation of functions. We instrument the modern LLVM compiler infrastructure to preprocess the CUDA program into LLVM-IR, a more general appoach that can be applied to any LLVM-IR program.

9. Conclusion

Optimizing GPU codes is a time-consuming process that requires deep knowledge in both the application domain and GPU architectures. This paper demonstrates the performance optimization potential of using GEVO on ADEPT, a GPU accelerated bioinformatics sequence alignment library, and SIMCoV, an agent-based COVID simulation of viral spread. We find improvements between 17% - 29% for ADEPT-V1, the expert-optimized version of ADEPT, and SIMCoV on various GPU platforms. Moreover, on ADEPT-V0, an earlier and less-optimized version, we find an incredible 30X improvement. This demonstrates the excellent potential of stochastic search methods such as GEVO to augment developer efforts to optimize GPU codes.

Our analysis of the evolved optimizations points to multiple interdependent edits, which leads to performance improvements that are challenging for human experts to discover. This is one of the strengths of our method, which we believe can augment code optimization to find useful interdependencies beyond what is achievable by application developers. As GPU architectures are still rapidly evolving, the availability of an automated code optimization tool to discover hidden performance optimization opportunities will continue to be useful as an aid to the code development process. We expect such methods to play an increasingly important role in lifting developer burden from focusing disproportionately on optimization, especially in cross-domain developments such as bioinformatics and many important application domains.

Acknowledgments

This work is supported in part by the National Science Foundation under grants CCF-1652132, CCF-1618039, and CCF-2211750. The authors acknowledge support for computational resources from the ASU Research Technology Office and the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. The authors would also like to thank Antonio Espinoza, Joshua Daymude, Joseph Renzullo, Kirtus Leyba, Pemma Reiter, and the anonymous reviewers for their valuable comments and suggestions.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*, 2013.

[2] M. G. Awan, J. Deslippe, A. Buluc, O. Selvitopi, S. Hofmeyr, L. Oliker, and K. Yelick, "Adept: a domain independent sequence alignment strategy for gpu architectures," *BMC bioinformatics*, vol. 21, no. 1, pp. 1–29, 2020.

[3] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron, "From relational verification to simd loop synthesis," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 123–134.

[4] W. Bateson, *Mendel's Principles of Heredity*. Cambridge: Cambridge University Press, 1909.

[5] A. Brownlee, J. Adair, S. Haraldsson, and J. Jabbo, "Exploring the accuracy-energy trade-off in machine learning," in *Genetic Improvement Workshop at 43rd International Conference on Software Engineering*. ACM, 2021.

[6] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation*, 2015.

[7] B. R. Bruce, J. Petke, M. Harman, and E. T. Barr, "Approximate oracles and synergy in software energy search spaces," *IEEE Transactions on Software Engineering*, 2018.

[8] S. Buchwald, A. Fried, and S. Hack, "Synthesizing an instruction selection rule library from semantic specifications," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 300–313.

[9] R. Bunel, A. Desmaison, M. P. Kumar, P. H. S. Torr, and P. Kohli, "Learning to superoptimize programs," in *International Conference on Learning Representations*, 2017.

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[11] B. Churchill, R. Sharma, J. Bastien, and A. Aiken, "Sound loop superoptimization for google native client," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 313–326, 2017.

[12] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of 3rd International Conference on Software Testing, Verification and Validation*, 2010.

[13] R. Evans, D. Saxton, D. Amos, P. Kohli, and E. Grefenstette, "Can neural networks understand logical entailment?" in *International Conference on Learning Representations*, 2018.

[14] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, 2009.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," *SIGPLAN Not.*, vol. 46, no. 6, p. 62–73, 2011.

[16] P. Klus, S. Lam, D. Lyberg, M. S. Cheung, G. Pullan, I. McFarlane, G. S. Yeo, and B. Y. Lam, "Barracuda-a fast short read sequence aligner using graphics processing units," *BMC research notes*, vol. 5, no. 1, pp. 1–7, 2012.

[17] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and computing*, vol. 4, no. 2, pp. 87–112, 1994.

[18] W. B. Langdon and M. Harman, "Evolving a cuda kernel from an nvidia template," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2010.

[19] ——, "Grow and graft a better cuda pknotsrg for rna pseudoknot free energy calculation," in *Proceedings of the Companion Publication of the 17th Annual Conference on Genetic and Evolutionary Computation*, 2015.

[20] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving cuda dna analysis software with genetic programming," in *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation*, 2015.

[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[23] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[24] S.-Y. Lee and C.-J. Wu, "Characterizing the latency hiding ability of gpus," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[25] J.-Y. Liou, S. Forrest, and C.-J. Wu, "Genetic improvement of gpu code," in *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*, 2019, pp. 20–27.

[26] ——, "Uncovering performance opportunities by relaxing program semantics of gpgpu kernels," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems: Workshop on Wild and Crazy Ideas (WACI)*, 2019.

[27] J.-Y. Liou, X. Wang, S. Forrest, and C.-J. Wu, "GEVO: Gpu code optimization using evolutionary computation," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3418055

[28] ——, "GEVO-ML: A proposal for optimizing ml code with evolutionary computation," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2020.

[29] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 90–121, 1980.

[30] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[31] M. E. Moses, S. Hofmeyr, J. L. Cannon, A. Andrews, R. Gridley, M. Hinga, K. Leyba, A. Pribisova, V. Surjadidjaja, H. Tasnim *et al.*, "Spatially distributed infection increases viral load in a computational model of sars-cov-2 lung infection," *PLoS computational biology*, vol. 17, no. 12, p. e1009735, 2021.

[32] L. D. Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of software, 14th International Conference on Tools and algorithms for the construction and analysis of systems*, 2008.

[33] NERSC. Cori gpu nodes. [Online]. Available: https://docs-dev.nersc.gov/cgpu/hardware/

[34] NVIDIA, "CUDA LLVM compiler," https://developer.nvidia.com/cuda-llvm-compiler/.

[35] ——, "GPU Boost," https://www.nvidia.com/en-us/geforce/technologies/gpu-boost/technology/.

[36] ——, "NVIDIA 1080ti GPU," https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1080-ti/.

[37] ——, "NVIDIA Tesla P100 GPU," https://www.nvidia.com/en-us/data-center/tesla-p100/.

[38] Nvidia, "NVIDIA V100 Tensor Core GPU," https://www.nvidia.com/en-us/data-center/v100/.

[39] NVIDIA, "Register Cache: Caching for Warp-Centric CUDA Programs," https://developer.nvidia.com/blog/register-cache-warp-cuda/.

[40] ——, "Using CUDA Warp-Level Primitives," https://developer.nvidia.com/blog/using-cuda-warp-level-primitive.

[41] A. Paliwal, S. Loos, M. Rabe, K. Bansal, and C. Szegedy, "Graph representations for higher-order logic and theorem proving," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 03, 2020, pp. 2967–2974.

[42] C. S. Pareek, R. Smoczynski, and A. Tretyn, "Sequencing technologies and genome sequencing," *Journal of applied genetics*, vol. 52, no. 4, pp. 413–435, 2011.

[43] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of ACM SIGARCH Computer Architecture News*, 2013.

[44] ——, "Stochastic optimization of floating-point programs with tunable precision," *SIGPLAN Not.*, vol. 49, no. 6, p. 53–64, 2014.

[45] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[46] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, 2014.

[47] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," in *International Conference on Learning Representations*, 2019.

[48] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Conditionally correct superoptimization," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[49] X. Si, Y. Yang, H. Dai, M. Naik, and L. Song, "Learning a meta-solver for syntax-guided program synthesis," in *International Conference on Learning Representations*, 2019.

[50] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence, "Genetic programming for shader simplification," in *Proceedings of the 2011 SIGGRAPH Asia Conference*, 2011.

[51] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[52] P. Walsh and C. Ryan, "Paragen: a novel technique for the autoparallelisation of sequential programs using gp," in *Proceedings of the 1st annual conference on genetic programming*, 1996, pp. 406–409.

[53] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[54] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Transactions on Evolutionary Computation*, 2011.

[55] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-objective Genetic Programming," *Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

Appendix

1. Abstract

We describe the necessary components and steps to evolve the two GPU programs (GPU-BSW and simcovGPU) which can be found in the artifact. This artifact also includes the evolving results that we discussed and analyzed in the paper.

2. Artifact check-list (meta-information)

- **Program:** GPU-BSW(ADEPT), simcovGPU
- **Compilation:** Python, C++, CUDA 11
- **Data set:** Gene sequence for GPU-BSW
- **Run-time environment:** Python 3.8, Ubuntu 20.04
- **Hardware:** We have tested on Nvidia P100, 1080Ti, and V100 GPU
- **Metrics:** Run-time reduction
- **Output:** Modified GPU program in LLVM-IR format
- **How much disk space required (approximately)?:** 5 GB per 7-day run
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 7 days
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:**
  - GPU-BSW: BSD-3-Clause-LBNL
  - simcovGPU: BSD-3-Clause-LBNL-UNM-ASU
- **Archived (provide DOI):** https://doi.org/10.6084/m9.figshare.21136768

3. Description

3.1. How to access. The target GPU programs' source code

- `GPU-BSW-iiswc.tgz,simcovGPU-iiswc.tgz`

and the evolving results

- `GPU-BSW-result.tgz,simcovGPU-result.tgz`

are available at https://doi.org/10.6084/m9.figshare.21136768

3.2. Hardware dependencies. The CUDA-compatible Nvidia GPU is required.

3.3. Software dependencies. GEVO is required to evolve the GPU program and can be installed through Python Package Index with the following command

```
> pip install gevo
```

It will also install all the python packages that GEVO requires. Furthermore, GEVO depends on CUDA-11 and LLVM-11 compiler infrastructure. Please refer to Nvidia and LLVM website for how to install CUDA-11 and LLVM-11 in the system.

4. Installation

The `GPU-BSW-iiswc.tgz` and `simcovGPU-iiswc.tgz` are the source code archives for GPU-BSW and simcovGPU. After downloading and extracting these two archives, we need to compile them into executable binary first.

For GPU-BSW:

```
> cd GPU-BSW-iiswc
> mkdir build
> cd build
> cmake ../ && make -j
```

For simcovGPU:

```
> cd simcovGPU-iiswc
> make gpu
```

Secondly, we need to compile the GPU kernel into LLVM-IR format that GEVO accepts as input with the following command in both program directories.

```
> make llvmir-opt
```

The `cuda-device-only-kernel.ll` will then be generated and ready for GEVO to evolve.

5. Experiment workflow

`evolve.sh` is the convenient script to start GEVO evolving the two target programs. It can be found in each program directory. This script has hard-coded the evolving parameters we used in the paper. The script also specifies the needed JSON file which tells GEVO how the target program is executed with needed arguments and which data-set is used in the verification process. Specifically, `dna_profile.json` and `profile_seed.json` are the JSON file for evolving GPU-BSW and simcovGPU respectively.

6. Evaluation and expected results

6.1. Evaluate modified GPU kernel. The `gevo-evaluate` is the tool under the GEVO framework to evaluate the modified GPU kernel. The following command demonstrates the basic usage of this tool.

```
> gevo-evaluate -P dna_profile.json -l
    gXX_maxerr.ll
```

This command evaluates both unmodified and modified GPU kernels and compares their performance difference. As before, the JSON file is to specify how the GPU program will be executed with necessary arguments and input data. It is not necessary to use the same JSON profile for

evolution and evaluation. For example, in GPU-BSW, we use `dna_profile.json` to evolve but use `dna_set2.json`, which specifies a much larger data-set, for the post-evolution evaluation.

6.2. Expected results. After GEVO starts, in each generation during the evolution, GEVO will print the best candidate's runtime performance. The best program candidate of each generation will be stored into `gXX_maxerr.ll` and `gXX_maxerr.edit` where "XX" is the number of generation. Here, `gXX_maxerr.ll` is the modified GPU kernel in LLVM-IR format whereas `gXX_maxerr.edit` contains all the GEVO edits that will transform the unmodified GPU kernel into `gXX_maxerr.ll`.

The artifact files:

- `GPU-BSW-result.tgz`
- `simcovGPU-result.tgz`

include the evolving result we reported, discussed, and analyzed in the paper, for GPU-BSW and simcovGPU respectively. Within each archive file, we label each GEVO run by the date it started as the folder name. The random seed used for each run is specified in the individual `evolve.sh` script and can be used to attempt reproducing the exactly same evolving history. However, the runtime measurement variation (For example, evaluating the same GPU kernel in multiple runs gives slightly different performance numbers) is not deterministic and cannot be eliminated. In the selection process during the evolution run, when comparing two equally performed GPU kernel candidates and determining who will stay in the population, this variation becomes the deciding factor no matter how small it is. Thus, completely reproducing the same evolving history cannot be achieved.