GEVO: GPU Code Optimization Using Evolutionary Computation

JHE-YU LIOU, Arizona State University
XIAODONG WANG, Facebook
STEPHANIE FORREST, Arizona State University and Santa Fe Institute
CAROLE-JEAN WU, Arizona State University and Facebook

GPUs are a key enabler of the revolution in machine learning and high-performance computing, functioning as de facto co-processors to accelerate large-scale computation. As the programming stack and tool support have matured, GPUs have also become accessible to programmers, who may lack detailed knowledge of the underlying architecture and fail to fully leverage the GPU's computation power. GEVO (Gpu optimization using EVOlutionary computation) is a tool for automatically discovering optimization opportunities and tuning the performance of GPU kernels in the LLVM representation. GEVO uses population-based search to find edits to GPU code compiled to LLVM-IR and improves performance on desired criteria while retaining required functionality. We demonstrate that GEVO improves the execution time of general-purpose GPU programs and machine learning (ML) models on NVIDIA Tesla P100. For the Rodinia benchmarks, GEVO improves GPU kernel runtime performance by an average of 49.48% and by as much as 412% over the fully compiler-optimized baseline. If kernel output accuracy is relaxed to tolerate up to 1% error, GEVO can find kernel variants that outperform the baseline by an average of 51.08%. For the ML workloads, GEVO achieves kernel performance improvement for SVM on the MNIST handwriting recognition (3.24×) and the a9a income prediction (2.93×) datasets with no loss of model accuracy. GEVO achieves 1.79× kernel performance improvement on image classification using ResNet18/CIFAR-10, with less than 1% model accuracy reduction.

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

Additional Key Words and Phrases: Genetic improvement, multi-objective evolutionary computation, GPU code optimization, approximate computing, LLVM intermediate representation

ACM Reference format:
Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO: GPU Code Optimization Using Evolutionary Computation. *ACM Trans. Archit. Code Optim.* 17, 4, Article 33 (November 2020), 28 pages.
https://doi.org/10.1145/3418055

This work is supported in part by the National Science Foundation under CCF-1618039 SHF-1652132, CCF-1908633; DARPA FA8750-19C-0003; AFRL FA8750-19-1-0501 for Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu at ASU.
Authors' addresses: J.-Y. Liou, Arizona State University, 1151 S. Forest Ave, Tempe, AZ 85287; email: jhe-yu.liou@asu.edu; X. Wang, Facebook, 1 Hacker Way, Menlo Park, CA 94025; email: xdwang@fb.com; S. Forrest, Arizona State University, 1151 S. Forest Ave, Tempe, AZ 85287 and Santa Fe Institute, 1399 Hyde Park Rd, Santa Fe, NM 87501; email: stephanie.forrest@asu.edu; C.-J. Wu, Arizona State University, 1151 S. Forest Ave, Tempe, AZ 85287 and Facebook, 1 Hacker Way, Menlo Park, CA 94025; email: carole-jean.wu@asu.edu.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).
1544-3566/2020/11-ART33
https://doi.org/10.1145/3418055

1 INTRODUCTION

The fields of large-scale machine learning and scientific algorithms are expanding quickly and pushing the limits of high-performance computing. Continued advances in these fields will require orders-of-magnitude improvements in computing power [28, 42, 105]. GPUs help address this challenge and have become de facto co-processors [2, 6, 38, 92, 113] for accelerating the performance of general-purpose, large-scale parallel workloads [5, 45, 83]. At the same time, the GPU programming interface has matured, making GPUs accessible to amateur programmers. However, it is challenging to optimize and fine-tune the performance of general-purpose GPU (GPGPU) programs without platform- and domain-specific knowledge. For example, programmers may be excessively cautious in their use of synchronization, which incurs a performance penalty. It is well known that the lack of concurrency semantics adds to the challenge of optimizing parallel programs such as GPU codes [10]. Relevant to our synchronization example, a compiler cannot eliminate unnecessary synchronization primitives without concurrency semantics. Figure 1 illustrates the size of this effect for one of the Rodinia benchmarks: Eliminating unneeded synchronization nearly doubles the performance gain over O2 optimization for nw.

Many research projects have investigated automated compilation optimization methods to reduce the burden on application programmers, including peephole [9], loop-unrolling using machine learning techniques [53], auto-vectorization [68], auto phase ordering [35], link-time [102], and profile-guided optimization [75], but additional efficiencies can be found by tailoring a binary to particular classes of inputs or particular architectures. For example, STOKE is a stochastic program synthesizer that uses random search to explore the high-dimensional space of possible program transformations and is an example of this performance tuning approach [85]. A related approach uses Evolutionary Computation (EC) to morph existing code from the target program to code that improves runtime and other non-functional program properties. In this work, we use EC, also known as Genetic Improvement, because earlier work showed that it scales well with program size [33, 89]. Some earlier work reports optimizations of specific GPU programs using EC [46–49], but these approaches are not easily extensible to large design spaces, in part because the representation of each program is customized and requires manual program translation before the search for optimizations can be performed.

We propose compiler post-pass performance tuning—**GEVO** (GPU code optimization using EVOlutionary computation) [56, 57]. GEVO encodes optimization objectives, such as execution time, energy use, or accuracy in its fitness function and implements a set of mutation and recombination operators for GPU kernel transformations in the LLVM-IR representation. We demonstrate GEVO first on the single-objective problem of reducing GPGPU kernel execution time (GEVO-default). Second, we show how GEVO can simultaneously tune GPGPU code to meet two independent objectives, such as performance and accuracy (GEVO-mO), using multi-objective search.

To assess the general applicability of GEVO, we evaluate on NVIDIA Tesla P100 using the Rodinia benchmark suite [75], which covers a set of important application domains such as medical imaging and data mining. Each application provides unique characteristics in terms of parallelism, data sharing, memory access patterns, and so forth. The results vary significantly across the Rodinia benchmarks, but on average GEVO-default improves GPGPU kernel runtime performance by 49.48% over the fully compiler-optimized baseline, and by as much as 412% in one case. If output accuracy is relaxed to tolerate up to 1% error, GEVO-mO can find kernel variants that outperform the baseline version by an average of 51.08%.

Although GEVO can be applied to any GPGPU program, we are particularly interested in machine learning programs, because machine learning tasks are computationally intensive and are by nature error-tolerant. For example, deep neural networks often require hours or days to train

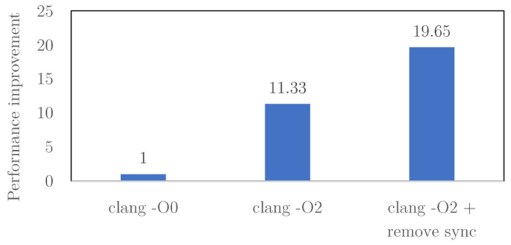

Fig. 1. Performance improvements for *nw* with no compiler optimization (clang -O0), with full compiler optimization (clang -O2), and when unneeded synchronization primitives are removed manually.

a single model, and training time is often traded off against model accuracy [38, 39, 80, 84, 114]. We present results for GEVO on supervised machine learning code from two production frameworks, ThunderSVM [107] and Caffe2 [31], considering standard handwriting recognition, income prediction, and image classification datasets.

To summarize, the key contributions of this article are:

- We present GEVO, a tool for automatically tuning the performance of GPGPU kernels represented in the LLVM intermediate representation (LLVM-IR) to meet multiple criteria. Our infrastructure scales to arbitrarily large program sizes. We demonstrate GEVO on the single objective of runtime optimization and on the multi-objective optimization criteria of runtime and accuracy.
- We evaluate GEVO on the Rodinia benchmark suite, which includes 13 applications covering a wide range of application domains. On average, GEVO improves kernel runtime by 49.48% with the output fidelity enforced or by 51.08% if the output fidelity can be relaxed by 1%.
- We evaluate GEVO on two machine learning kernels, using Thunder SVM and Caffe2, with standard machine learning benchmark datasets. In these experiments, model accuracy is interpreted as output fidelity. Compared to the original baseline, we find that GEVO can improve kernel runtime performance by 1.79× to 3.24×. In most cases, these runtime improvements are achieved without loss of accuracy, and in some cases model accuracy actually improves.
- In-depth analysis of GEVO optimizations identified several architectural-, domain-, and dataset-specific improvements. We provide explanations for many of the performance optimizations discovered by GEVO, such as eliminating conservative synchronization primitives (Section 5.2.1), removing redundant `store` instructions (Section 5.1.2), reducing conditional executions (Section 5.2.1), loop perforation (Section 5.2.3), memoization (Section 5.2.4), and algorithm improvements (Sections 6.1.2 and 6.2).
- Multi-objective optimization: We demonstrate that when output fidelity is relaxed, solutions can be found that improve both optimization criteria—runtime and output fidelity—simultaneously. These optimization points are not accessible to the search when output fidelity is strictly enforced.

The remainder of the article is organized as follows: Section 2 provides relevant background and places the article in the context of earlier work. Section 3 describes the GEVO design in detail; and Section 4 describes the system environment and benchmarks we used to evaluate GEVO. Experimental results for GEVO-default and GEVO-mO are reported in Section 5 and Section 6, respectively. We discuss limitations and future directions in Section 7 and conclude the article in Section 8.

2 RELATED WORK

This section discusses related work from program synthesis, superoptimization, and evolutionary computation. It also reviews the few earlier works that have applied EC to GPUs and the growing body of work using EC to improve machine learning.

2.1 Program Synthesis and Superoptimization

Program synthesis methods automatically generate computer programs based on programmer-defined specifications. In some cases, the goal is to produce programs that run faster, which is known as superoptimization. Early work on superoptimization dates back to Massalin's superoptimizer [64] in 1987, which exhaustively searched and tested a subset of the Motorola 68020 assembly instruction set against testing inputs and synthesized the shortest instruction sequence for a target function.

Since the number of possible code sequences in most programming languages is enormous, finding an optimal sequence is usually intractable, and all recent program synthesis techniques use some form of search to cut down the search space. Such search algorithms can be roughly divided into two categories: deductive synthesis and inductive synthesis. Deductive synthesis encodes a given program into a Boolean formula and searches for a logically equivalent formula using a SAT/SMT solver. Developing such encodings is challenging and time-consuming, and significant research effort has been devoted to making such processes more accessible to programmers, ranging from Rosette, a symbolic framework language interpreter [100, 101], to Z3, an efficient SMT solver with C and Python binding [72]. Synthesized programs using the deductive approach are provably correct and do not require verification. However, as the length of the computer program increases linearly, the size of the corresponding Boolean expressions grows exponentially. Therefore, programs that are synthesized using the deductive approach are relatively small and often have additional constraints. For example, the largest loop-free program synthesized by Gulwani in 2011 [34] has only 16 lines in the pseudo assembly.

Instead of representing the program's specification as a Boolean formula, inductive synthesis relies on the original program and a set of input/output examples (test cases). In this case, the search usually begins by sampling local deviations from the given program or by generating random code combinations from scratch. Each variant is then checked against the test cases to verify functionality. Depending on the particular search method, new code recombinations may be tried based on previous observations. These methods use stochastic search, such as Markov Chain Monte Carlo (MCMC) sampling or Evolutionary Computation (EC). Schkufza et al. proposed STOKE [24, 85, 86, 93] using MCMC sampling for X86-64 assembly codes to improve runtime. STOKE has the same overarching goal as GEVO, which is to search for program optimizations without guaranteeing exact program semantics. However, STOKE does not naturally scale up to large code sizes, because it considers the entire X86-64 instruction set, even vector instructions, and it focuses on synthesizing instruction sequences from scratch. GEVO differs from STOKE in two ways: It uses EC to modify an existing program using existing instructions, and it can scale to arbitrarily large program sizes (over 100,000 instructions [89]), whereas the test programs in STOKE are only a few hundred instructions [24].

We chose EC primarily because of its scalability to realistic program sizes. Beyond scalability, an inductive method was appropriate, because many applications on parallel accelerators, e.g., image processing and machine learning, do not require an exact result. Instead, these applications often have domain-specific metrics for assessing acceptability. This feature of many GPGPU programs mitigates the requirement to preserve exact program semantics.

2.2 Evolutionary Computation (EC)

Earlier work developed EC methods to improve computer programs, e.g., to automatically repair bugs in legacy software [27, 32, 33, 52, 106], and this class of applications is sometimes referred to as *genetic improvement*. Transitions to industrial practice include Facebook's SapFix tool [30] and the Janus Manager deployment [37]. Although most work has been conducted at the source-code level using abstract syntax trees, similar methods have been applied to assembly programs [88] and object code [91].

Subsequent analysis showed that the applied mutations often have no observable effect on program behavior [11, 15, 36, 90, 103]. These *neutral* mutations occur frequently (20%–40% of the time), even when the mutations are restricted to sections of code covered by the tests. Although it is surprising that the rate of neutral mutations is so high, equivalent mutations are well-known in mutation testing, e.g., Reference [62]. These results suggested the possibility of using EC to optimize non-functional properties of software by finding modifications that are neutral with respect to the test suite but improve the non-functional property in question.

White et al. proposed the idea of using EC to improve program performance [108], and Schulte et al. achieved significant energy reductions for several Parsec benchmarks [89]. Bruce et al. applied a similar technique for MiniSAT to reduce energy consumption [15]. Other works [18, 63] constrain EC's search space for improved energy consumption of Java programs by asking users to provide predefined locations or equivalent functions or class implementations. These, and several subsequent works [16, 29], demonstrate the potential for stochastic search methods such as EC to find machine- or architecture-specific optimizations that improve a program's performance or energy efficiency. However, these methods are not yet mature or carefully analyzed. In contrast with our work, they focus on energy reduction rather than runtime, typically targeting the CPU; and their general applicability is not well understood. The results reported here address these limitations.

2.3 Evolutionary Computation for GPGPU Programs

There is some previous work applying EC to GPU kernels, including a graphics shader program [95]. This work 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-specific experts. In the GPGPU domain, Langdon et al. used EC to reduce the runtime of a CUDA program, reporting results for two specific programs: `gzip` [46] and an RNA analysis program [49]. This prior work targets a single program operating in a specific domain, and the methodology used in References [46–49] represents the program object as a custom-designed, line-based Backus Normal Form (BNF) grammar. We seek a method that is generalizable across multiple programs with minimal manual intervention and uses modern tooling, which is Clang/LLVM.

Clang/LLVM is a widely-used, multi-language, and highly modular compiler infrastructure. Schulte's thesis is the only work we are aware of that has experimented with evolving the LLVM intermediate representation (LLVM-IR) [87], but this was a preliminary proof-of-concept rather than a robust implementation, and no significant experiments were conducted. Now that Clang/LLVM supports CUDA compilation, it is feasible to compile GPGPU kernels into LLVM-IR, but this has only been available since 2016 [109]. Thus, we adopt the Clang/LLVM infrastructure for GEVO including the LLVM-IR, as shown in Figure 2. This avoids developing novel parsing and syntax manipulating infrastructure, but it introduces new challenges for implementing the basic mutation and recombination operations.

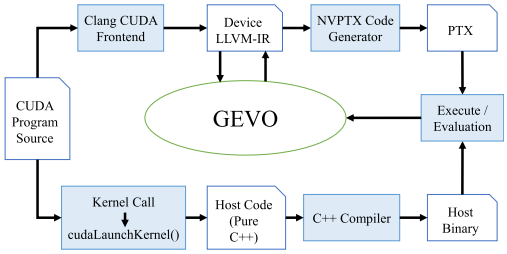

Fig. 2. GEVO in the LLVM/Clang compilation flow.

2.4 Genetic Improvement of Machine Learning (ML)

While EC can be applied to any computer program, machine learning is a particularly attractive domain, because of its popularity and its high computational cost. Moreover, most ML applications can be accelerated using GPUs. Although no prior work is known that uses EC to accelerate ML programs on GPUs, EC has often been used to improve neural network designs and to optimize weights. This work dates back at least to an 1989 [71] paper that used EC to train a neural network. The most established and popular approach in this domain is NEAT [97], proposed by Stanley et al. in 2002, which uses EC to simultaneously learn the neural network connection topology, and the weight of each neuron. The original NEAT design performed well in a comparatively small and homogeneous neural network, and the following works expanded its scalability to larger networks and more complex tasks [79, 96, 104].

More recently, convolution neural networks (CNNs) have achieved extraordinary performance in image classification tasks by adding convolution layers as filters. These layers are used to determine important spatial patterns in the image so the number of features can be reduced before being fed into a traditional neural network. Many approaches for identifying good architectures (topologies) for CNNs have been proposed [13, 43, 58, 59, 61, 76, 111, 116], and these have outperformed manually designed architectures on several tasks. Similar to NEAT, Reak et al. proposed using EC to design CNNs in a limited search space of convolution layers composed of common arithmetic operations [78]. This work achieves state-of-the-art classification performance on the ImageNet dataset compared to other network architecture searches, which use random search and reinforcement learning.

In state-of-the-art machine learning programming frameworks, deep learning models are represented as computational graphs of various types of operators. This exposes opportunities for operator-level optimization, such as operator fusion, using domain-specific compilers. For example, XLA [1] is developed for TensorFlow [3], Glow [82] for PyTorch [74], TVM [22] for MXnet [21], and so forth. Domain-specific compilers can perform further optimization when lowering high-level, neural network operators onto machine-specific implementations using optimized libraries. These optimizations differ from the aforementioned neural architecture searches in that the functional behavior of a given network is preserved. Built on top of the prior superoptimization approach [64], TASO [41] was recently proposed to automatically optimize the computational graph. Given a small computational graph, TASO enumerates all combinations of operator implementations and selects the operator graph implementation that minimizes runtime. The functional behavior of the original graph is preserved with satisfiability verification using a SAT solver. TASO has the same scalability limitations discussed earlier for deductive program synthesis (Section 2.1), and the referenced work does not scale beyond graphs of size 4 operators.

Our approach to optimizing NNs is complementary to this earlier work. GEVO explores joint optimization opportunities by (1) discovering better-performing operator implementations and (2) changing neural network architectures, which extends previous work (Section 6.2.2).

3 GEVO DESIGN

We propose *GEVO*—a tool for automatically improving kernel implementations for GPUs. GEVO enables GPU code optimization using EVOlutionary computation. It is a post-pass performance tuning approach to optimizing GPGPU kernel implementations.

GEVO takes as input a GPGPU program, user-defined test cases that specify required functionality, and a fitness function to be optimized. GEVO attempts to maximize the fitness function by evolving and evaluating mutated kernel variants in an iterative population-based search. Figure 2 presents GEVO in the context of the LLVM/Clang compilation flow. Kernels in a GPGPU program that will run on GPU are first separated and compiled into the LLVM-IR using the Clang compiler. GEVO takes these kernels in LLVM-IR format as inputs, modifies them to produce different kernel variants, and translates the variants into PTX files. The host code running on the CPU is modified to load the generated PTX file into the GPU. GEVO then evaluates how the kernel variant performs as defined by the objectives encoded in the fitness function.

As shown in Algorithm 1, the search begins with an initial population of PopSize individuals (LLVM-IR kernel variants) that is formed by taking the original program, making PopSize copies, and applying random mutations to each (Line 3 where InitDist, the number of mutations applied to each individual, defaults to 3), giving the initial population some diversity. GEVO then forms the next generation of individuals by ranking individuals according to the objectives, recombining instructions between kernel variants (*Crossover*), and randomly adding, deleting, or moving instructions in each variant (*Mutation*). Finally, GEVO compares the new variants to a set of elites retained from the previous generation (*Selection*), eliminating low-fitness individuals and retaining those with higher fitness for the next generation. The next few subsections give details of how we implemented these operations for GPGPU optimization.

ALGORITHM 1: The main loop of GEVO.

```
    Parameter: PopSize, CrossRate, MutateRate, InitDist
    Input: GPGPU kernel Program, P
 1: pop ← Initialize(PopSize, P)
 2: for all individual in pop do
 3:     Mutate(individual)*InitDist
 4: rank ← NonDominatedSort(pop)
 5: while not interrupt do
 6:     offspring ← SelTournament(pop, rank, PopSize)
 7:     elites ← SelBest(pop, rank, PopSize /4)
 8:
 9:     for every 2 individual (ind1, ind2) in offspring do
10:         if random() < CrossRate then
11:             Crossover(ind1, ind2)
12:     for all individual in offspring do
13:         if random() < MutateRate then
14:             Mutate(individual)
15:
16:     rank ← NonDominatedSort(elites + offspring)
17:     pop ← SelBest(elites + offspring, rank, PopSize)
```

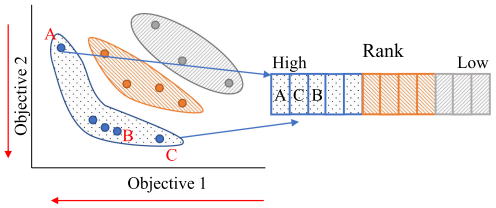

Fig. 3. Non-dominated sorting.

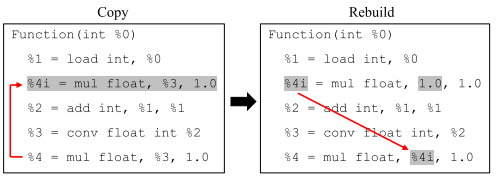

Fig. 4. Mutate-Copy: Operand dependency is rebuilt to preserve LLVM-IR program validity. Since LLVM-IR is strongly typed, the constant value 1.0 is used if no other value in the requested type is available.

3.1 Individual Representation

GI methods typically use either a program-based (each variant consists of the entire program) or a patch-based (each variant is a list of mutations (edits) applied to the original program) representation. For large programs, the patch-based representation is convenient, because it is more space-efficient. GEVO uses both representations. That is, each individual kernel variant contains both the LLVM-IR code and the set of the mutations that produced it from the original.

This design decision relates to the many data dependencies built into the LLVM-IR. Because of the repair process that is required after most mutations, it would be expensive to reapply all the mutations for a variant each time it is evaluated. Similarly, crossover exchanges subsections of the kernel code. Doing this naively can break a large number of data dependencies, so it is more efficient to implement crossover using the patch representation. Because the number of mutations applied to any kernel variant tends to be low, and because kernels are relatively small-sized programs, the memory requirement of storing both representations is reasonable.

3.2 Fitness Evaluation

Although GEVO can optimize any desired fitness function, we first focus on the problem of reducing kernel execution time (*GEVO-default*). In this case, the fitness function is simply the runtime of the kernel variant with the requirement that it produce identical output as the original program. When we consider approximate computing, where output accuracy can be relaxed to improved execution time, we include output accuracy in the fitness function as multi-objective (*GEVO-mO*), i.e., $argmin(time, error)$.

Using these fitness criteria each kernel variant is evaluated by running it against all available test cases. To protect against overfitting, we also evaluate at the end of the search against a set of held-out test cases, generated randomly. The fitness value is reported as a vector corresponding to the number of objectives. Each element in the vector is a single scalar value, i.e., the mean performance on that objective across the test cases (see Section 4.3).

3.3 Selection

GEVO uses the Non-dominated Sorting Genetic Algorithm (NSGA-II) [26] to select individuals according to multi-objective fitness criteria. Figure 3 illustrates a set of kernel variants, plotted according to two dimensions of the fitness function, say, error and runtime, where the goal is to minimize both objectives, retaining individuals that represent the best tradeoffs between the two objectives (shown in blue in the figure). NSGA-II uses Pareto dominance, where individual i is said to dominate individual j if i is better than j on at least one objective and no worse on the others.

NSGA-II calculates the Pareto fronts and ranks individuals according to which front they belong. Then a crowding factor is calculated for each individual based on the density of other individuals along its Pareto front, and these two values are combined to produce a single fitness value for each kernel variant (see Reference [26] for details). Finally, based on this single fitness value, NSGA-II uses a popular EC selection method known as *tournament selection* [69] to choose kernel variants for the next generation (Line 6 of Algorithm 1). GEVO retains the top quarter of the population at time t and copies it unchanged to the population at time $t + 1$ (Lines 7, 16, and 17 of Algorithm 1), a method known as *elitism* [8]. It then chooses the remaining 3/4 of the population using the tournament selection.

3.4 Mutation Operators

Mutation modifies the linear array of instructions stored for each variant using one of the following operations:

- **Mutate-Copy:** Duplicate an instruction and insert it in another location.
- **Mutate-Delete:** Remove an instruction.
- **Mutate-Move:** Move an instruction to a different location.
- **Mutate-Replace:** Replace an instruction with another instruction. This can occur either at the instruction or the operand level. In the second case, a single operand is replaced with another operand.
- **Mutate-Swap:** Swap the location of two instructions.

Since the LLVM-IR is based on Static Single Assignment (SSA) where each variable (like `%0`, `%1` in Figure 4) can be assigned only once at creation, our mutations are likely to create invalid programs by breaking the SSA constraint. Thus, we introduce an extra repair step. As illustrated in Figure 4, the instruction `mul` is copied, and we see that the first operand relies on `%3`, which is invalid in the new location. GEVO repairs it with the constant 1.0, as the two existing values (`%0, %1`) are not of the proper type. To our knowledge, only one other work [87] has attempted to design mutation operations for SSA. GEVO implements similar mutations to this earlier work, although our mutations repair SSA dependencies more robustly.

The operators Mutate-Copy and Mutate-Move insert new instructions, which has no effect unless a subsequent instruction can use the result of the inserted instruction. Figure 4 illustrates how GEVO enforces this by changing the first operand of the fifth instruction to use the value from second instruction. This instruction was selected because its types agree with the mutated instruction. In addition to the type checking shown in the above repair procedure, GEVO uses dominator analysis to eliminate values if the creator and the user of a value are in separated basic blocks that do not share the same execution path.

As depicted in Line 14 of Algorithm 1, when mutation is called, one of the aforementioned mutation operations is selected randomly (with equal probability) and applied as an edit to generate a new kernel variant. Since GEVO does not use domain-specific knowledge to select a mutation or rely on program semantics, we immediately evaluate the individual by running the available test cases, as Section 3.2 describes, and eliminate invalid modifications (that do not pass all tests). Mutation is iteratively applied to the same individual until a valid kernel variant is identified. Depending on the kernels, the acceptance rate of any single mutation is typically 5%–30%.

3.5 Crossover

GEVO uses the patch-based representation for crossover, because combining two random program slices would require more extensive repair. GEVO implements one-point messy crossover, which combines shuffle [17] and variable-length [54] crossover operations. Figure 5 illustrates the

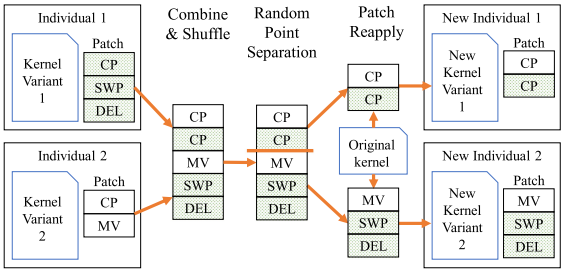

Fig. 5. One-point messy crossover.

process. Beginning with a list of mutations (edits) associated with each individual, GEVO combines them into a single sequence, shuffles the sequence, and randomly separates it back into two distinct patch sequences. Finally, GEVO reapplies each patch sequence to the original GPGPU kernel and generates two new individuals. This form of crossover was selected, because it generates a wide diversity of recombinations from a minimal number of mutations, and our mutations are relatively expensive.

Similar to mutation, after crossover, each new individual is evaluated to test if the combinations are valid. Otherwise, GEVO repeats the process until it finds a successful recombination. The acceptance rate of crossover is as high as 80%, because each individual mutation has already been validated.

4 EXPERIMENTAL SETUP

This section describes our experimental setup for the empirical evaluation on real GPUs.

4.1 Infrastructure and Configurations

We developed GEVO using an existing Python framework for evolutionary algorithms, called DEAP [25], implementing the genetic operators described in Section 3, and integrating them into the DEAP framework.[1] We instrumented the LLVM compiler (LLVM 8.0) to implement the mutation operations in C++ as an LLVM pass. We evaluate GEVO using two machines of the same configuration. Each machine has an Intel Xeon E5-2640 CPU with 40 cores, 256 GB system memory, and an NVIDIA Tesla P100 GPU with 16 GB GPU memory. The machine is installed with Ubuntu 16.04 with NVIDIA CUDA 10.1 and NVIDIA GPU driver 418.87. For each optimization, GEVO was given a 48-hour wall-clock budget on one machine. The Nvidia profiler (nvprof) was used to collect kernel execution time, which became the runtime metric used by the fitness function. Although nvprof does increase overall application execution time, our optimization target is kernel execution time, and in our experiments nvprof introduced no overhead to kernel execution time. The collected kernel execution time consistently varied less than 1% over multiple profiling runs on the same GPGPU kernel.

All GEVO experiments were conducted with population size of 256, crossover rate of 80% (80% of individuals in population are selected for crossover), and a mutation rate of 30% (every individual has 30% chance of receiving one mutation). The 48-hour budget for each GEVO run translates into a variable number of generations (shown in the last column of Table 1), depending on the program, the test cases, and the success rate of the mutation operation. For our experiments, the number of generations ranged from a low of 12 to over 80. For example, for the NN benchmark, GEVO spent the majority of its time searching for valid mutations and was able to complete only 18 generations

[1]The GEVO code is available at https://github.com/lioujheyu/cuda_evolve.

Table 1. Benchmarks Used for Evaluation

Application	Abbr.	GPGPU Kernel Lines of LLVM-IR	Generations
Breadth first Search	bfs	72	18
B+Tree	b+t	168	63
CFD Euler3D	cfd	1,079	53
Gaussian elimination	gau	186	29
Heart Wall	hw	3,806	36
Hotspot	hot	189	28
LU decomposition	lud	2,491	81
Nearest Neighbor	nn	32	18
Needleman-Wunsch	nw	715	21
Particlefilter	pf	1,442	55
Pathfinder	path	109	25
SRAD_v2	sv2	446	16
Stream Cluster	sc	231	12
Handwriting recognition (C=5, g=0.05)	mnist	(c_smo_solve) 256	27
Income prediction (C=32, g=0.0078125)	a9a	(c_smo_solve) 256	61
Image classfication	cifar-10	(momentumSGD) 39	15

within 48 hours. We speculate that in cases where GEVO was unable to find useful optimizations it is partially because the runs did not complete enough iterations of the search, and providing a larger search budget could improve results for these programs.

4.2 Benchmarks

Table 1 summarizes the benchmarks we used to evaluate GEVO: the Rodinia benchmark suite and ML workloads on ThunderSVM and Caffe2.

For the first set of benchmarks, we validated optimized kernel variants using the default inputs provided with the Rodinia benchmarks. For each benchmark, we then generate additional tests by randomly generating three sets of input values using the Rodinia built-in input generator. Depending on the benchmark, each input set contains from tens of thousands to millions of input values. Each test was run with the original, unmodified kernel and its output was used as an oracle to validate the output of the candidate kernel variants. GEVO rejects variants that fail to produce outputs that are identical to the oracle (GEVO-default) or exceed the default 1% error tolerance (GEVO-mO). After evolution, we validate the highest fitness kernel variant found during the search on held-out tests. We generate the held-out tests by rerunning the test-generation process. This step helps ensure that GEVO does not overfit the kernel to the existing test cases during the evolution.

For the ML benchmarks, we focused on a Support Vector Machine (SVM) and Stochastic Gradient Descent. Because GEVO searches the optimization space at the granularity of instructions, it requires full access to the intermediate representation and the corresponding optimized library implementations. This consideration led us to a supervised ML framework, ThunderSVM, which is a support vector machine library that is fully open-sourced and optimized for GPU implementation.

We evaluated GEVO on ThunderSVM (referred to as SVM) with two standard datasets: handwriting recognition using MNIST [51] and income prediction using a9a [77]. We downloaded the datasets from libsvm [20]'s data repository, which consists of 60,000 training and 10,000 testing

data samples for MNIST, and 32,561 training and 16,281 testing samples for a9a. Additionally, we used the MNIST large dataset, which contains 8M image samples, solely for the post-evolution evaluation. Specifically, we asked GEVO to optimize the `c_smo_solve` kernel for both training time and inference prediction accuracy of the trained model. We present the results in Section 6.2.1.

Many popular deep learning frameworks, such as TensorFlow [3], PyTorch [74], and Caffe2 [31], rely heavily on the NVIDIA closed source cuDNN library to drive the GPU and cannot be directly targeted by GEVO. However, a few frameworks maintain a small custom CUDA implementation when required functionality has no direct mapping from the cuDNN library. For example, in Caffe2, stochastic gradient decent with momentum (momentumSGD) is custom implemented as a CUDA kernel and open sourced within Caffe2 source repository. This provides an opportunity for us to include Caffe2 in our evaluation.

We evaluated Caffe2 using an 18-layer residual neural network (referred as ResNet18) to perform image classification against the CIFAR-10 dataset [44] with 50,000 training and 10,000 testing images. We used GEVO to optimize the momentumSGD kernel, because it is the only major operator used in ResNet that is available to us through open source. This kernel updates the weight of the neural network based on the loss function, evaluating the difference between the true label and predicted label. Since the search space is constrained to a single operator, we include this application to demonstrate GEVO's capability and do not attempt a complete solution for image classification. The results are given in Section 6.2.2.

4.3 Error Metric

For the Rodinia benchmarks, error represents the maximum difference between the output produced by the unmodified, original kernel implementation and that of GEVO-mO, across all tested inputs. Kernel variants are eliminated if the error rate of any test case exceeds the prespecified threshold, i.e., 1%.

For both SVM and ResNet18, we consider the runtime to train the model and the accuracy of the trained model's performance. For SVM, we use two-fold cross validation on the training dataset to report the error to GEVO during optimization. In ResNet18, we also report the training error for GEVO when the model is trained for three epochs, which shortens the training time to one minute. However, even with this simplification, GEVO required seven days to search for 20 generations, which is a low number of generations for an evolutionary search. The testing dataset, regardless of which framework and application, was used only for measuring the testing error and was never presented to GEVO.

Similar to Rodinia, the ML kernel variants are rejected if the training error exceeds the error achieved from the original kernel by the 1% threshold. For example, if the unmodified kernel achieves 3% training error, then a GEVO kernel variant with 4.1% training error will be rejected and another one with 3.9% training error will be accepted. In ResNet18, we set the threshold to 10%, because we trained for only three epochs, which is generally not long enough for the model to converge. Thus, the training error is noisier with this training regime, so we allowed a more generous error tolerance.

5 EVALUATION OF GEVO-DEFAULT

We first present the results of our experimental evaluation on the Rodinia benchmark suite to evaluate GEVO's applicability across a variety of programs (GEVO-default). Next, we investigate the most common *architecture-specific* and *application-specific* optimizations that GEVO discovered (Section 5.1). In Section 6.2, we consider *dataset-specific* optimizations.

Figure 6 reports the overall performance improvement for GEVO-default by comparing to the default baseline with full compilation optimization for the Rodinia benchmarks. GEVO-default

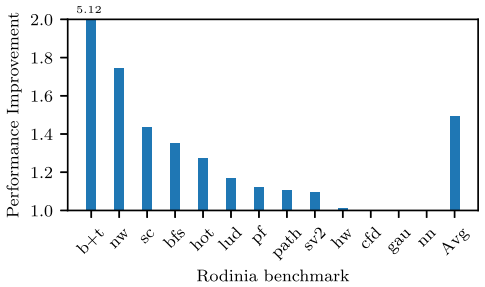

Fig. 6. Normalized performance improvement over the default baseline with full compilation optimization for GEVO-default in the Rodinia Benchmark. (For example, the 1.27× improvement in hot reduces runtime from 1.07 seconds to 1.07/1.27 = 0.84 seconds.)

improves the performance of the Rodinia benchmark suite by an average of 49.48% and by as much as 412% for `b+t`. As Figure 6 shows, there is significant variability in the achieved improvement among programs, including three, `cfd`, `gau`, and `nn`, for which we found no optimizations. There are several possible explanations for this variability. It could be a feature of the program itself, it could be that we did not let GEVO search long enough, or perhaps the program was perfectly optimized by the original programmer. However, this variability is consistent with results reported using evolutionary methods on the related problem of reducing energy use by assembly programs [89].

5.1 Architecture-specific Optimizations

GEVO discovered several different optimizations related to GPU architecture design in the evolved Rodinia Benchmarks, including synchronization issues and memory order issues. Some of these optimizations arise from a combined effect of the architecture and the particular application/algorithm.

5.1.1 Eliminating Synchronization Primitives. One of the most common GEVO optimizations removed synchronization primitives, specifically `syncthread()`, calls in CUDA. For example, when a programmer wants to exchange data between threads in a thread block through the shared or global memory, synchronization primitives are used to synchronize the progress of GPU threads. There are multiple reasons why a `syncthread()` call might not be required and could be removed without damaging the application. We give several examples, taken from GEVO runs on the Rodinia benchmarks.

nw: Figure 7 shows a simplified code snippet taken from `nw`. The `syncthread` function is used three times in this particular kernel (Lines 6, 8, and 10). The first two `syncthread` calls (Lines 6 and 8) can be eliminated, because neither `ref` nor `temp` are read before a new value is written into the same memory location. It appears that the programmer was unnecessarily conservative in this case, which increased performance cost without additional semantic value. Most of the performance improvements discovered by GEVO for `nw` eliminated overly conservative uses of `syncthread`. Such optimizations are, of course, somewhat risky in general. However, they illustrate the value of tailoring a kernel for exactly the workloads it will experience. As part of a wide deployment, additional post hoc methods, such as test-case generation or program analysis, could be employed to double-check that specific optimizations are indeed safe under the relevant use cases.

lud: Other synchronization-related optimizations found by GEVO are architecture-specific and concern scope. In a GPGPU application, massive numbers of parallel threads are created to

```
__shared__  int temp[...][...];
__shared__  int ref[...];
int tid = threadId.x;

ref[tid] = referrence[...];
__syncthreads();
temp[tid+1][0] = matrix_cuda[...];
__syncthreads();
temp[0][tid+1] = matrix_cuda[...];
__syncthreads();

for (int i=0; i<BLOCK_SIZE; i++)
  temp[tid][tid] =
    temp[i][0] + temp[0][i] + ref[i];
```

Fig. 7. Simplified code snippet from nw with conservative syncthread() calls.

(a) Unmodified

```
__shared__ s[BLOCK];
int c = CONST;
int tid = ThreadId.x;
for(i=0; i < 16; i++)
  s[tid] = init(tid);
__syncthread();


for(i=0; i < 16; i++)
  s[tid] = s[tid] - s[i]*s[i];

s[tid] = s[tid] / c;
__syncthread();
```

(b) Post-Compilation

```
__shared__ s[BLOCK];
int c = CONST;
int tid = ThreadId.x;
for(i=0; i < 16; i++)
  s[tid] = init(tid);
__syncthread();

float temp = s[tid];
for(i=0; i < 16; i++) {
  temp = temp - s[i]*s[i];
  s[tid] = temp; }
s[tid] = temp / c;
__syncthread();
```

(c) GEVO Optimized

```
__shared__ s[BLOCK];
int c = CONST;
int tid = ThreadId.x;
for(i=0; i < 16; i++)
  s[tid] = init(tid);
__syncthread();

float temp = s[tid];
for(i=0; i < 16; i++)
  temp = temp - s[i]*s[i];
  s[tid] = temp;
s[tid] = temp / c;
__syncthread();
```

Fig. 8. Code snippet from lud illustrates a GEVO optimization, which removed redundant store instructions. (a) is the original implementation, (b) is the LLVM/Clang compiled version, and (c) is the optimized kernel variant with GEVO.

execute the same piece of code in a kernel. At runtime, multiple threads form a *thread-block* or a *cooperative thread array*. A thread-block is the basic unit of execution—all threads within a thread-block have the same life-cycle and are dispatched onto a GPU streaming multiprocessor at the same time. Depending on the width of the vector functional units/SIMD lanes, threads within a thread-block are grouped into small batches (typically 32 or 64 threads), called a warp or a wavefront. At every cycle, the GPU hardware warp scheduler selects a ready warp from the warp pool for execution (known as the SIMT execution). Warps within a thread-block and threads within a warp are tightly coordinated, bounded by the same synchronization barrier (architecture-specific). To explicitly synchronize threads within a thread-block, syncthread can be used. In the case of lud, the programmer specified exactly the same number of threads in a thread-block as the warp thread size. GEVO finds a kernel variant that leverages the implicit synchronization boundary implemented at the warp granularity and eliminates the syncthread call.

All: Depending on the specific implementation of warp and thread-block scheduling policies, additional syncthread calls can often be eliminated without altering the execution order between GPU threads (hotspot, lud, nw).

5.1.2 Removing Redundant Store. Another optimized kernel variant discovered by GEVO removed redundant store instructions, illustrated with a code snippet from lud in Figure 8. We show three versions of the code snippet: (a) the original implementation, (b) the LLVM-IR after compilation with the Clang compiler, and (c) the optimized kernel variant found by GEVO.

Variable `s` is stored in the GPU shared memory (Line 1, Figure 8(a)) and is initialized in Line 5. Each parallel thread accumulates its own `s` and reads/writes to `s` in the shared memory directly (Lines 9–10). Finally, the value of `s` is written back to the shared memory (Line 12). Compared to updating values in the GPU register file, updating `s` in the shared memory (Line 10) can incur significant latency and stress the memory subsystem unnecessarily.

Instead of constantly reading and writing the value of `s` to the shared memory, the post-compiled LLVM-IR version eliminates the variable reuse patterns for `s` and replaces `s` with a temporary variable (`temp`) that is stored in the register file. To preserve semantic correctness, `s` in the shared memory is updated with the new value of `temp` (Line 11). Interestingly, this instruction is removed by GEVO. While in theory other threads could access `s` and receive a stale value, this does not change kernel outputs. GEVO, in this case, trades off program semantics for improved kernel execution time. We conjecture that such optimization opportunities, although not completely generalizable, are well-suited for GPUs. This is because GPUs implement more relaxed memory models. General-purpose programs are inherently more sequential and might be less likely to benefit from GPU offloading. If strict ordering between memory operations needs to be enforced, `threadfence` or `syncthread` could be inserted.

5.2 Application-specific Optimizations

GEVO discovers optimizations related to the particular application. We highlight four such optimizations next.

5.2.1 Removing Conditional Execution. GEVO removes dead code that does not affect program output. In the case of GPUs, GEVO could eliminate code blocks in the conditional path entirely if the input space does not touch that portion of the kernel. Such kernel variants were identified in `hot`, `lud`, and `pf`.

5.2.2 Removing Redundant Load. In `bfs`, GEVO removed certain load instructions from a loop that repeatedly loaded data from the same address. In this case, the compiler inserts these load instructions to guarantee that the latest updates to the particular memory address will be loaded correctly in different iterations of the loop. Programmers can avoid these redundant loads by declaring the corresponding variable using a constant modifier, indicating that the variable is read-only for the entire program execution. In this case, GEVO discovered the data characteristics and addressed the inefficiency without the programmer's involvement.

5.2.3 Loop Perforation. Loop perforation is an optimization technique that skips iterations of a loop based on the *skip factor* and has been explored for approximate computing [94]. GEVO discovers similar optimizations, for example, when loops have been unrolled post-compilation, GEVO removes some part(s) of the unrolled loop while optimizing the fitness function. We observed this behavior in `sc`, `lud`, and `hot`.

5.2.4 Memoization. Memoization is an optimization technique that stores results of expensive function calls and returns the stored value without re-computation when the same inputs occur again. At the LLVM-IR level, GEVO sometimes identifies similar memoization opportunities by eliminating unneeded instructions and using stored results directly. We find this optimization in the HotSpot temperature modeling tool (`hot`). A kernel in the HotSpot tool performs preprocessing based on the physical dimension of a processor chip. Since the shape of simulated chips is the same across all loop iterations, GEVO discovers memoization opportunities to reuse the preprocessing results of the x-dimension for the y-dimension. Another extreme case was found in `b+tree`, where one of the input arguments to the program already represents the desired indices

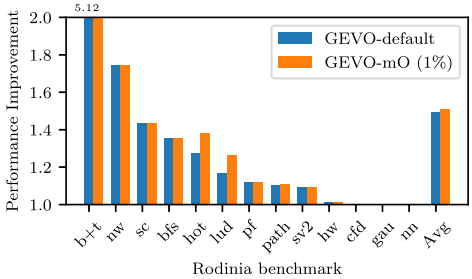

Fig. 9. Comparison of normalized performance improvement between GEVO-default and GEVO-mO in the Rodinia Benchmark.

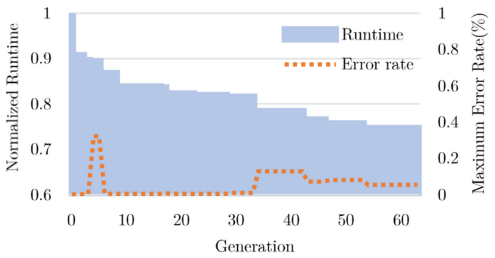

Fig. 10. Temporal evolution of a `hot` kernel variant.

in the program output. In this case, GEVO omits almost the entire kernel, leading to more than 5× performance speedup.

In summary, we have identified several categories of performance improvements found by GEVO, but we have not studied all such optimizations, and in some cases, we require additional domain-specific knowledge to complete a full analysis. Because GEVO is stochastic, it is not guaranteed to find every possible optimization on every run.

6 EVALUATION OF GEVO-MO

In this section, we evaluate GEVO (GEVO-mO) in settings where exact output fidelity is not required. First, we consider GEVO-mO running the Rodinia benchmarks and compare its performance to GEVO-default. We then consider ML workloads and report how GEVO-mo can extract significant performance improvement when domain-specific metrics (model prediction error) are included and co-optimized.

6.1 Rodinia Benchmarks with Error Tolerance

Figure 9 reports overall performance improvement from GEVO-default and GEVO-mO. When accepting up to 1% kernel output difference, GEVO-mO scavenges additional improvements, reducing runtime by an average of 51.08% over the baseline. This is mainly achieved by the additional performance improvement in `hot` and `lud`, from 27.3% to 38% and from 17% to 26.5%, respectively.

6.1.1 Temporal Analysis. To better understand how GEVO-mO co-optimizes runtime and output error, we consider one run of GEVO in closer detail for the program `hot`, as shown in Figure 10. On each generation, the figure plots runtime (primary y-axis) and error rate (minor y-axis) for the most fit kernel variant in that generation. As expected, runtime decreases over the run, but the corresponding error rate increases at Generations 5 and 34. This illustrates the design space tradeoff between performance and accuracy. In both cases, GEVO-mO then "repairs" the error rate by introducing other mutations, a phenomenon known as *compensatory evolution* [98] in evolutionary biology.

There are three key mutations in the last generation. When combined, they reduce the error rate to less than 0.1%, whereas if individually applied, the error rate is much higher at 0.3%. This highlights the strength of a population-based search method like GEVO—sub-optimal individuals in one generation can be combined and/or serve as a stepping stone to the discovery of successful combinations of mutations. Further, the best kernel variant would not be found if a tighter error bound had been enforced from the beginning.

6.1.2 Optimization Analysis. For `hot` and `lud`, mutation analysis reveals how additional improvements are achieved.

`hot`**:** Additional performance improvement is achieved by removing additional synchronization primitives. This optimization raises the possibility of a race condition, because the outputs vary slightly from run to run but always remain under the 1% threshold. Although race conditions are a potential concern, earlier work proposes lock-free approaches for specific algorithms and includes a proof that the algorithm can converge even with a race condition [80].

Our analysis of the `hot` optimization provided two possible explanations for how removing the synchronizations leaves a viable program. First, in `hot`, each thread updates the temperature of a spot in a two-dimension grid, based on the ambient temperature and the temperature of the surrounding spots. Synchronization on each timestep allows every other thread to calculate and update its temperature based on up-to-date temperature values nearby. However, there are two levels of synchronization in `hot` : in-kernel synchronization using the `syncthreads` function and a global synchronization when the kernel is relaunched. Removing the `syncthreads` call inside the kernel implies that the temperature over the grid will be synchronized only every other timestep, which might leave the simulation within its error tolerance. The second explanation derives from the simulation setting. In `hot`, the temperature difference between the ambient and initial grid temperature (ambient is set to 80°K while the grid is mostly between 320°K and 345°K) is much larger than the difference between a given spot and its neighbors (usually within 10°K). As a result, the ambient temperature contributes more than surrounding temperature to the thermal simulation. This effect mitigates the effect of the inaccuracy introduced by removing the synchronization.

`lud`**:** GEVO finds performance improvement by reusing the result from an earlier iteration of a loop and avoiding computation in the latter iteration because the loop has been unrolled by the compiler. This optimization is an example of memoization introduced in the previous section. `lud`, standing for Lower-Upper decomposition, decomposes a given matrix into a product of two triangular matrices where each one has the lower/upper part of matrix filled with zeroes. Although it might seem unacceptable to tolerate any error in the solution of a linear system, a method known as incomplete LU factorization [67] approximates the solution of LU decomposition for lower computational cost. We suspect that GEVO accidentally re-discovered this technique for improving `lud` 's performance, providing a nice example of approximate computing.

In other Rodinia benchmarks, GEVO failed to find significant improvements when output fidelity was relaxed. Although we do not know why these applications were more challenging, there are several possibilities, including the most obvious one that the 1% error tolerance based only on the raw kernel output from the GPU is too constrained to allow appreciable expansion of the optimization search space. Therefore, in the next section, we change the error definition from GPGPU kernel output difference to application-defined error. For ML, the natural application error is model prediction error. This change allows GEVO to find application-specific optimizations that produce significantly different kernel output while maintaining model accuracy.

6.2 Evaluation of Machine Learning Applications (Dataset-specific Optimizations)

Machine learning (ML) is a popular class of intrinsically error-tolerant applications, which consume large computational resources, and is particularly suitable for the GEVO approach. We consider two ML models, SVM and ResNet18, and use them to illustrate how performance and accuracy can be co-optimized. Although earlier work examined accuracy/runtime tradeoffs of implementations [39], we are unaware of earlier work targeting genetic improvement of ML LLVM-IR kernels.

6.2.1 SVM. Figure 11 shows the Pareto frontiers found by GEVO for handwriting recognition (SVM with `MNIST`) and income prediction (SVM with `a9a`). The x-axis represents the measured

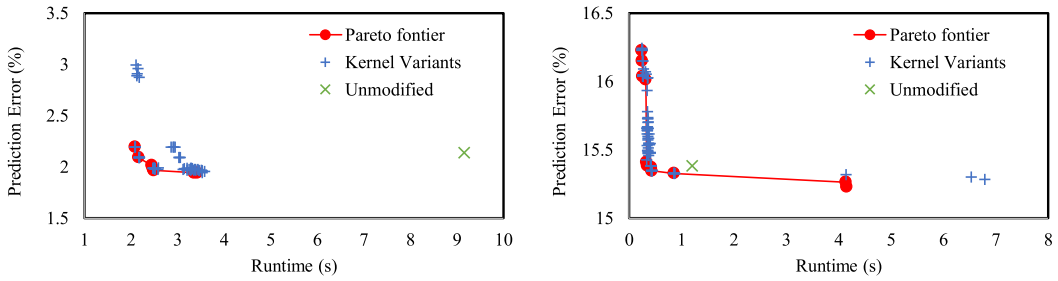

Fig. 11. Pareto-frontiers and other kernel variant measurements for (a) the handwriting recognition (SVM with MNIST) and (b) the income prediction (SVM with a9a).

kernel runtime and the y-axis represents the training inference prediction error in %. We report results for each kernel variant in GEVO's final generation, relative to the original unmodified kernel. Figure 11 also shows how GEVO-mO navigates away from the original, sub-optimal kernel implementation and explores the better-performing part of the search space.

Considering the kernel variant in the Pareto frontier that represents the best combined improvement, we find that GEVO-mO achieves 3.24× improvement for handwriting recognition (MNIST) and 2.93× performance improvement for income prediction (a9a)'s kernel performance, which increases overall model training speed by 50% and 24.8%, respectively. At the same time, accuracy on the training set improved from 97.86% to 98.03% (MNIST) and from 84.61% to 84.65% (a9a), which was unexpected, as we imagined the optimization would trade off accuracy against training time. Next, we tested the trained GEVO-optimized models on their official testing datasets, where we find accuracy is improved slightly, from 98.37% to 98.5% (MNIST) and from 84.59% to 84.64% (a9a).

We also consider whether the SVM evolved for training a specific dataset can achieve similar improvements on a different dataset in the same class (after all, that would be the main advantage of optimizing the training procedure for a particular type of application). We tested SVM optimized for the MNIST common dataset (60,000 samples) by using it to train the large MNIST dataset (8M handwriting samples). Since the large MNIST dataset does not have a separate testing dataset, we report the 10-fold cross validation accuracy for the model from unmodified and optimized ThunderSVM, which are 100% and 99.997% with the respective training time in 1,182 and 121 minutes.

Our optimization analysis, shown in Figure 12, shows how GEVO changes the termination condition of a while loop by increasing the lower bound by 1 in line 11. As a result, there is a chance of producing a smaller value in the `if` statement in line 14, causing the execution to exit the while loop sooner. This loop implements an SVM solver using *sequential minimal optimization*, which iteratively approaches the optimal solution, terminating when progress has slowed. Thus, GEVO relaxes the convergence condition, which would normally be expected to reduce solution correctness. However, for MINST, this change actually improves model accuracy, perhaps by avoiding overfitting. We leave further analysis of this surprising result for future work.

6.2.2 ResNet18. Similar to SVM, Figure 13(a) presents the Pareto-frontier for image classification (ResNet18 with CIFAR-10). Here, we select the kernel variant that gives the best training accuracy on the third epoch (56% compared to 47% in the Baseline). This variant is also 1.79× faster than original kernel. However, the kernel contributes less than 1% of the entire training time. Recall that we do not have access to the other kernels for ResNet18 operators (Section 4.2).

```
1  ...
2  while (1)
3    // select f Up
4    if (is_I_up(...))
5      f_val_reduce[tid] = f;
6    up_val = f_val_reduce[...];
7
8    // select f Low
9    if (is_I_low(...))
10     // f_val_reduce[tid] = -f;
11     f_val_reduce[tid] = 1 - f;
12   down_val = f_val_reduce[...];
13
14   if (up_val - down_val < epsilon)
15     break;
```

Fig. 12. Code snippet from ThunderSVM illustrates an optimization discovered by GEVO. The comment at line 10 is the original code and line 11 indicates the GEVO modification.

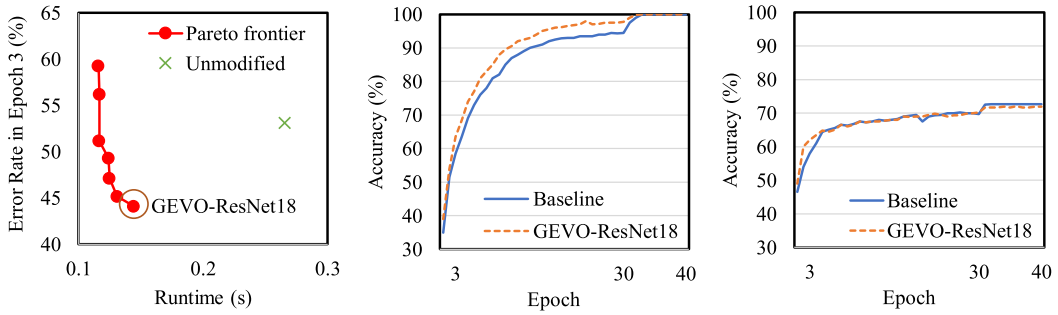

Fig. 13. (a) Pareto-frontiers for the image classification (ResNet18 with CIFAR-10). The kernel in the circle is manually selected to retrain the model until model converges with its (b) training accuracy and (c) testing accuracy across epochs.

Recalling that we train ResNet18 for only three epochs during GEVO's optimization searches and only on the momentumSGD kernel, we first examine the kernel performance throughout the training process until the point where model accuracy has converged. Figures 13(b) and 13(c) report training and testing accuracy across epochs for the original baseline kernel and a GEVO-optimized variant. Considering training accuracy, the GEVO-optimized kernel consistently beats the baseline across epochs by up to 7% until epoch 30, when both kernels achieve 100% training accuracy. The sudden jump in training accuracy around epoch 30 is caused by a learning rate change in epoch 29, which occurs in code that is not available to GEVO. If we ignore that external learning rate change, the GEVO-optimized kernel would converge at epoch 25, and the baseline would converge at epoch 28, with 2% lower training accuracy. Turning to testing accuracy, both kernels have comparable accuracy through epoch 30 when the learning rate is changed. After epoch 30, the baseline is 1% more accurate than the variant (73.24% to 72.31%).

Figure 14 shows an important code optimization found by GEVO in the Caffe2 momentumSGD operator. There are two modifications leading to the accuracy and performance improvements. First, GEVO changes the loop boundary so the loop is executed only once (line 7 in Figure 14). The momentumSGD kernel updates the parameters (weight and bias) of the neural network and

```
/*    N = number of parameters
 * m[i] = momentum
 * g[i] = gradient
 * BETA = momentum decay rate
 *   LR = learning rate
 */
for (i=tid; i<N; i+= GRID_SIZE N) {
  float mi = m[i];
  float mi_new = BETA*mi + LR*g[i];
  m[i] = mi_new LR*g[i];
  g[i] = (1+BETA)*mi_new - BETA*mi;

  if (param)
    param[i] -= g[i];
}
```

Fig. 14. Code snippet from Caffe2 momentumSGD operator illustrates two optimizations discovered by GEVO.

the loop here represents how many parameters need to be updated. Thus, GEVO's success when reducing the number of loop iterations suggests that the ResNet18 model is overly complicated for the CIFAR-10 dataset. This is similar to weight pruning [70] or hyperparameter search [14] for a particular dataset. GEVO also changes how the momentum is calculated by using only the current gradient and not considering the prior gradient. This optimization illustrates how GEVO can tailor an algorithm to a particular dataset.

7 DISCUSSION

This article presents GEVO, a new method that uses stochastic population-based search to discover optimizations of GPGPU kernels. GEVO trades off absolute program semantics for other important non-functional design aspects. The experimental results demonstrate that by relaxing program semantics, GEVO can find novel and substantial improvements, both for runtime alone and for the case of multiple optimization objectives, e.g., accuracy and runtime. The proposed approach, while not intended for applications with critical correctness requirements (e.g., inner loops of avionics software or some systems programs), is suitable for many other applications, including the important class of ML codes. When we consider handwriting recognition, income prediction, and image recognition ML workloads, our results show that GEVO explores the optimization search space and finds multiple points along the Pareto frontier that maximize tradeoffs between performance and accuracy. In some cases, however, GEVO can "have the best of both worlds" by finding a significant 3.24× speedup of the handwriting recognition kernel (SVM with `MNIST`) and achieve modest improvements of prediction accuracy. This translates to 50% training time reduction with 0.17% improvement on the prediction accuracy, reflecting absolute improvements in both dimensions.

By design, GEVO does not aspire to preserve exact program semantics, and in many cases, it can identify algorithmic improvements that are inaccessible to methods that require complete semantic consistency with the original program. In other circumstances, however, GEVO could potentially find optimizations that break required semantic properties not enforced by the test suite. Optimizations that relax memory synchronization requirements provide a good example. In the cases we examined, eliminating redundant synchronizations did not affect program behavior. However, this strategy is risky in general, because it depends on specific memory access patterns, which in turn rely on the combined effect of the target application and its execution environment (hardware architecture and system software). There is currently a great deal of research interest

in studying how memory accessing order can be relaxed for better performance. This includes application-specific approaches, e.g., to schedule and prioritize memory access for specific tasks [4, 73], and system-level approaches, such as non-blocking or wait-free synchronization with system or architecture support [23, 110]. GEVO could potentially be applied to identify these optimization spots for researchers to further analyze when necessary.

More generally, as we learn more about when and how GEVO succeeds and fails, we foresee new methods for post hoc validation of evolved codes, e.g., by synthesizing new test cases on the fly to test synchronization or ultimately, using program analysis methods to highlight semantic differences between original and evolved kernels [19].

We have also explored the idea of enhancing the proposed design by considering system-specific architecture features. Cache efficiencies can affect the performance of GPUs. We extended the scope of GEVO to explore the performance optimization space of the GPU cache configuration. In this case, we enabled GEVO to control cache bypassing by introducing `ld.cg` (for caching at the L2 cache but bypassing the L1 cache) and `ld.ca` (for caching at both the L1 and L2 caches) to the genetic operations. By doing so, the mutation operation can specify whether data are bypassed from the GPU L1 cache at the granularity of instructions in the NVIDIA PTX ISA (by injecting inline-assembly in LLVM-IR) and discover performance speedup opportunities, similar to on-demand cache fetching [40] or cache bypassing optimization [7, 55]. Overall, this mutation operator did not produce frequent enough performance improvements to justify adding it to GEVO's mutation suite. It seems that GEVO often finds equivalent optimizations without explicitly using the cache-specific mutation, simply by moving load instructions.

The results reported here are specific to the programs, inputs, and the particular GEVO runs we studied. There were some programs for which GEVO was unable to find improvement. Thus, further experimentation is required to understand the generality of our results. GEVO found application-specific, architecture-specific, and dataset-specific optimizations. In future work, we plan to test GEVO on other applications and analyze more carefully why some programs admit significant improvements and others do not. Since GEVO's approach is agnostic about optimization criteria, it is easy to imagine other compelling optimizations. For example, GEVO could customize the LLVM-IR for particular classes of inputs or even generate diverse versions of the kernel, each of which uses a different power budget, to defeat some power side channel attacks.

GEVO itself has many possible parameter settings, including population size, mutation and crossover rates, and there are many existing evolutionary algorithms with different strategies for selection and multi-objective function optimization. We began with the most popular multi-objective framework (DEAP), modified it for our application, and conducted several initial experiments to find a configuration that works well for GPGPU optimization. However, it is certainly possible that other evolutionary algorithms or other parameter settings for GEVO would produce better results. Similarly, we chose a 1% error tolerance arbitrarily for the GEVO-mO experiments. Acceptable errors may vary across programs, and in some cases, could translate into improved optimizations. An example can be found in Yazdanbakhsh's work [112] where `srad` from the Rodinia benchmark could accept up to 10% error, with additional optimizations being revealed. Increasing GEVO's error tolerance to 10% for applications such as this could lead to additional optimizations.

Our mutation operators are more expensive than those used in earlier work on genetic improvement of software. The additional cost arises from the nature of the single static assignment discipline in the LLVM-IR. As a result, as Table 1 shows, GEVO searches for a very low number of generations on many of the benchmarks. A general rule-of-thumb would suggest running the EC search for at least as many generations as there are individuals in the population (250 in our experimental setup). With additional computational resources, we could expect additional performance improvements, especially on the benchmarks that ran for fewer than 30 generations.

It is well-known that significant human expertise is required to tune an ML model to extract the best performance on a particular task. For instance, in SVM, the regularization parameter known as C is manually determined to balance the generalization and the training error (underfitting versus overfitting). Similarly, in neural networks human expertise and experimentation are used to find an appropriate network architecture, which determines how many neurons are used and how they are connected. Currently, these design decisions are determined empirically for each dataset and often tested repeatedly. Automating these design decisions, commonly referred to as *hyperparameter search* [14] or *AutoML* [99], to reduce human effort is an active area of current research, and many algorithms have been proposed, including simple grid search [49], random search [14], reinforcement learning [12, 115], evolutionary computation [97], and gradient decent [60]. These hyperparameter search algorithms differ from GEVO in that they do not touch the underlying code implementation. The results presented here show that GEVO can discover effects that are similar to those found with hyperparameter search at the same time that it improves the implementation itself. This suggests that GEVO may be capable of performing hyperparameter search along with network optimizations. We plan to conduct additional experiments expanding the work described here, with the MLPerf Training and Inference Benchmark Suites [65, 66, 81].

8 CONCLUSION

Programmers today develop software that is expected to meet functional correctness, to avoid opening up new security vulnerabilities, and to execute efficiently. The software is often developed on complex programming stacks, is compiled to run on complex proprietary architectures that lack transparency, and it often has unanticipated interactions with runtime environments and workloads. This article presents one approach to managing this complexity, focusing on code optimization for GPUs. GPUs are an appealing target, because they are widely used to accelerate compute-intensive applications and because GPGPU codes are often error-tolerant and amenable to approximate optimization methods that do not guarantee exact semantic equivalence to the original program.

Our approach, implemented as GEVO, uses population-based stochastic search on LLVM-IR GPGPU programs. GEVO finds versions of programs that retain required functionality, as assessed by test cases, and optimize one or more fitness criteria. We focus on the dual objectives of minimizing runtime and application error, finding significant reductions in runtime for most programs with little or no penalty in output error. GEVO finds optimizations that leverage details about the architecture, the application design, and even particular workloads. For large-scale computation-intensive applications such as ML, we hope that the methods presented here can contribute to improved software deployments. We also hope that some of the unusual optimization opportunities identified by GEVO will lead to improved software development practices, whether through improved tools or through improved awareness on the part of application developers.

ACKNOWLEDGMENTS

We thank F. Esponda, W. Weimer, E. Schulte, and the reviewers for many insights, code, and helpful comments.

REFERENCES

[1] TensorFlow. 2018. XLA is a compiler that optimizes TensorFlow computations. Retrieved from https://www.tensorflow.org/xla/.

[2] Advanced Micro Devices, Inc. 2020. AMD Exascale Supercomputer. Retrieved from https://www.amd.com/en/products/exascale-era.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*.

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. *SIGPLAN Not.* 50, 8 (2015), 11–20. DOI: https://doi.org/10.1145/2858788.2688523

[5] Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (2008), 5342–5359.

[6] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-chip-module GPUs for continued performance scalability. *SIGARCH Comput. Archit. News* 45, 2 (2017), 320–332. DOI: https://doi.org/10.1145/3140659.3080231

[7] Akhil Arunkumar, Shin-Ying Lee, and Carole-Jean Wu. 2016. ID-cache: Instruction and memory divergence based cache management for GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*.

[8] Shumeet Baluja and Rich Caruana. 1995. Removing the genetics from the standard genetic algorithm. In *Proceedings of the International Conference on Machine Learning*.

[9] Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole superoptimizers. *SIGARCH Comput. Archit. News* 34, 5 (2006), 394–403. DOI: https://doi.org/10.1145/1168919.1168906

[10] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The problem of programming language concurrency semantics. In *Proceedings of the European Symposium on Programming Languages and Systems*.

[11] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. 2015. Automatic software diversity in the light of test suites. *arXiv preprint arXiv:1509.00144* (2015).

[12] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* (2016).

[13] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. 2018. Understanding and simplifying one-shot architecture search. In *Proceedings of the International Conference on Machine Learning*.

[14] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, Feb. (2012), 281–305.

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

[16] Bobby Ralph Bruce, Justyna Petke, Mark Harman, and Earl T. Barr. 2019. Approximate oracles and synergy in software energy search spaces. *IEEE Trans. Softw. Eng.* 45, 11 (2019), 1150–1169. DOI: https://doi.org/10.1109/TSE.2018.2827066

[17] Forbes J. Burkowski. 1999. Shuffle crossover and mutual information. In *Proceedings of the Congress on Evolutionary Computation (CEC'99)*.

[18] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. Object-oriented genetic improvement for improved energy consumption in Google Guava. In *Proceedings of International Symposium on Search Based Software Engineering*.

[19] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding automatically generated patches through symbolic invariant differences. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*.

[20] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (2011), 27 pages. DOI: https://doi.org/10.1145/1961189.1961199

[21] 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 preprint arXiv:1512.01274* (2015).

[22] 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 13th USENIX Symposium on Operating Systems Design and Implementation*.

[23] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. 2010. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society.

[24] Berkeley Churchill, Rahul Sharma, J. F. Bastien, and Alex Aiken. 2017. Sound loop superoptimization for Google native client. *SIGARCH Comput. Archit. News* 45, 1 (2017), 313–326. DOI: https://doi.org/10.1145/3093337.3037754

[25] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python framework for evolutionary algorithms. In *Proceedings of the 14th Conference on Genetic and Evolutionary Computation*.

[26] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* (2002).

[27] 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*.

[28] Inderjit S. Dhillon and Dharmendra S. Modha. 2002. A data-clustering algorithm on distributed memory multiprocessors. In *Large-scale Parallel Data Mining*. Springer, 245–260.

[29] Jonathan Dorn, Jeremy Lacomis, Westley Weimer, and Stephanie Forrest. 2019. Automatically exploring tradeoffs between software output fidelity and energy costs. *IEEE Trans. Softw. Eng.* 45, 3 (2019), 219–236. DOI: https://doi.org/10.1109/TSE.2017.2775634

[30] Facebook. 2018. Finding and Fixing Software Bugs Automatically with Sapfix and Sapienz. Retrieved from https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/.

[31] Facebook. 2019. Caffe2. Retrieved from https://caffe2.ai/.

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

[33] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72. DOI: https://doi.org/10.1109/TSE.2011.104

[34] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *SIGPLAN Not.* 46, 6 (2011), 62–73. DOI: https://doi.org/10.1145/1993316.1993506

[35] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, John Wawrzynek, Krste Asanovic, and Ion Stoica. 2020. AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning. In *Proceedings of the 3rd Conference on Machine Learning and Systems (ML-Sys'20)*.

[36] Saemundur O. Haraldsson, John R. Woodward, Alexander, E. I. Brownlee, A. V. Smith, and V. Gudnason. 2017. Genetic improvement of runtime and its fitness landscape in a bioinformatics application. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[37] Saemundur O. Haraldsson, John R. Woodward, Alexander E. I. Brownlee, and Kristin Siggeirsdottir. 2017. Fixing bugs in your sleep: How genetic improvement became an overnight success. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[38] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*.

[39] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. 2017. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[40] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*.

[41] 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 Symposium on Operating Systems Principles (SOSP'19)*.

[42] Dan Judd, Philip K. McKinley, and Anil K. Jain. 1998. Large-scale parallel data clustering. *IEEE Trans. Pattern Anal. Mach. Intell.* 20, 8 (1998), 871–876.

[43] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P. Xing. 2018. Neural architecture search with Bayesian optimisation and optimal transport. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*.

[44] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 1097–1105.

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

[47] William B. Langdon and Mark Harman. 2014. Genetically improved CUDA C++ software. In *Proceedings of 17th European Conference on Genetic Programming*.

[48] William B. Langdon and Mark Harman. 2015. Grow and graft a better CUDA pknotsRG for RNA Pseudoknot free energy calculation. In *Proceedings of the 17th Conference on Genetic and Evolutionary Computation*.

[49] 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 Conference on Genetic and Evolutionary Computation*.

[50] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. 2007. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning*.

[51] Yann Le Cun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324. DOI: https://doi.org/10.1109/5.726791

[52] 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*.

[53] 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.

[54] C.-Y. Lee and E. K. Antonsson. 2000. Variable length genomes for evolutionary algorithms. In *Proceedings of the 2nd Conference on the Genetic and Evolutionary Computation Conference*.

[55] Shin-Ying Lee and Carole-Jean Wu. 2016. Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs. In *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD'16)*.

[56] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Genetic improvement of GPU code. In *Proceedings of the 6th International Workshop on Genetic Improvement (GI'19)*.

[57] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Uncovering Performance Opportunities by Relaxing Program Semantics of GPGPU Kernels. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems: Wild and Crazy Idea session*.

[58] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*.

[59] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436* (2017).

[60] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[61] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*.

[62] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Softw. Eng.* 40, 1 (2014), 23–42. DOI: https://doi.org/10.1109/TSE.2013.44

[63] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*.

[64] Henry Massalin. 1987. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[65] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2019. MLPerf training benchmark. *arXiv preprint arXiv:1910.01500* (2019).

[66] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, G. Wei, and C.-J. Wu. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro* 40, 2 (2020), 8–16. DOI: https://doi.org/10.1109/TSE.2013.44

[67] J. Andvandervorst Meijerink and Henk A. Van Der Vorst. 1977. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. of Comput.* 31, 137 (1977), 148–162. DOI: https://doi.org/10.2307/2005786

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

[69] Brad L. Miller, David E. Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems* 9, 3 (1995), 193–212. Retrieved from https://www.complex-systems.com/abstracts/v09_i03_a02/.

[70] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient inference. In *Proceedings of International Conference on Learning Representations*.

[71] David J. Montana and Lawrence Davis. 1989. Training feedforward neural networks using genetic algorithms. In *Proceedings of the International Joint Conferences on Artificial Intelligence*.

[72] Leonardo De Moura and Nikolaj Björner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Theory and Practice of Software*.

[73] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[74] 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 *NeurIPS Autodiff Workshop*. Retrieved from https://openreview.net/forum?id=BJJsrmfCZ.

[75] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[76] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268* (2018).

[77] John C. Platt. 1999. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, USA, 185–208. https://dl.acm.org/doi/10.5555/299094.299105

[78] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4780–4789.

[79] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning, Vol. 70*. JMLR. org.

[80] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*.

[81] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf inference benchmark. *arXiv preprint arXiv:1911.02549* (2019).

[82] 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 preprint arXiv:1805.00907* (2018).

[83] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 73–82.

[84] David Saad. 1998. Online algorithms and stochastic approximations. *Online Learn.* 5 (1998), 6–3.

[85] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *SIGARCH Comput. Archit. News* 41, 1 (2013), 305–316. DOI: https://doi.org/10.1145/2490301.2451150

[86] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *SIGPLAN Not.* 49, 6 (2014), 53–64. DOI: https://doi.org/10.1145/2594291.2594302

[87] Eric Schulte. 2014. *Neutral Networks of Real-world Programs and Their Application to Automated Software Evolution*. Ph.D. Dissertation. University of New Mexico, Albuquerque.

[88] Eric Schulte, Jonathan DiLorenzo, Stephanie Forrest, and Westley Weimer. 2013. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

[90] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Prog. Evolv. Mach.* 15, 3 (2014), 281–312. DOI: https://doi.org/10.1007/s10710-013-9195-8

[91] Eric M. Schulte, Westley Weimer, and Stephanie Forrest. 2015. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Proceedings of the 1st Genetic Improvement Workshop*.

[92] Michael J. Schulte, Mike Ignatowski, Gabriel H. Loh, Bradford M. Beckmann, William C. Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K. Reinhardt, and Gregory Rodgers. 2015. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro* 35, 4 (2015), 26–36. DOI: https://doi.org/10.1109/MM.2015.71

[93] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. In *Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications*.

[94] 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 Conference on Foundations of Software Engineering*.

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

[96] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artif. Life* 15, 2 (2009), 185–212. DOI: https://doi.org/10.1162/artl.2009.15.2.15202

[97] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evol. Comput.* 10, 2 (2002), 99–127.

[98] Wolfgang Stephan. 1996. The rate of compensatory evolution. *Genetics* 144, 1 (1996), 419–426. Retrieved from https://www.genetics.org/content/144/1/419.

[99] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'13)*.

[100] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with ROSETTE. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '13)*.

[101] Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

[102] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. 2005. DIABLO: A reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*.

[103] Nadarajen Veerapen, Fabio Daolio, and Gabriela Ochoa. 2017. Modelling genetic improvement landscapes with local optima networks. In *Proceedings of the Genetic and Evolutionary Computation Conference*.

[104] Phillip Verbancsics and Kenneth O. Stanley. 2011. Constraining connectivity to encourage modularity in HyperNEAT. In *Proceedings of the 13th Conference on Genetic and Evolutionary Computation*. ACM.

[105] Lizhe Wang, Jie Tao, Marcel Kunze, Alvaro Canales Castellanos, David Kramer, and Wolfgang Karl. 2008. Scientific cloud computing: Early definition and experience. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*. IEEE, 825–830.

[106] 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*.

[107] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2018. ThunderSVM: A fast SVM library on GPUs and CPUs. *J. Mach. Learn. Res.* 19, 21 (2018), 1–5. Retrieved from http://jmlr.org/papers/v19/17-740.html.

[108] D. R. White, A. Arcuri, and J. A. Clark. 2011. Evolutionary improvement of programs. *IEEE Trans. Evol. Comput.* 15, 4 (2011), 515–538. DOI: https://doi.org/10.1109/TEVC.2010.2083669

[109] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An open-source GPGPU compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*.

[110] Shucai Xiao and Wu-chun Feng. 2010. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*. IEEE.

[111] Lingxi Xie and Alan Yuille. 2017. Genetic CNN. In *Proceedings of the IEEE International Conference on Computer Vision*.

[112] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2016. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Des. Test* 34, 2 (2016), 60–68. DOI: https://doi.org/10.1109/MDAT.2016.2630270

[113] Jieming Yin, Zhifeng Lin, Onur Kayiran, Matthew Poremba, Muhammad Shoaib Bin Altaf, Natalie Enright Jerger, and Gabriel H. Loh. 2018. Modular routing design for chiplet-based systems. In *Proceedings of the ACM/IEEE 45th International Symposium on Computer Architecture (ISCA'18)*.

[114] Sixin Zhang, Anna E. Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 685–693.

[115] Barret Zoph and Quoc V. Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[116] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

Received November 2019; revised April 2020; accepted July 2020