Evolving to Find Optimizations Humans Miss: Using Evolutionary Computation to Improve GPU Code for Bioinformatics Applications

JHE-YU LIOU, Arizona State University, Tempe, AZ, USA
MUAAZ AWAN, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
KIRTUS LEYBA and PETR ŠULC, Arizona State University, Tempe, AZ, USA
STEVEN HOFMEYR, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
CAROLE-JEAN WU, META, Menlo Park, CA, USA
STEPHANIE FORREST, Arizona State University, Tempe, AZ, USA

GPUs are used in many settings to accelerate large-scale scientific computation, including simulation, computational biology, and molecular dynamics. However, optimizing codes to run efficiently on GPUs requires developers to have both detailed understanding of the application logic and significant knowledge of parallel programming and GPU architectures. This paper shows that an automated GPU program optimization tool, GEVO, can leverage evolutionary computation to find code edits that reduce the runtime of three important applications, multiple sequence alignment, agent-based simulation and molecular dynamics codes, by 28.9%, 29%, and 17.8% respectively. The paper presents an in-depth analysis of the discovered optimizations, revealing that (1) several of the most important optimizations involve significant epistasis, (2) the primary sources of improvement are application-specific, and (3) many of the optimizations generalize across GPU architectures. In general, the discovered optimizations are not straightforward even for a GPU human expert, showcasing the potential of automated program optimization tools to both reduce the optimization burden for human domain experts and provide new insights for GPU experts.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Heuristic function construction*;

Additional Key Words and Phrases: Genetic improvement, Evolutionary programming, Bioinformatics, Genetic programming

We gratefully acknowledge support from ONR grants N000142012094, N000142112876, NSF DMR-2239518, NSF CCF-2211750, NSF OAC-2115075, ARPA-H(SP4701-23-C-0074), and the Santa Fe Institute.
Authors' Contact Information: Jhe-Yu Liou (corresponding author), Arizona State University, Tempe, AZ, USA; e-mail: jhe-yu.liou@asu.edu; Muaaz Awan, Lawrence Berkeley National Laboratory, Berkeley, CA, USA; e-mail: mgawan@lbl.gov; Kirtus Leyba, Arizona State University, Tempe, AZ, USA; e-mail: kleyba@asu.edu; Petr Šulc, Arizona State University, Tempe, AZ, USA; e-mail: psulc@asu.edu; Steven Hofmeyr, Lawrence Berkeley National Laboratory, Berkeley, CA, USA; e-mail: shofmeyr@lbl.gov; Carole-Jean Wu, META, Menlo Park, CA, USA; e-mail: carolejeanwu@meta.com; Stephanie Forrest, Arizona State University, Tempe, AZ, USA; e-mail: steph@asu.edu.

ACM 2688-3007/2024/11-ART21
https://doi.org/10.1145/3703920

ACM Reference format:
Jhe-Yu Liou, Muaaz Awan, Kirtus Leyba, Petr Šulc, Steven Hofmeyr, Carole-Jean Wu, and Stephanie Forrest. 2024. Evolving to Find Optimizations Humans Miss: Using Evolutionary Computation to Improve GPU Code for Bioinformatics Applications. *ACM Trans. Evol. Learn. Optim.* 4, 4, Article 21 (November 2024), 29 pages.
https://doi.org/10.1145/3703920

1 Introduction

The use of GPUs in bioinformatics applications has become increasingly important due to the growing size of biological datasets and the complex computations required for their analysis. The parallel architecture of GPUs can significantly accelerate many bioinformatics algorithms, such as sequence alignment [Klus et al., 2012; Korpar and Šikić, 2013; Liu et al., 2012], protein structure prediction [Mrozek et al., 2014; Pang et al., 2012; Stivala et al., 2010], agent-based simulation [Richmond et al., 2010], and molecular dynamics simulations [Eastman et al., 2013; Kylasa et al., 2014; Salomon-Ferrer et al., 2013]. This acceleration has led to faster and more extensive analyses of biological data, which in turn facilitates the discovery of new biological insights and the development of new treatments for diseases.

However, it is well known that maximizing the potential of GPUs can be a challenging task, for several reasons. First, GPU programming requires a different mindset compared to traditional CPU programming, including parallelization, memory management, and data transfer between the CPU and GPU. Second, GPUs often have a more complex architecture than CPUs, which requires specific optimization techniques. Last, GPU architectures evolve rapidly. Almost annually, GPU manufacturers, such as Nvidia and AMD, update their products with improved designs, which often introduce more specific optimization techniques. It is a challenging programming task for a GPU expert, not to mention for bioinformatics researchers who might not have a deep understanding of GPU architecture.

To address the aforementioned challenges for GPUs, prior works, such as [Grauer-Gray et al., 2012; van Werkhoven, 2019], explored automated compilation optimization to reduce the programming and performance optimization burden on application programmers. These approaches mostly apply their search methods on a predefined search space, such as to find the best combination of compiler flags or kernel configurations for specific GPU architecture. Another approach uses **evolutionary computation (EC)** to optimize GPU programs represented in the **LLVM** [Lattner and Adve, 2004] **intermediate representation (LLVM-IR)** [Liou et al., 2020a]. The strength of this approach is its ability to freely explore optimization opportunities that don't preserve exact program semantics. An earlier study demonstrated that the EC-based approach achieved runtime improvements on a wide variety of general-purpose, but mostly unoptimized, GPU programs by an average of 51%, performing especially well on 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 with an EC-based tool called **Gpu EVOlution (GEVO)** [Liou et al., 2020a], analyzing performance optimization opportunities for bioinformatics applications in three key fields: sequence alignment, a **SARS-CoV-2 (SIMCoV)** agent-based infection simulation, and a molecular dynamics code. Aligning sequences of DNA, RNA or proteins is a fundamental operation in computational biology and underpins the success of many bioinformatics and medical applications [Pareek et al., 2011]. The SIMCoV model simulates how the virus interacts with the immune system while spreading through a human lung and causing tissue damage. Accelerating the performance of the SIMCoV simulation is crucial for understanding the many complexities of COVID-19 and other respiratory infections. The **molecular dynamics application (oxDNA)** [Poppleton et al., 2023; Rovigatti et al., 2015; Snodin et al., 2015] is a coarse-grained model which represents each nucleotide in a DNA molecule as a single rigid body, with interactions between them parameterized empirically to reproduce structural, thermodynamic and mechanical properties of DNA. The model was primarily developed for simulations of designed DNA nanostructures and to handle the large system sizes and long simulation timescales needed to capture their properties.

All three applications are computation-intensive. For example, in the first 6 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 [Awan et al., 2020], a state-of-the-art GPU accelerated sequence alignment library which we use in our investigation. Similarly, on a modern, consumer-level CPU it would take over two weeks for SIMCoV to simulate a single infection trajectory, even for a single two-dimensional slice of human lung tissue. For oxDNA, sampling a single rigid DNA nanostructure on one consumer-level CPU requires on the order of three days to a week and much longer more for flexible designs.

The three applications represent three quite different types of bioinformatics applications: sequence alignment (which forms the core of many widely used bioinformatics tools); simulation (used for studying biological processes that are difficult or impossible to measure experimentally); and molecular dynamics (used widely for studying molecular-level genetic). The applications also represent different development stages, which allows us to observe how GEVO interacts with different stages of software maturity. We applied the GEVO optimization method to two versions of ADEPT, each downloaded from its public open-source code repository. ADEPT-V0 is the version of the code before hand-tuning, whereas ADEPT-V1 represents a 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. SIMCoV was, at the time of writing, in its early development stage where porting the CPU implementation to GPU just started. Despite less than participated performance gain, on SimCoV, GEVO finds optimizations providing 29% performance improvement for the simulation code running on the P100 GPU. Lastly, oxDNA is considered to have the most mature GPU implementation of over 10 years of development. Still, GEVO improves the performance of oxDNA simulation codes by over 17.8%.

Although GEVO does not enforce exact program semantics and relies instead on extensive test suites, 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 and optimization 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 a 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 several of the most impressive performance improvements arise from multiple interdependent code modifications or *epistasis*. To gain insight into how the search process assembles these interdependent code modifications, we recapitulate and analyze the history of 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 as optimization opportunities, which has implications for automated compiler optimization in general.

The main contributions of the paper are summarized as follows[1]:

— Although EC methods were shown in prior work [Liou et al., 2020a] to improve the performance of naive GPU programs, we demonstrate that EC can compete directly with human experts, outperforming hand-tuned (observed from ADEPT), sometimes even vendor built-in (observed from oxDNA), GPU programs (Section 4).
— We conduct a detailed study and code analysis to characterize discovered performance improvements in three bioinformatics applications and 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 hundreds of mutations, and we define a multi-step process to identify relevant interdependent clusters, reporting how they were discovered (Section 5).
— We demonstrate the benefits of using EC methods in earlier stages of GPU program development, identifying performance hot-spots and strengthening a programmer's understanding of system performance improvement opportunities. These lessons can suggest further algorithmic improvements to the programmer and/or manual adjustment of suggested optimizations, e.g., to avoid unwanted side effects if any.

By focusing on three 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 currently achievable by algorithm and hardware domain experts.

2 Preliminaries

This section first reviews how a GPU is programmed and what challenges a programmer might face, then it describes how our evolutionary algorithm, GEVO, searches for optimizations in GPU programs. We then provide relevant background on the three bioinformatic applications: ADEPT, SIMCoV, and oxDNA, and give details about their corresponding GPU implementations.

2.1 The Challenges of GPU Programming

GPU programming, like most parallel programming, requires programmers to define the kernel, a function that is repeatedly computed with different data input. The calculation is similar to a loop in which loop iterations are distributed as threads to different computing cores on the GPU for parallel execution. The first challenge for a programmer is to find a suitable property in their application and rewrite that part of the code into a kernel for the GPU to accelerate. During this process, programmers must decide how many iterations or threads (how many times the kernel executes) are needed, how they will be mapped to the GPU thread model, and how data are transferred into the kernel, including the pattern of data movement between CPU and GPU.

Up to this point, programmers do not need to know much about GPU hardware beyond the size of the GPU memory and the size of the kernel that can be executed on the target GPU. There are some common approaches, such as reducing $if/else$ statements as much as possible due to inefficient execution on GPU hardware, but how much this can achieve is constrained by the application logic. The situation becomes more complex, however, if data are communicated

[1]This study extends a prior work published in IISWC'22 [Liou et al., 2022]. The code and the benchmarks are available at https://github.com/lioujheyu/gevo/tree/master/benchmark

between threads. Programmers can separate calculations into multiple kernels and use global data communication between kernels, but this incurs high overhead. Or, if data communication is limited to a small area of threads, programmers can optimize the code with shared memory along with an in-kernel synchronization point, or even with private register sharing. Each of these options requires a certain understanding of GPU architecture, and in many cases details about the particular GPU are important. For example, the Nvidia V100 GPU can achieve fine-grain synchronization compared to its prior generation, P100 GPU, which allows the programmer to control the degree of synchronization for a smaller performance impact.

2.2 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 and can be applied generically to many software problems, including automated bug repair [Le Goues et al., 2011; Yuan and Banzhaf, 2020], energy reduction [Bruce et al., 2015; Schulte et al. 2014a], and runtime optimization [Langdon and Harman, 2010; White et al., 2011]. Many tools have been developed over the past decade for evolving program text [Le Goues et al., 2011; Marginean et al., 2019; Sitthi-Amorn et al., 2011; Walsh and Ryan, 1996; Yuan and Banzhaf, 2020], 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 (usually test cases), 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 that are readily implemented in source code or assembly (e.g., those that modify a single program statement) are more complex for the single static assignment discipline of LLVM-IR. The only mature EC tool that operates on LLVM-IR is GEVO [Liou et al., 2020a], 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. Although operand replacement can be an independent mutation, its primary use is for repairing mutations that break a value-use chain or to allow a value generated by a newly inserted instruction to be used in the computation. The host code running on the CPU is then modified to load the generated PTX file into the GPU. Finally, GEVO evaluates the kernel variant according to the fitness function. This process is illustrated in Figure 1. A full description of GEVO is given in [Liou et al., 2020a].

2.3 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 [Smith et al., 1981].

2.3.1 *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 respectively (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.

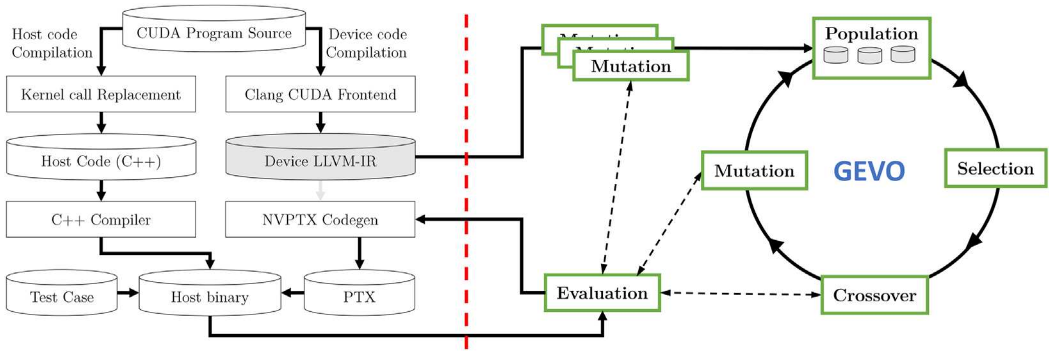


Fig. 1. The GPU program compilation flow with GEVO interposed to dynamically modify and evaluate variants of the kernel code.

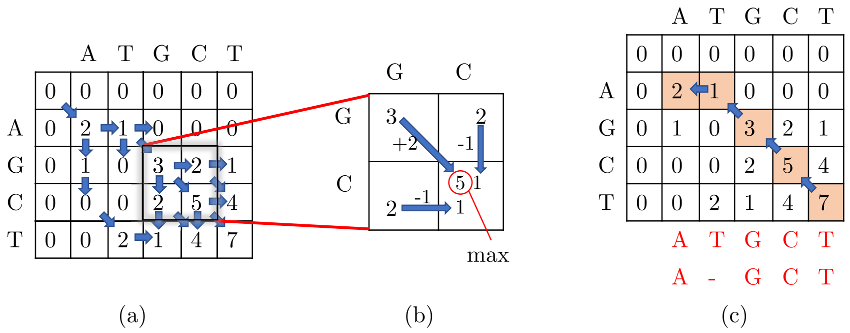


Fig. 2. Example of the Smith-Waterman algorithm aligning two sequences, ATGCT and AGCT. (a) The forward pass calculates the scoring matrix with arrows showing how the scores are derived. (b) A single score calculation from the three neighboring cells. (c) The reverse pass from the calculated scoring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

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)).

2.3.2 *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 dependent values are ready to be shared from other threads.

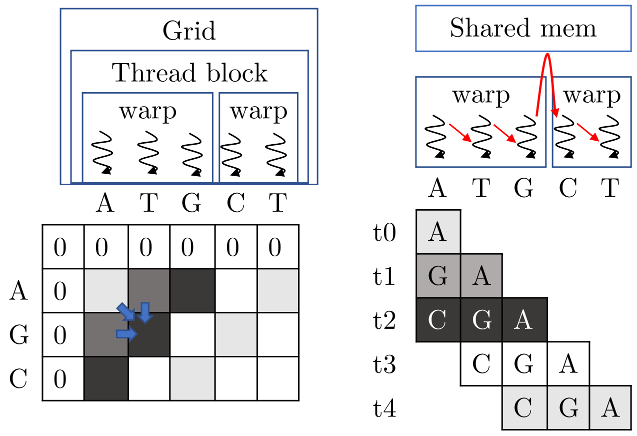


Fig. 3. Illustration of the GPU-accelerated Smith-Waterman algorithm. The kernel runtime performance can be improved depending on the data communication patterns: spatial (bottom, left) vs. temporal (bottom, right).

In the GPU CUDA programming model, developers can exchange thread data through global/host memory, GPU device memory, shared memory, or private-thread register [NVIDIA, 2017]. 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.

2.4 Coronavirus Simulation Model

Moses et al. developed a computationally intensive, spatially explicit model (SIMCoV) to study why SIMCoV infection trajectories vary so widely across different patients, even those with identical comorbidities [Moses et al., 2021]. SIMCoV simulates both the spread of virus (SIMCoV) through the complex physical structure of the lung and important aspects of the immune response. The model represents the spatio-temporal dynamics of four important elements: epithelial cells, virions, inflammatory signals, and T cells. Given a simulation space, the model is initialized with an epithelial cell at each relevant grid point (voxels containing lung tissue), and a set of infection sites. For simplicity in the following, we will consider a grid that represents a two-dimensional slice of lung tissue. On each iteration, the model simulates four tasks for each occupied grid point:

—Circulating T cells *extravasate* from the vascular system into the epithelial tissue with a probability determined by the presence of inflammatory signals.
—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), or dead. Virions (individual viruses) 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.

2.4.1 GPU-Accelerated SIMCoV. SIMCoV's GPU implementation parallelizes its multi-core CPU implementation to use GPU kernels by assigning each grid point's calculation to a thread. This leverages the fact that over 90% of the GPU kernel runtime is spent moving T cells and spreading virus and inflammatory signals.

2.4.2 Stochastic Nature of the SIMCoV Simulation. Many components of SIMCoV 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.

2.5 DNA Simulation Model Using Molecular Dynamics

Simulating nucleic acids is important from the fundamental point of view for understanding how biomacromolecules behave, and, from an application standpoint, it is important for predicting their behavior under particular conditions. However, detailed simulations of molecular dynamics are so computationally expensive that coarser-grained models have been developed, which describe nucleic acids at the nucleotide level [Doye et al., 2013]. oxDNA is an example of a coarse-grained model, and its software package has become a popular choice for investigating the dynamics, thermodynamics, and self-assembly behavior of DNA and RNA systems [Poppleton et al., 2021; Šulc et al., 2014]. To date, oxDNA has been used in more than a hundred publications,[2] but computational cost remains a challenge.

Initially developed for a single CPU platform, oxDNA has supported GPU acceleration since 2014. Its GPU implementation is considered to be mature, has been optimized over several years of development, and is thus an appealing example application for evaluating our GEVO-based approach. oxDNA leverages the GPU to parallelize the model in an edge-based approach. A thread is mapped for each interacting pair of particles, using atomic operations and Newton's third law to calculate the resulting force acting on each particle. For more details about the oxDNA implementation, the interested reader is referred to [Rovigatti et al., 2015].

3 Experimental Setup

This section describes how we set up our system for GEVO to optimize target applications. This includes how the applications are compiled, what hardware and system software we used, and how GEVO was configured for the experiments.

3.1 Compilation Preprocessing

First, we compile the ADEPT, SIMCoV, and oxDNA GPU kernels from CUDA into LLVM-IR using the Clang compiler with full optimization (Compiler flag: -O2). Additionally, to enable code correspondence between the CUDA source and the GEVO-transformed codes, we instrumented the Clang compiler to enable source code debugging information (Compiler flag: -g1) and modified GEVO's mutation operator to encode source code location for each mutation. Note that this additional step only adds source code information using the LLVM meta field without additional debugging instructions, meaning that performance isn't affected.

Next, we modified all three applications' (ADEPT, SIMCoV, and oxDNA) host code to invoke the GPU kernel from an external PTX file—the final product of a mutated LLVM-IR which is executable

[2] https://www.webofscience.com/wos/author/record/14753

by the CUDA binary. The host code is compiled using NVIDIA's nvcc compiler [NVIDIA, 2024a]. 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 source code from one CUDA kernel, 1,097 LLVM-IR instructions)
— *ADEPT-V1* is a manually-optimized version by an expert in both the application and GPU domains (623 lines of code from two CUDA kernels, 1,707 LLVM-IR instructions).

ADEPT-V1 contains NVIDIA hardware-specific intrinsics, which use both shared memory and private registers for data exchanges (Section 2.3). ADEPT-V1 executes twenty to thirty times faster than ADEPT-V0 across the GPUs used in this paper.

For SIMCoV, the only available GPU code was an initial GPU port from its multi-core CPU implementation, similar to ADEPT-V0, with 1,197 lines of source code from 8 GPU kernels, translating to 1712 LLVM-IR instructions.

For oxDNA, we asked GEVO to search for optimizations in almost all of its twelve GPU kernels, except for three kernels using the texture function which are not compilable by LLVM CUDA compiler. The targeted GPU kernels cover over 97% of the total runtime spent on GPU computation, and they comprise 1,023 lines of source code, although this count excludes many auxiliary functions which we have a hard time counting accurately (a rough estimation is that they contain 2,000+ lines of code in them). Nevertheless, the compiled GPU kernels, including those auxiliary functions mentioned above, have 13,748 LLVM-IR instructions.

3.3 Validating Code Transformations

It is important to verify that any code transformations imposed through mutation and crossover generate the same behavior as the original code. This is achieved by running a set of test data through both the modified and unmodified programs and comparing their results. However, this process also dominates the time cost of running GEVO on the three bioinformatics applications. To speed up the process, for each application, we divide the test sets into training sets and held-out sets. GEVO uses only the training tests during the search process. The training test sets are fairly small so that GEVO can run within a reasonable time budget (recall that we have to rerun the tests on each program variant that GEVO considers). After GEVO completes its optimization run, we manually verify the final optimized code using the held-out test sets. This ensures that the optimized application behaves the same as the original applications.

For ADEPT, We used the 30,000 pairs of DNA gene sequences in the ADEPT repository for fitness evaluation, holding out 4.6 million pairs of sequences to validate the final optimized ADEPT code. Each pair of DNA gene sequences is run through the alignment process once per fitness evaluation and generates one aligned sequence. Although GEVO can trade off error tolerance against performance objectives, gene sequence alignment usually requires strict accuracy, so we require 100% accuracy on the validation tests.

SIMCoV does not have a formal testing dataset for verification. Therefore, we controlled the simulation environment by fixing the initial random seed so the simulation's trajectory, including virus spread, epithelial cell state, and number of T cells was as similar as possible across runs. We use the simulation output generated from the unmodified SIMCoV as ground truth.

To evaluate the fitness of a SIMCoV variant, we run the simulation on a small, 100×100 grid for 2,500 simulation steps, which is generally insufficient for the simulation to reach a steady

Table 1. Architectural Characteristics of the GPUs

GPU	P100	1080Ti	V100	A100
Architecture Family	Pascal	Pascal	Volta	Ampere
CUDA cores	3,584	3,584	5,120	6,912
Core Frequency	1,386 MHz	1,999 MHz	1,530 MHz	1,410 MHz
Memory Size	16 GB HBM	11 GB GDDR5X	16 GB HBM2	40 GB HBM2e

state. To accommodate the simulation non-determinism, we introduce the concepts of per-value mean and per-value variance to measure how close the output is to ground truth. Initially, GEVO will run the unmodified SIMCoV ten times to collect the mean and variance of various metrics, such as virus or T cell count, on a per-gridpoint basis. The verification process then compares whether each metric generated by the GEVO-optimized simulation falls into its corresponding $mean \pm 3 * variance$. Similar to ADEPT's held-out tests, after the run completes we further validate the final GEVO-optimized SIMCoV program by first running the same 100×100 grid size for 10,000 simulation steps and then by simulating a much larger, $2{,}500 \times 2{,}500$, grid. We were unable to run our optimized SIMCoV on a $10{,}000 \times 10{,}000$ grid, as the original paper did, due to the size limit of our GPU memory.

For oxDNA, we set up a fairly small simulation environment, with 32,768 nucleotides simulated for 1,000 steps, as the test set for GEVO to use for fitness evaluation. This small simulation environment is included in the oxDNA repository. The simulation output contains the calculated energy and position on each nucleotide after a set number of steps. Due to a similar non-determinism issue to the one we faced in SIMCoV, We again apply the same per-value mean and per-value variance for the verification process. The same number of 32,768 nucleotides, but run longer for 100,000 simulation steps to a final steady state was then used as the held-out test.

Unlike CPU compiler, because of the lack of coverage statistics from both Nvidia CUDA compiler and Clang/LLVM, GEVO may modify the code outside the execution path of these test sets. Also, despite the use of both training and testing sets, the code transformation could potentially overfit the data. The former can be mitigated by the edit-minimization as a post-processing step introduced in Section 5.1. Still, careful analysis of the code transformation is required to make sense of their purpose and function (Section 6).

3.4 System Hardware and Software

We evaluated and analyzed performance improvement using three generations of NVIDIA GPUs: P100 [NVIDIA, 2024e], 1080Ti GPU [NVIDIA, 2024c], V100 [NVIDIA, 2024f], and A100 [NVIDIA, 2024d], summarized in Table 1. We disabled the GPU Boost Technology [NVIDIA, 2024b] to maintain constant GPU operating frequency for the experiments. The machine equipped with a P100 GPU features a 20-core CPU and 256 GB of memory, and the one with the A100 GPU has a 32-core CPU and 384 GB of memory. For the V100 GPU, we used the NERSC Cori Supercomputer's GPU instances [NERSC, 2024], which have one V100 GPU with 10 CPU cores and 16 GB memory in each instance.

All systems are configured with CUDA 11.4 with the Nvidia driver 470 installed. In addition, we developed our own profiling tool using Nvidia CUDA Profiling Tools Interface (CUPTI) instead of using Nvidia's default profiling tools (e.g., *nvprof* or *nsight*) to measure the kernel execution time. This reduced the overhead of the Nvidia profilers and made the profiling process consistent across

Nvidia GPU generations. Nvidia requires two profilers, nvprof and nsight, to profile a GPU before and after Pascal architecture.

3.5 GEVO Specification

Kernel execution time is the fitness target, averaged across all test cases in the test set. Individuals that fail one or more tests are deleted and not included in 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. These parameters are taken from the original GEVO paper, where they were determined empirically. Different search budgets were given to GEVO for ADEPT (7 days), SIMCoV (2 days), and oxDNA (7 days), which roughly translates to between 130 to 300 generations for each application.

4 Performance Evaluation Results

Figures 4, 5, and 6 report the performance improvements found by GEVO for ADEPT-V0, ADEPT-V1, SIMCoV, and oxDNA on four generations of the GPUs. Execution time improved for ADEPT-V0 by 32.8X, 32X, 18.36X, and 30.2X on the P100, 1080ti, V100, and A100 GPUs, reducing the kernel runtime from 2,362 ms to 72 ms, from 1,442 ms to 45 ms, from 918 ms to 50 ms, and from 638 ms to 21 ms, respectively. For the hand-tuned, well-optimized version, ADEPT-V1, GEVO found an optimization that achieves 1.28X, 1.31X, 1.17X, and 1.37X performance improvement on the P100, 1080ti, V100, and A100 GPUs. The performance improvements for SIMCoV and oxDNA are 1.29X and 1.18X on P100; 1.42X and 1.19X on 1080ti; 1.16X and 1.09X on V100; and 1.56X and 1.06X on A100 GPU. oxDNA developers report that the observed performance improvement on oxDNA through one Nvidia GPU generation is roughly 20%. Although the improvement that GEVO achieved on oxDNA is small compared to other applications, it is significant enough for consideration and analysis.

Because GEVO implements a stochastic search, we next ask how much variation there is across experimental runs. Each experiment is computationally expensive, so we focused our analysis on on the P100 GPU, conducting ten independent runs for each configuration (Figure 7). For ADEPT-V1, compared to the initial run (1.29X improvement indicated by the solid blue line in Figure 7(a)), the highest speedup found was 1.33X while the lowest was 1.1X. The mean is 1.20X and the variance is ±0.08. Figure 7(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. Figure 7(c) shows that for oxDNA, the highest speedup was 1.22X and the lowest was 1.13X, with a mean of 1.17X and variance of ±0.03. These results convey the value of running GEVO multiple times to discover the best possible optimization. The sources of performance improvement for ADEPT, SIMCoV, and oxDNA are quite distinct, which we analyze and discuss in detail in Section 6.

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-V0 which GEVO optimized natively for the V100. The former achieves 99% of the performance gain of the latter and similarly for the 1080Ti and A100 GPUs, suggesting that many of the optimizations generalize across the three GPUs, even though they feature distinct compute and memory architectures. We observed similar generality with optimized SIMCoV and oxDNA. 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 and A100 GPU, suggesting that some performance optimizations are GPU architecture-dependent.

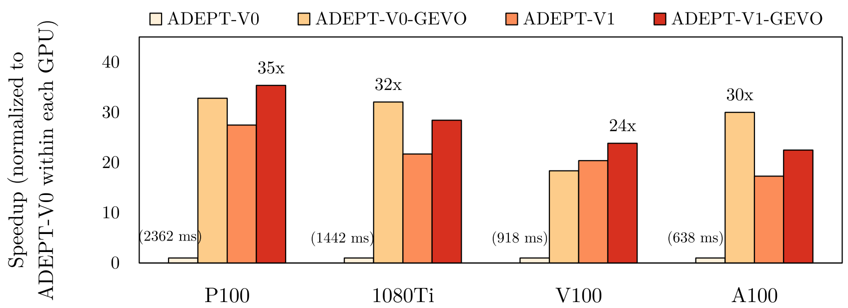


Fig. 4. Performance results for GEVO-optimized ADEPT on four generations of the GPUs.

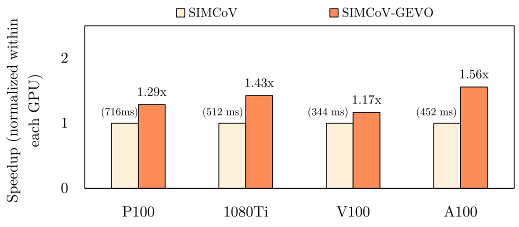


Fig. 5. Performance results for GEVO-optimized SIMCoV on four generations of the GPUs.

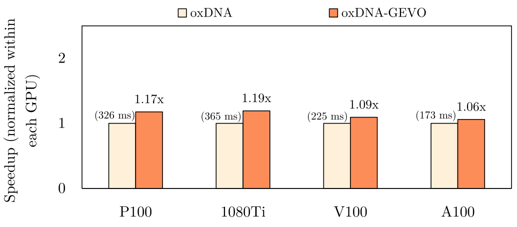


Fig. 6. Performance results for GEVO-optimized oxDNA on four generations of the GPUs.

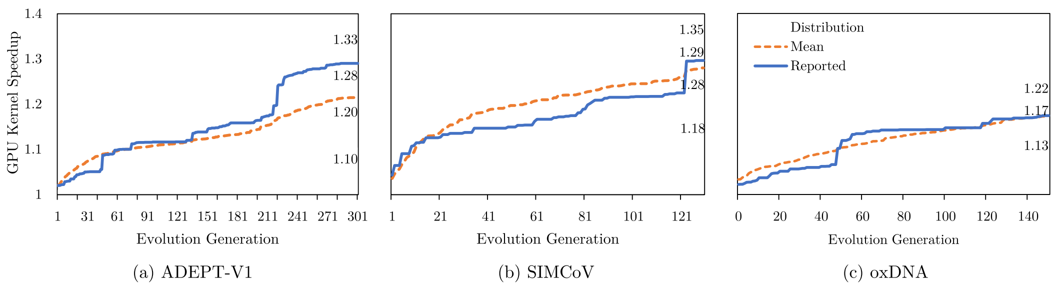


Fig. 7. Distribution of performance improvements across ten GEVO runs for (a) ADEPT-V1, (b) SIMCoV, and (c) oxDNA GPU kernels on the P100 GPU. The shaded area encloses the historical path for all runs, while the dashed line indicates the average.

5 Understanding the Optimizations

To study the GEVO-discovered optimizations, we first define a multi-step process which eliminates edits that contribute less than 1% performance improvement (*weak* mutations), then separates 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 on P100 as an example.

Algorithm 1: Identify Weak Edits

```
   Parameter: Edit set $S = \{e_1, ..., e_n\}$, Performance threshold $T\%$
   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 $f(S - weaks - e_i)$ fails then
4:         continue
5:     if $\frac{f(S - weaks) - f(S - weaks - e_i)}{f(S - weaks - e_i)} < T\%$ then
6:         $weaks \leftarrow weaks + e_i$
```

5.1 Edit Minimization

Overall, the best performing code variants from ADEPT-V1, SIMCoV, and oxDNA on a P100 GPU contained a total of 1394, 384, and 489 mutations, respectively. It is remarkable that the code is robust to so many mutations while preserving the ability to pass the validation test suite, especially because the total number of instructions in each kernel is relatively small. To focus on the performance-critical changes, and to avoid side effects, we removed weak edits from consideration (Algorithm 1).

We systematically measured the performance difference between the optimized program 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 to better fitness. Our systematic reduction identified these false-negative cases for weak edits. It is possible, however, that multiple weak edits can have an identical effect. 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. Our implementation removes whichever one is tested first.

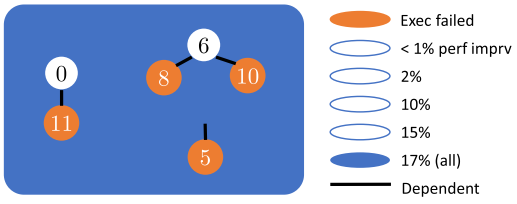
With the performance threshold set to 1%, the process outlined above reduces the number of code edits in our set from 1394 to 17 for ADEPT-V1, with a minimal reduction of performance (0.9%) from 28.9% to 28%. However, when we applied the 1% threshold to oxDNA's edits, our procedure reduced the number of edits from 489 to 8 and eliminated half of the performance improvements (17.8% to 8%). We then experimented with a more relaxed threshold of 0.5% for oxDNA, which only reduced the number of edits to 101 from 489, corresponding to a performance improvement of 14.3% instead of 17.8%. This result reveals that much of the improvement for oxDNA arises from many weak edits. This led us to study nearly all of oxDNA's edits to understand how the weak edits contribute to performance improvement (Section 6.5).

5.2 Edit Interactions

Next, we describe how to identify particular 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 9 of Algorithm 2). If the runtime 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

Algorithm 2: Separate Independent and Epistatic Edits

```
    Parameter: Edit set $S = \{e_1, \ldots, e_n\}$
    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:  $Epistasis \leftarrow S - Indep$
```



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

Fig. 8. Relationships among epistatic edits for GEVO-optimized ADEPT-V1 on P100 GPU, together with their corresponding performance improvement. Each node represents a single edit labeled with its index, and the table on the right briefly describes each edit's behavior. The different backgrounds indicate the performance improvement for the different edit combinations, where orange color denotes edits that have execution failures when applied individually, e.g., edit 8.

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, SIMCoV, or oxDNA.

5.3 Epistatic Edit Set Analysis

While prior work in EC for software improvement rarely discovers epistasis (e.g., in bug repair there are usually only one or two relevant mutations and when there are two, they rarely interact), epistasis is common in biology [Bateson, 1909]. Our analysis of epistasis in ADEPT-V1 identified twelve edits that interact with others in some way. Here, we show the dependency graph (Figure 8) for the most significant 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 in detail. Figure 9 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 8 in the 47th generation, edit 10 in the 213th generation, and edit 5 in the 221st generation.

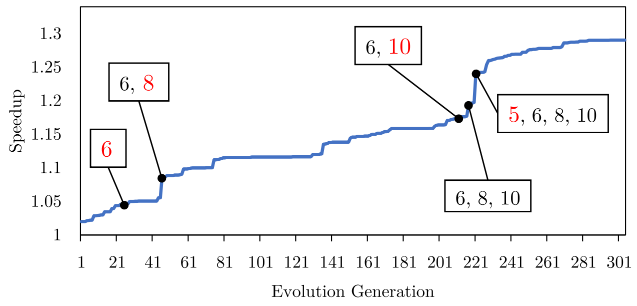


Fig. 9. The discovery sequence for edits in the largest epistasis set (edits 5, 6, 8, and 10) across 303 generations. These are the same edits to ADEPT-V1 shown in Figure 8. The group of edits in each box indicates in which generation this group was found, and edits colored red indicate the first time that edit was discovered.

The performance variation from run to run (Figure 7(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 identified in 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. We first consider important ADAPT-V1 optimizations (Sections 6.1, 6.2, and 6.3), then SIMCoV (Section 6.4), and finally oxDNA (Section 6.5).

6.1 Rearrange Usage of Sub-memory Systems on GPU

The epistatic edits identified in Section 5.3 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 2.3, ADEPT-V1 uses both private registers and shared memory to exchange data. Its implementation is shown in Figure 10 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 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 to write 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

```
 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    } ...
```

Fig. 10. 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.

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 [Lee and Wu, 2014].

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 2.3 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 and A100 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 [NVIDIA, 2018]. 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 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 2.4). As Figure 11(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 11(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 access outside the boundary. This example

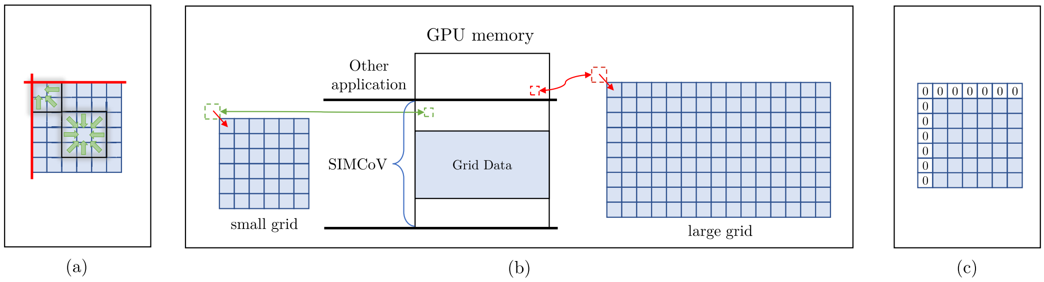


Fig. 11. (a) illustrates that boundary check is a necessary step in the SIMCoV code; (b) illustrates how the boundary check removal is acceptable in a small grid but would fail for a large grid, which can be resolved by (c) padding the grid borders with extra grid points of 0 manually.

Table 2. Edits Distribution across the Mathematic Functions in oxDNA

sine	cosine	arccos	log	sincos	abs
158	78	14	13	4	1

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: 2,500 × 2,500. 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 11(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 11(c)), the application can achieve a 14% performance improvement with a negligible increase in the memory requirement.

6.5 Optimizing Nvidia's Built-in Math Library

In oxDNA there were many weak edits, each contributing a small amount to the performance improvement. For each such edit we identified its source code location and discovered that 268 out of 489 of them were modifying functions inside the Nvidia built-in math library. Table 2 shows how the edits are distributed across the various functions in the library. This raises two additional questions:

—Why does GEVO modify the built-in math library with such a large number of edits?
—What are the optimizations doing?

We addressed the first question by examining the Nvidia built-in math library (*libdevice.bc*) and found that it is not in binary format. Instead, it is in the LLVM bitcode format which can be converted back to LLVM-IR format using the *llvm-dis* command-line tool. Inspection of the converted math library showed that these math functions are inline functions in which function invocations (the function call instruction) are replaced by the function body. As it happens, oxDNA uses a large number of inline sine/cosine functions (32 sine and 23 cosine separate function calls)

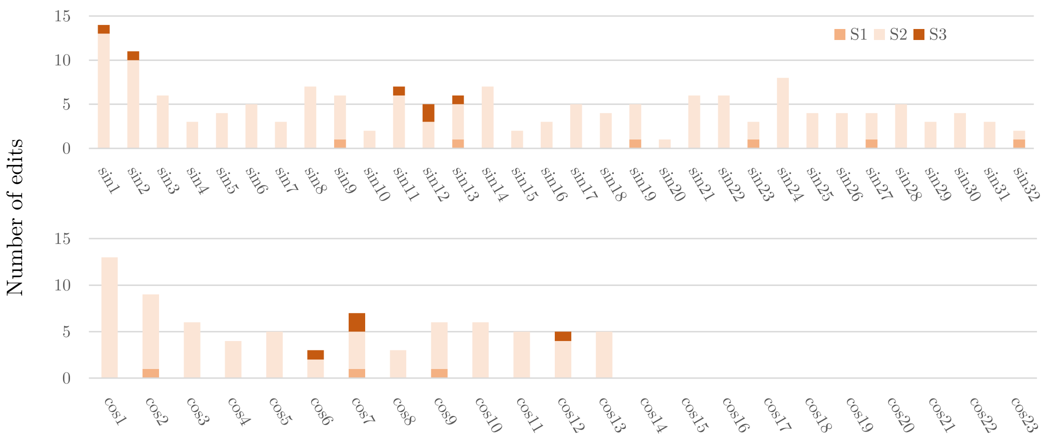
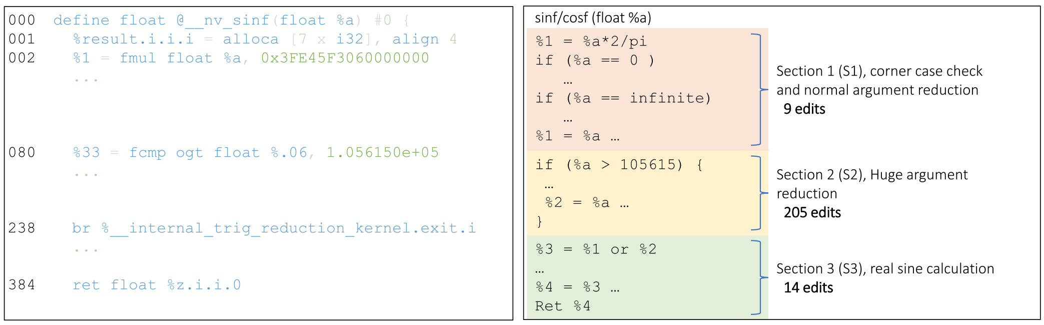


Fig. 12. Distribution of GEVO mutational edits across sine and cosine function instances and code section each edit falls in. S1, S2, and S3 are the code sections explained in Figure 13. For example, there are 14 edits in sin1 instances. 13 out of 14 edits are in section 2, 1 in section 3, and none in section 1. This figure shows that every sine function call in oxDNA receives at least one GEVO edit while there are ten cosine function calls with zero edits, implying that certain cosine functions cannot be optimized.



```
000   define float @__nv_sinf(float %a) #0 {
001     %result.i.i.i = alloca [7 x i32], align 4
002     %1 = fmul float %a, 0x3FE45F3060000000
        ...


080     %33 = fcmp ogt float %.06, 1.056150e+05
        ...


238     br %__internal_trig_reduction_kernel.exit.i
        ...

384     ret float %z.i.i.0
```

(a) sine in libdevice.bc in LLVM-IR

```
sinf/cosf (float %a)
%1 = %a*2/pi
if (%a == 0 )
  …
if (%a == infinite)
  …
%1 = %a …
if (%a > 105615) {
 …
 %2 = %a …
}
%3 = %1 or %2
…
%4 = %3 …
Ret %4
```

Section 1 (S1), corner case check and normal argument reduction
9 edits
Section 2 (S2), Huge argument reduction
205 edits
Section 3 (S3), real sine calculation
14 edits

(b) Manual decompile

Fig. 13. (a) shows the original sine function in LLVM-IR representation found in Nvidia built-in math library (*libdevice.bc*). The numbers on the left are the line numbers in the code. (b) is the author's manual decompilation from (a). Based on this understanding the code structure falls naturally into 3 sections, and the GEVO edits are categorized into each section. (In the original code, 0x3FE45F3060000000 in line 2 is the double-precision floating-point representation of $\frac{2}{\pi}$), but many other magic hexadecimal numbers similar to this appear in the code, which the authors could not decipher. The cosine function shares identical code structures except for a few instructions with different parameters.

to determine the relative position and orientation of nucleotides, compute the direction of the bonding force, and calculate their movement. Thus, each of the 228 out of 268 edits (as Table 2 shows) to the sine/cosine function apparently modifies a separate copy of the same sine/cosine code. Further investigation of these mutations showed that GEVO modified the function instances in different ways. And, certain cosine function calls remained unmodified, as Figure 12 shows.

Without access to the source code, it is challenging to fully characterize GEVO's optimizations, but reverse engineering provides some insight into how GEVO edits improved runtime. The Nvidia built-in sine and cosine functions have nearly identical code structures, which can be separated into three logical sections as shown in Figure 13. Notably, the majority of edits appear in Section 2 of the code path which is executed only when the input argument (angle) is larger than 105,615. We suspect that this code handles the rounding error of huge argument reduction. Since here are only $2\pi radians$ in a circle, any input to a sine/cosine function is equivalent to some angle in $[-\frac{\pi}{4}, \frac{\pi}{4}]$. Arguments that are outside this range are computed by reduction, subtracting integral multiples of $\frac{\pi}{2}$ as shown in Equation (1).

$$x - k \cdot \frac{\pi}{2} = r, k \in Z, r < 1. \tag{1}$$

For example, instead of computing *sin(x)*, the function computes *sin(r)* where r is in $[-\frac{\pi}{4}, \frac{\pi}{4}]$. When performing floating-point operations with a huge input argument x, the accuracy of computing r is dominated by the rounding error of $x - k \cdot \frac{\pi}{2}$, which is limited by the mantissa bits of floating-point representation. In short, the larger x is, the less accurate r that is produced. This is a well-known problem discovered in 1992 [Ng, 1992], with many follow-up discussions [Boldo et al., 2008; Brisebarre et al., 2005; De Dinechin et al., 2019; Henderson, 2000] and solutions that are incorporated into compilers such as gcc and clang.

Eliding many details about how rounding errors are managed in huge argument cases, we believe that the modifications GEVO made to the Section 2 code region disables this functionality using various strategies. We scanned all oxDNA input values to the sine function and discovered that they are always smaller than the Nvidia threshold value for activating fixed-point argument reduction. Thus, it is safe for oxDNA to disable argument reduction. Next, we manually disabled the argument reduction of the sine function in the Nvidia math library, and tested it in a standalone CUDA kernel where only the sine function is invoked with a large input range. This standalone test shows that the observable error, compared to the unmodified sine function, increases only after the input is greater than 105,615. More importantly, the performance of the sine function in the standalone environment on P100 improved by 54%, regardless of the input value. It is surprising that the performance improvement is so large, even when the input values do not require reduction. Finally, disabling huge argument reduction through directly modifying in the sine function in Nvidia math library and then recompiling oxDNA improved the performance by 4.7%.

Recall that certain cosine functions in oxDNA received zero edits as Figure 12 shows. We discovered that, unlike the sine functions, some oxDNA cosine functions do receive input arguments larger than 105,615, which likely prevents GEVO from optimizing these parts of the code.

There are some small edits in section 1 of Figure 13(b) similar to those discussed above, but they disable checks on other corner cases. For example, one edit disables checking for infinite input arguments in the floating-point format. Notably, shown in Figure 12, only certain function instances receive such edits, meaning they are only applicable based on the different use cases of those function instances. In hindsight, a human expert might devise cleaner ways to streamline these corner cases, but these results show that GEVO can discover many optimization opportunities in built-in functions per function usage.

6.6 Dead Code Removal

In oxDNA, all calculations are performed in three-dimensional space. For historical reasons dating to the early development of oxDNA, some important variables are defined in a *struct* with four elements, and the last element is never used. In the past, GPU programmers often used internal data structures for color space with RGBA 4 elements to store vector-like data. However, this is no longer required. GEVO discovered this redundancy and removed a few load instructions for the 4th element when passing variables between functions.

Normally, such redundant code could be detected and optimized by the compiler as part of the dead code removal pass. We are unsure what condition in this case prevents the LLVM compiler from performing or detecting the dead code.

6.7 Remaining Edits

We attempted to analyze every GEVO edit that has a performance impact greater than 1%, 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 should slow down program runtime. Surprisingly, it improves the kernel performance by 1% when run on the P100 GPU.

7 Discussion

The mutational edit analysis (Section 6) showed that many performance-enhancing 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 hardware-related programming language features. Bioinformatic 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 appealing choice for GPU code optimization [Liou et al., 2019a, 2019b, 2020a, 2020b]. When we discussed GEVO's optimizations with the original developers of both ADEPT and SIMCoV, they were both surprised that EC could discover 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 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.

Although the developers of oxDNA have requested our assistance to incorporate GEVO's optimizations into their latest code release, modifying built-in libraries is generally beyond the scope of most developers, particularly, without access to source code. In this context, GEVO provides an opportunity for library or compilers developers, including engineers inside a GPU manufacturer like Nvidia, discover optimization opportunities. In fact, we learned that LLVM provides a series of instruction flags for floating operations to individually control many fast-math flags such as disabling checking infinite or not-a-number,[3] which the Nvidia compiler does not support. Neither LLVM or Nvidia compiler can individually disable the argument reduction discussed in Section 6.5. Using the comprehensive fast-math flag (-ffast-math for LLVM or -use_fast_math for Nvidia compiler) does evidently disable argument reduction for both compilers, but the resulting accuracy also changes even for small input values.

[3] https://llvm.org/docs/LangRef.html#fastmath

Our approach does not require programmer domain knowledge for optimization. We acknowledge that EC-driven GPU 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. Perhaps the most striking examples of this in our study were the removal of the boundary check in SimCoV and the disabling of huge argument reduction in oxDNA's sine function. Relaxing semantics is a limitation because test suites are often 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 6.7), the programmer can choose 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 live up to performance expectations, 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 better 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. In applications like oxDNA, humans would look first for high payoff optimizations and might never consider the hundreds of individual modifications, which each contributed a small improvement. 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 and oxDNA may generalize to other bioinformatics kernels and programs.

Beyond its contribution to automated optimization, GEVO's results are striking from the EC perspective. The optimized ADEPT program we analyzed in detail contained 1395 mutations, each of these is *neutral* with respect to the test cases. Most of the mutations are weak (contribute less than 1% performance improvement), but it is still remarkable that it is even possible to apply that many random mutations to a program that is only 1,700 instructions long and not break the program. We don't yet understand why GEVO produces so many neutral mutations. It was built on NSGA-II, and an area for future investigation is disentangling the effect of our EC algorithm from the properties of the LLVM-IR. Once we discarded the weak mutations, the remaining contain a large number of interacting edits, which is vastly more than what has been reported by any earlier EC work for software (one or two edits are much more typical). This could arise from several factors: basic properties of the LLVM-IR representation and the 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 way 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, and could even include space beyond the application, like in oxDNA where

GEVO modified Nvidia math library functions. This 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 improvements with such a high number of interacting edits shows how automated methods can discover complex modifications to the target program. There is significant variability across programs in terms of the success that GEVO had in improving runtime, although we found significant improvements in all three examples that we studied. We do not yet understand the source of this variability and leave that for future work. As expected, there is also variability in the performance gains that are found by GEVO in different runs on the same program. However, some programs such as ADEPT-V1 had much higher variability than the others, as shown in Figure 7 and explained in Section 5.3. Finally, reverse engineering the discovered optimizations is challenging and to a large extent remains a manual process. We presented a procedure that identifies a set of edits that interact with each other, but the final step of the process involves testing all their combinations to find epistatic subsets. This will not scale well beyond the roughly twenty edits we considered. How to reliably discover edit interactions and effectively analyze them remains unsolved and warrants future study.

8 Related Work

Code generation optimization has been actively investigated in the compiler community. This includes, but is not limited to, peephole methods [Bansal and Aiken, 2006], loop-unrolling using machine learning techniques [Leather et al., 2009], loop perforation [Sidiroglou-Douskos et al., 2011] auto-vectorization [Mendis et al., 2019], and profile-guided optimization [Pettis and Hansen, 1990]. Traditionally, most of these techniques are achieved through pattern matching to ensure that exact program semantics are preserved. More recently, the need for compilers to optimize domain-specific languages has become important. For example, Halide [Ragan-Kelley et al., 2013] targets image processing, XLA [TensorFlow, 2018] developed for TensorFlow [Abadi et al., 2016], Glow [Rotem et al., 2018] for PyTorch [Paszke et al., 2017], TVM [Chen et al., 2018] for MXnet [Chen et al., 2015], and so forth. Because GPUs provide unparalleled performance in this domain, all of these examples are capable of generating GPU kernels for acceleration and optimization. A major component of these frameworks is identifying efficient loop partitioning and unrolling patterns tailored for target hardware memory configuration to achieve better memory access locality. Domain-specific compilers can perform further optimizations when lowering neural-network operators onto machine-specific implementations using optimized libraries. However, all of these approaches are still primarily based on human-derived pattern-matching, although, in certain domains, search algorithms are used when the target problem is unrelated to program semantics. For example, Halide uses the genetic algorithm to search for improved pipeline scheduler decisions, which translates to determining the order of loop partitioning and unrolling.

Beyond these traditional compiler techniques, other methods are designed to be generic in the sense that they are agnostic about the particular application being optimized. There are three main approaches that have been used: program synthesis [Alur et al., 2013; Barthe et al., 2013; Buchwald et al., 2018; Gulwani et al., 2011; Jia et al. 2019; Manna and Waldinger, 1980], superoptimization [Churchill et al., 2017; Schkufza et al., 2013, 2014; Sharma et al., 2015], and EC [Koza, 1994; White et al., 2011]. One key difference among the branches is the validation method. Program synthesis and superoptimization typically use a SAT/SMT solver [Moura and Bjørner, 2008] to check the logical equivalence of program rewrites, while EC relies on test suites to encode the intended program specification. The trade-off is that the SAT/SMT approaches can guarantee exact program semantics but they do not scale well, while test-based methods sacrifice strict semantic equivalence for improved scalability. As a result, most earlier work in this domain applies only to programs of a limited length, usually under 200 lines of code.

Deep learning methods have recently been used to analyze programs as well, including neural-network based logical reasoning [Evans et al., 2018; Paliwal et al., 2020] and SAT solvers [Selsam et al., 2019; Si et al., 2019] and superoptimization [Bunel et al., 2017]. However, for optimizing parallel codes like GPU programs, EC may be more viable because logical reasoning about thread communications in an SAT solver requires deducing the entire parallel programming model in a logical form which is time-consuming and challenging.

Large language models (LLM) have emerged recently as a new tool for program synthesis. Although many language models were originally developed to solve natural language processing tasks such as translation, researchers have discovered that by increasing the degree of language model architecture with a larger, albeit huge, training dataset, they can unlock many interesting properties including natural language reasoning and interaction, and even logical reasoning. One of LLMs' capabilities is generating simple programs even though the LLM was not explicitly trained on programming languages [Brown et al., 2020]. Since then, using LLMs for program-related tasks has become an active research area. Jacob et al. explored and evaluated general program synthesis using Python docstrings as input for behavior specification [Austin et al., 2021]. Other examples include CodeBert [Feng et al., 2020], Codex [Chen et al., 2021], and many more [Ahmad et al., 2021; Clement et al., 2020; Wang et al., 2021] were trained specifically for programming and coding, eventually leading to the popular and well-known commercial application, Github Copilot [Microsoft, 2023].

The only related LLM work in program optimization to date is from Cummins et al. [2023], which uses Meta the LlaMa 2 model [Touvron et al., 2023] to learn how the LLVM compiler optimizes code in LLVM-IR format. The learned LLM model can predict the least amount of compiler flags needed to optimize a target program or even directly generate the optimized code without using a traditional compiler. It is unknown, however, whether the trained LLM can generate code or optimize code that is unseen in the training data.

EC is a popular approach for improving computer programs, e.g., to automatically repair bugs [Debroy and Wong, 2010; Forrest et al., 2009; Le Goues et al. 2011, 2012; Weimer et al., 2009]. Surprisingly, prior analysis [Schulte et al., 2014b] 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 [White et al., 2011] and energy cost [Brownlee et al., 2021; Bruce et al., 2015, 2018; Schulte et al., 2014a].

Earlier EC work targeting GPU programs dates back to Sitthi-Amorn's work [Sitthi-Amorn et al., 2011], 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 [Langdon and Harman, 2010] to RNA and DNA analysis [Langdon and Harman, 2015; Langdon et al., 2015]. Specifically, BarraCUDA [Klus et al., 2012], a DNA sequence alignment program, was one of the target programs in the DNA analysis study [Langdon et al., 2015]. 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 approach 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 GEVO applied to three different types of bioinformatics workloads: ADEPT, a GPU-accelerated bioinformatics sequence alignment library; SIMCoV, an agent-based COVID simulation of viral spread; and oxDNA, a DNA model using molecular dynamic simulation. We find improvements between 17% and 29% for ADEPT-V1 (the expert-optimized version of ADEPT), SIMCoV, and oxDNA on various GPU platforms. Moreover, on ADEPT-V0, an earlier and less-optimized version, we find a 30X improvement. This demonstrates the excellent potential of stochastic search methods such as GEVO to augment developer efforts to optimize GPU codes.

While we did not find optimizations that are generalizable to all three applications, the diverse optimizations that GEVO discovered demonstrate its strength to tailor optimizations to particular applications based on their characteristics. The interdependent modifications (epistasis) found for ADEPT, the boundary check removal for SIMCoV, and the built-in math library optimizations for oxDNA are all distinct, unanticipated, and more importantly, challenging to achieve by the application developers alone. As GPU architectures continue to evolve, the availability of an automated code optimization tool that can 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 reducing the developer burden of developing efficient code, especially for application areas such as bioinformatics and other scientific domains.

Acknowledgments

We thank Lorenzo Rovigatti for helpful discussions.

References

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conf. on Operating Systems Design and Implementation.*

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* ACL, 2655–2668. Retrieved from https://www.aclweb.org/anthology/2021.naacl-main.211

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732

Muaaz G Awan, Jack Deslippe, Aydin Buluc, Oguz Selvitopi, Steven Hofmeyr, Leonid Oliker, and Katherine Yelick. 2020. ADEPT: A domain independent sequence alignment strategy for gpu architectures. *BMC Bioinformatics* 21, 1 (2020), 1–29.

Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. *SIGARCH Computer Architecture News* 34, 5 (2006), 394–403.

Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From relational verification to SIMD loop synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13).* ACM, New York, NY, 123–134. DOI: https://doi.org/10.1145/2442516.2442529

William Bateson. 1909. *Mendel's Principles of Heredity.* Cambridge University Press, Cambridge.

Sylvie Boldo, Marc Daumas, and Ren-Cang Li. 2008. Formally verified argument reduction with a fused multiply-add. *IEEE Transactions on Computers* 58, 8 (2008), 1139–1145.

Nicolas Brisebarre, David Defour, Peter Kornerup, J-M Muller, and Nathalie Revol. 2005. A new range-reduction algorithm. *IEEE Transactions on Computers* 54, 3 (2005), 331–339.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 1877–1901.

Alexander Brownlee, Jason Adair, Saemundur Haraldsson, and John Jabbo. 2021. Exploring the accuracy-energy trade-off in machine learning. In *Proceedings of the Genetic Improvement Workshop at 43rd International Conference on Software Engineering*. ACM, New York, NY.

Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing energy consumption using genetic improvement. In *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation*.

Bobby Ralph Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2019. Approximate oracles and synergy in software energy search spaces. *IEEE Transactions on Software Engineering* 45, 11 (2019), 1150–1169.

Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '18)*. ACM, New York, NY, 300–313. DOI: https://doi.org/10.1145/3168821

Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. 2017. Learning to superoptimize programs. In *Proceedings of the International Conference on Learning Representations*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274. Retrieved from https://arxiv.org/abs/1512.01274

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 578–594.

Berkeley Churchill, Rahul Sharma, J. F. Bastien, and Alex Aiken. 2017. Sound loop superoptimization for google native client. *SIGARCH Computer Architecture News* 45, 1 (2017), 313–326.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: Multi-mode translation of natural language and Python code with transformers. arXiv:2010.03150. Retrieved from https://arxiv.org/abs/2010.03150

Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. 2023. Large language models for compiler optimization. arXiv:2309.07062. Retrieved from https://arxiv.org/abs/2309.07062

Florent De Dinechin, Luc Forget, Jean-Michel Muller, and Yohann Uguen. 2019. Posits: The good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic*, 1–10.

Vidroha Debroy and W. Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of 3rd International Conference on Software Testing, Verification and Validation*.

Jonathan P. K. Doye, Thomas E. Ouldridge, Ard A. Louis, Flavio Romano, Petr Šulc, Christian Matek, Benedict E. K. Snodin, Lorenzo Rovigatti, John S. Schreck, Ryan M. Harrison, et al. 2013. Coarse-graining DNA for simulations of DNA nanotechnology. *Physical Chemistry Chemical Physics* 15, 47 (2013), 20395–20414.

Peter Eastman, Mark S. Friedrichs, John D. Chodera, Randall J. Radmer, Christopher M. Bruns, Joy P. Ku, Kyle A. Beauchamp, Thomas J. Lane, Lee-Ping Wang, Diwakar Shukla, et al. 2013. OpenMM 4: A reusable, extensible, hardware independent library for high performance molecular simulation. *Journal of Chemical Theory and Computation* 9, 1 (2013), 461–469.

Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. 2018. Can neural networks understand logical entailment?. In *Proceedings of the International Conference on Learning Representations*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*.

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of the Innovative Parallel Computing (InPar)*, 1–10. DOI: https://doi.org/10.1109/InPar.2012.6339595

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

Darrall Henderson. 2000. Elementary functions: Algorithms and implementation. *Mathematics and Computer Education* 34, 1 (2000), 94.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symp. on Operating Systems Principles (SOSP '19)*.

Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles S. H. Yeo, and Brian Y. H. Lam. 2012. BarraCUDA – A fast short read sequence aligner using graphics processing units. *BMC Research Notes* 5, 1 (2012), 1–7.

Matija Korpar and Mile Šikić. 2013. SW#–GPU-enabled exact alignments on genome scale. *Bioinformatics* 29, 19 (2013), 2494–2495.

John R. Koza. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing* 4, 2 (1994), 87–112.

Sudhir B. Kylasa, Hasan Metin Aktulga, and Ananth Y. Grama. 2014. PuReMD-GPU: A reactive molecular dynamics simulation package for GPUs. *The Journal of Computational Physics* 272 (2014), 343–359.

William B. Langdon and Mark Harman. 2010. Evolving a CUDA kernel from an nVidia template. In *Proceedings of IEEE Congress on Evolutionary Computation*.

William B. Langdon and Mark Harman. 2015. 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*.

William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA analysis software with genetic programming. In *Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation*.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, 75–86.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. 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*.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2011), 54–72.

Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 81–91.

Shin-Ying Lee and Carole-Jean Wu. 2014. Characterizing the latency hiding ability of GPUs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Jhe-Yu Liou, Muaaz Awan, Steven Hofmeyr, Stephanie Forrest, and Carole-Jean Wu. 2022. Understanding the power of evolutionary computation for GPU code optimization. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 185–198. DOI: https://doi.org/10.1109/IISWC55918.2022.00025

Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019a. Genetic Improvement of GPU Code. In *Proceedings of the IEEE/ACM International Workshop on Genetic Improvement (GI)*, 20–27. DOI: https://doi.org/10.1109/GI.2019.00014

Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019b. 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)*.

Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020a. GEVO: GPU code optimization using evolutionary computation. *ACM Transactions on Architecture and Code Optimization* 17, 4 (Nov. 2020), Article 33, 28 pages. DOI: https://doi.org/10.1145/3418055

Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020b. GEVO-ML: A proposal for optimizing ML code with evolutionary computation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*.

Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. 2012. CUSHAW: A CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform. *Bioinformatics* 28, 14 (2012), 1830–1837.

Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems* 2, 1 (1980), 90–121.

Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated end-to-end repair at scale. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. Compiler auto-vectorization with imitation learning. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 14598–14609.

Microsoft. 2023. Github Copilot. Retrieved from https://github.com/features/copilot

Melanie E. Moses, Steven Hofmeyr, Judy L. Cannon, Akil Andrews, Rebekah Gridley, Monica Hinga, Kirtus Leyba, Abigail Pribisova, Vanessa Surjadidjaja, Humayra Tasnim, et al. 2021. Spatially distributed infection increases viral load in a computational model of SARS-CoV-2 lung infection. *PLoS Computational Biology* 17, 12 (2021), e1009735.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.*

Dariusz Mrozek, Miłosz Brożek, and Bożena Małysiak-Mrozek. 2014. Parallel implementation of 3D protein structure similarity searches using a GPU and the CUDA. *Journal of molecular modeling* 20 (2014), 1–17.

NERSC. 2024. Cori GPU Nodes. Retrieved from https://docs-dev.nersc.gov/cgpu/hardware/

Kwok C. Ng. 1992. Argument Reduction for Huge Arguments: Good to the Last Bit. Unpublished draft, available from the author (kwok.ng@eng.sun.com).

NVIDIA. 2024. CUDA LLVM Compiler. Retrieved from https://developer.nvidia.com/cuda-llvm-compiler/

NVIDIA. 2024. GPU Boost. Retrieved from https://www.nvidia.com/en-us/geforce/technologies/gpu-boost/technology/

NVIDIA. 2024. NVIDIA 1080ti GPU. Retrieved from https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1080-ti/

NVIDIA. 2024. NVIDIA A100 Tensor Core GPU. Retrieved from https://www.nvidia.com/en-us/data-center/a100/

NVIDIA. 2024. NVIDIA Tesla P100 GPU. Retrieved from https://www.nvidia.com/en-us/data-center/tesla-p100/

NVIDIA. 2024. NVIDIA V100 Tensor Core GPU. Retrieved from https://www.nvidia.com/en-us/data-center/v100/

NVIDIA. 2017. Register Cache: Caching for Warp-Centric CUDA Programs. Retrieved from https://developer.nvidia.com/blog/register-cache-warp-cuda/

NVIDIA. 2018. Using CUDA Warp-Level Primitives. Retrieved from https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34, 2967–2974.

Bin Pang, Nan Zhao, Michela Becchi, Dmitry Korkin, and Chi-Ren Shyu. 2012. Accelerating large-scale protein structure alignments with graphics processing units. *BMC Research Notes* 5, 1 (2012), 1–11.

Chandra Shekhar Pareek, Rafal Smoczynski, and Andrzej Tretyn. 2011. Sequencing technologies and genome sequencing. *Journal of Applied Genetics* 52, 4 (2011), 413–435.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Proceedings of the NeurIPS Autodiff Workshop.*

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation.*

Erik Poppleton, Michael Matthies, Debesh Mandal, Flavio Romano, Petr Šulc, and Lorenzo Rovigatti. 2023. oxDNA: Coarse-grained simulations of nucleic acids made simple. *Journal of Open Source Software* 8, 81 (2023), 4693.

Erik Poppleton, Roger Romero, Aatmik Mallya, Lorenzo Rovigatti, and Petr Šulc. 2021. OxDNA.org: A public web-server for coarse-grained simulations of DNA and RNA nanostructures. *Nucleic Acids Research* 49, W1 (2021), W491–W498.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (Jun 2013), 519–530. DOI: https://doi.org/10.1145/2499370.2462176

Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. 2010. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics* 11, 3 (2010), 334–347.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. arXiv:1805.00907. Retrieved from https://arxiv.org/abs/1805.00907

Lorenzo Rovigatti, Petr Šulc, István Z. Reguly, and Flavio Romano. 2015. A comparison between parallelization approaches in molecular dynamics simulations on GPUs. *Journal of Computational Chemistry* 36, 1 (2015), 1–8.

Romelia Salomon-Ferrer, Andreas W Gotz, Duncan Poole, Scott Le Grand, and Ross C. Walker. 2013. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. *Journal of Chemical Theory and Computation* 9, 9 (2013), 3878–3888.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Proceedings of ACM SIGARCH Computer Architecture News.*

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* 49, 6 (2014), 53–64.

Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014a. Post-compiler software optimization for reducing energy. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.*

Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014b. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT solver from single-bit supervision. In *Proceedings of the International Conference on Learning Representations.*

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Xujie Si, Yuan Yang, Hanjun Dai, Mayur Naik, and Le Song. 2019. Learning a meta-solver for syntax-guided program synthesis. In *Proceedings of the International Conference on Learning Representations*.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering*.

Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic programming for shader simplification. In *Proceedings of the SIGGRAPH Asia Conference*.

Temple F. Smith, and Michael S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195–197.

Benedict E. K. Snodin, Ferdinando Randisi, Majid Mosayebi, Petr Šulc, John S. Schreck, Flavio Romano, Thomas E. Ouldridge, Roman Tsukanov, Eyal Nir, Ard A. Louis, et al. 2015. Introducing improved structural properties and salt dependence into a coarse-grained model of DNA. *The Journal of Chemical Physics* 142, 23 (2015), 06B613_1.

Alex D. Stivala, Peter J. Stuckey, and Anthony I. Wirth. 2010. Fast and accurate protein substructure searching with simulated annealing and GPUs. *BMC Bioinformatics* 11 (2010), 1–17.

Petr Šulc, Flavio Romano, Thomas E. Ouldridge, Jonathan P. K. Doye, and Ard A. Louis. 2014. A nucleotide-level coarse-grained model of RNA. *The Journal of Chemical Physics* 140, 23 (2014), 06B614_1.

TensorFlow. 2018. XLA Is a Compiler That Optimizes TensorFlow Computations. Retrieved from https://www.tensorflow.org/xla/

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288. Retrieved from https://arxiv.org/abs/2307.09288

Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358. DOI: https://doi.org/10.1016/j.future.2018.08.004

Paul Walsh and Conor Ryan. 1996. Paragen: a novel technique for the autoparallelisation of sequential programs using gp. In *Proceedings of the 1st Annual Conference on Genetic Programming*, 406–409.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*.

David R. White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation* 15, 4 (2011), 515–538.

Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated repair of java programs via multi-objective genetic programming. *Transactions on Software Engineering* 46, 10 (2020), 1040–1067.

Received 18 June 2023; revised 20 September 2024; accepted 30 October 2024