Genetic Improvement of GPU Code

Jhe-Yu Liou, Stephanie Forrest, Carole-Jean Wu
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ
{jhe-yu.liou, stephanie.forrest, carole-jean.wu}@asu.edu

***Abstract*—As the programming stack and tool support for GPU have matured, GPUs have become accessible to programmers who often lack domain-specific knowledge of the underlying architecture and fail to fully leverage the GPU's computation power. This paper presents GEVO (Gpu EVOlution), a tool for automatically tuning the performance of GPU kernels in the LLVM representation to meet desired criteria. GEVO uses population-based search to find edits to programs compiled to LLVM-IR that improve performance on desired criteria and retain required functionality. GEVO extends earlier GI work by operating directly on the LLVM-IR without custom representations or other manual interventions. We demonstrate that GEVO improves runtime on NVIDIA Tesla P100 for many programs in the Rodinia benchmark suite and a supervised machine learning code, ThunderSVM. For the Rodinia benchmark, GEVO improves GPU kernel runtime performance by an average of 13.87% and as much as 43% over the fully compiler-optimized baseline. If the kernel output accuracy is relaxed to tolerate 1% error, GEVO can find kernel variants that outperform the baseline version by an average of 15.47%. For ThunderSVM, GEVO reduces entire model training time by 50% and 24.8%, for MNIST handwriting recognition dataset and a9a income prediction, where the accuracy of trained model are improved by 0.17% and 0.04% respectively.**

***Index Terms*—Genetic Improvement; Multi-objective Evolutionary Computation; GPU code optimization; LLVM Intermediate Representation;**

I. Introduction

The fields of big data analytics, large-scale machine learning and scientific algorithms are expanding quickly and are one of the biggest drivers of high-performance computing. Continued advances in these fields will not be possible without orders-of-magnitude improvements in performance of computing systems. GPUs help address this challenge and have become de-facto co-processors for accelerating the performance of general-purpose, large-scale parallel workloads. However, the maturation of the GPU programming interface has made GPUs accessible to programmers who may lack knowledge of GPU architecture, and it is challenging to further optimize and fine-tune the performance of general purpose GPU programs without platform- and domain-specific knowledge. For example, programmers may be excessively cautious in their use of synchronization, which inhibits potential speedups.

There are many post-compilation optimization methods already in use, including peephole, link-time, and profile-guided optimization, but additional efficiencies can be found by tailoring the 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 [1]. STOKE uses Markov Chain Monte Carlo (MCMC) sampling to search for codes that run faster but does not naturally scale up to large code sizes, which is why we favor genetic improvement (GI) approach However, there have been few attempts to apply stochastic methods to the LLVM-IR, in part because the IR has many data dependencies and requires careful implementation of code modification operations.

We propose a post-compilation performance tuning approach, called GPU EVOlution or *GEVO*, for discovering optimized GPGPU kernel implementations using Genetic Programming (GP). GEVO encodes desired optimization objectives as the fitness function and implements a set of mutation and recombination operators for GPU kernel transformations that can be applied to the LLVM-IR. 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 code to meet two independent objectives, such as performance and accuracy (GEVO-mO).

To assess the general applicability of GEVO, we evaluate on NVIDIA Tesla P100 with the Rodinia benchmark suite. GEVO-default improves GPU kernel runtime performance by an average of 13.87% and by as much as 43% over the fully-compiler optimized baseline. If the output accuracy is relaxed to tolerate 1% error, GEVO-mO can find kernel variants that outperform the baseline version by an average of 15.47%.

We also evaluate GEVO on a supervised machine learning code for handwriting recognition (the MNIST data [2]) and for income prediction (a9a dataset [3]). We evaluate GEVO-mO by encoding both machine learning model training time and inference prediction accuracy into the fitness function. GEVO-mO improves the overall model training speed for handwriting recognition and income prediction datasets by 50% and 24.8% respectively, with improved accuracies of 0.17% and 0.04%, reflecting absolute improvements in both dimensions.

Our key contributions are:

- We present GEVO, a tool for automatically tuning the performance of CUDA GPU kernels represented in 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 run-time optimization and on the multi-objective optimization criteria of runtime and accuracy.

- For the machine learning kernel, thunderSVM on the MNIST dataset, GEVO finds an astonishing 3.24X speedup of the training kernel with improved inference prediction accuracy. This translates to 50% training time reduction with 0.17% improvement on the prediction accuracy, reflecting absolute improvements in both dimensions.
- Our analysis of GEVO optimizations discusses both architectural- and domain-specific improvements. We provide explanations for several performance optimization features discovered by GEVO, such as eliminating conservative synchronization primitives, removing redundant `store` instructions, reducing conditional executions, loop perforation, and memoization (Section V-A and V-B).

II. Related Work

Genetic Programming (GP) methods have been used to improve computer programs, e.g., to automatically repair bugs in legacy software [4]–[8]. Transitions to industrial practice include Facebook's SapFix tool [9] and the Janus Manager deployment [10]. Although most work has been conducted at the source-code level using abstract syntax trees, similar methods have been applied to assembly programs [11] and object code [12]. Subsequent analysis showed that the applied mutations often have no observable effect on program behavior [13]–[17]. 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., [18]. These results suggested the possibility of using GP 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 GP to improve program performance [19], and Schulte et al. achieved energy reductions for several Parsec benchmarks [20]. Bruce et al. also applied the similar technique for MiniSAT to reduce energy consumption [15]. There are other works [21], [22] to constraint the GP's search space on improving energy consumption of Java programs by asking users to provide the predefined locations or equivalent function or class implementations. These and several subsequent works [23], [24] demonstrate the potential for stochastic search such as GP to improve a program's performance or energy efficiency through machine- or architecture-specific optimizations. However, these methods are not yet mature or carefully analyzed; they typically target the CPU; and their general applicability is not well understood.

There is some previous work on GPU kernels, including a graphics shader program [25]. This work began with a basic lighting algorithm and used GP 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 GP to reduce runtime on a CUDA program, reporting results on the program `gzip` [26] and an RNA analysis program [27]. This prior work targets a single program operating in a specific domain, and the methodology

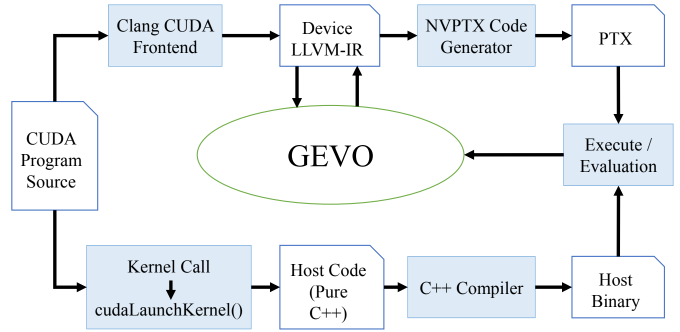

Fig. 1: GEVO in the LLVM/Clang compilation flow.

used in [26]–[29] 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 is the only work we are aware of that has experimented with evolving the LLVM intermediate representation (LLVM-IR) [30], 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 GPU kernels into LLVM-IR, but this has only been available since 2016 [31]. Thus, we adopt the Clang/LLVM infrastructure for GEVO including the LLVM-IR, as shown in Figure 1. This avoids developing novel parsing and syntax manipulating infrastructure, but it introduces new challenges for implementing the basic mutation and recombination operations.

Early work on superoptimization dates back to Massalin's superoptimizer [32] in 1987, which exhaustively searched a subset of the Motorola 68020 assembly instruction set, and synthesized the shortest instruction sequence, for a target function. More recently, Schkufza et al. extended this work by proposing STOKE [1], [33]–[35] using Markov Chain Monte Carlo (MCMC) to focus the search effort on a hot area of the program, so the unrealistically large search space can be reduced for practical applications. STOKE has the same overarching goal as our work, searching for program optimizations without forcibly preserving program semantics. However, STOKE focuses on synthesizing instruction sequences from scratch using the entire instruction set, while we modify existing programs using existing instructions and scale to program sizes at a similar order. Thus, regardless of the language targets and system platforms, the test programs in STOKE are in the range of a dozen up to a few hundred instructions, while our approach has been tested on programs consisting of thousands of LLVM-IR instructions.

III. The Proposed Design: GEVO

We propose *GEVO*—a tool for automatically improving kernel implementations for GPUs. 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

Algorithm 1 The main loop of GEVO.

Parameter: *PopSize, CrossRate, MutateRate*
Input: *GPU kernel Program, P*

```
1:  pop ← Initialize(PopSize, P)
2:  for all individual in pop do
3:      Mutate(individual)*3
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)
```

evaluating mutated kernel variants in an iterative population-based search. Figure 1 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 LLVM intermediate representation (LLVM-IR) with the clang compiler. GEVO then takes 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 CPU is modified to load the generated PTX file into the GPU. GEVO then evaluates how the kernel variant performs as defined by 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 three random mutations to each (Line 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 forms the next generation by comparing 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.

A. 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 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

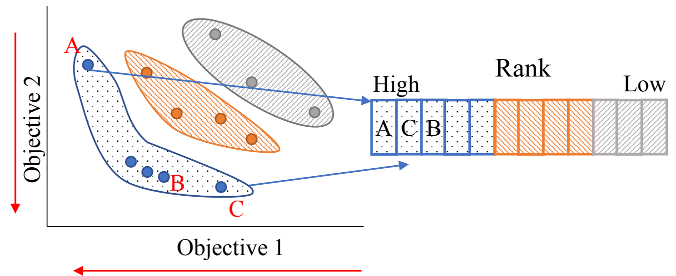

Fig. 2: Non-dominated sorting with a crowding method to enforce diversity.

variant consists of the LLVM-IR code and a set of the mutations that produced it from the original.

This design decision relates to 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 relatively low, and because kernels are relatively small-sized programs, the cost of storing both representations is reasonable.

B. Fitness Evaluation

Although GEVO can optimize any desired fitness function, we focus first on the problem of reducing kernel execution time (*GEVO-default*). In this case, the fitness function is simply the runtime of the kernel variant. When we consider the tradeoff between execution time and output accuracy (*GEVO-approximate*), the fitness function is multi-objective, i.e., $argmin(time, error)$.

Using the fitness criteria, as above, each kernel variant is evaluated by running it against all available test cases. To test for 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 IV-C).

C. Selection

GEVO uses the Non-dominated Sorting Genetic Algorithm (NSGA-II) [36] to select individuals according to multi-objective fitness criteria. Figure 2 illustrates a set of kernel variants, plotted according to two dimensions of the fitness function, say error and run-time, 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

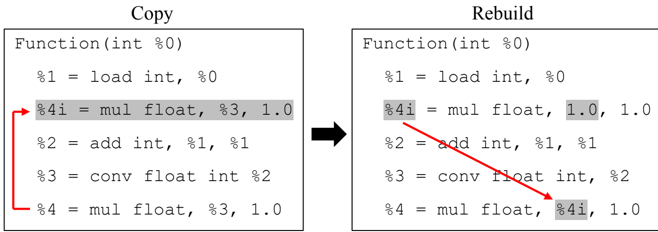

Fig. 3: Mutate-Copy: Operand dependency is rebuilt to preserve the LLVM-IR program validity. Since LLVM-IR is strongly typed, the constant value 1.0 is used if no other value in the requested type is available.

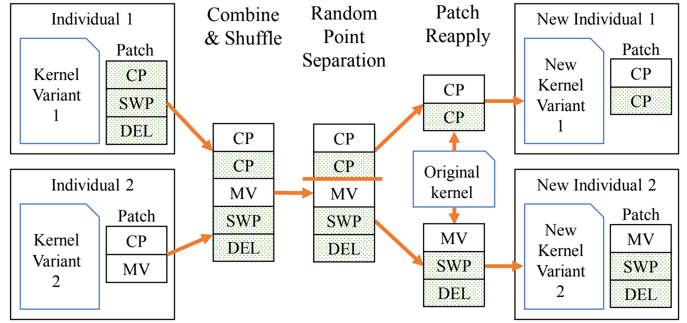

Fig. 4: One-point messy crossover.

other individuals in its Pareto front, and these two values are combined to produce a single fitness value for each kernel variant. See [36] for details. Finally, NSGA-II uses tournament selection based on this single fitness value for selecting kernel variants for the next generation (Line 6 of Algorithm 1).

GEVO uses elitism, retaining the top quarter of the population in the next population generation (Lines 7, 16, and 17 of Algorithm 1).

D. 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 3) can be only assigned once at creation, our mutations are likely to create invalid programs by breaking the SSA constraint. Thus, we introduce an extra repair step. As shown in Figure 3, 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.

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 3 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.

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 to weed out invalid modifications (sanity check). 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%.

E. 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 [37] and variable-length [38] crossover operations. Figure 4 illustrates the 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 GPU kernel and generates two new individuals. This form of crossover was selected because it generates a wide diversity of recombinations.

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.

IV. Experimental Setup

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

A. Infrastructure and Configurations

We developed GEVO on top of an existing python framework for evolutionary algorithms, called DEAP [39], by implementing the genetic operators described in Section III [1]. We instrument the LLVM compiler (LLVM 8.0) to implement the mutation operations in C++ as a LLVM pass. For each optimization, GEVO was given a 48-hour budget. We evaluate GEVO using NVIDIA Tesla p100 GPUs, under CUDA 9.2 and NVIDIA GPU driver 410. The Nvidia profiler (nvprof) was used to collect kernel execution time, which became the runtime metric used by the fitness function. In our experiments nvprof introduces no overhead to kernel execution time, and the measurement varies less than 1%.

All GEVO experiments were conducted with population size of 256, crossover rate of 80% (i.e., 80% of individuals in

[1]Code is available at https://github.com/lioujheyu/cuda_evolve

TABLE I: Benchmarks used for evaluation.

Application	Abbr.	GPU Kernel Line of LLVM-IR
Breadth first Search	bfs	72
B+Tree	b+t	168
CFD Euler3D	cfd	1079
Gaussian Elimination	gau	186
Heart Wall	hw	3806
Hotspot	hot	189
LU decomposition	lud	2491
Nearest Neighbor	nn	32
Needleman-Wunsch	nw	715
Particlefilter	pf	1442
Pathfinder	path	109
SRAD_v2	sv2	446
Stream Cluster	sc	231
Handwriting recognition (C=5, g=0.05)	mnist	256
Income prediction (C=32, g=0.0078125)	a9a	256

population are selected for crossover), and a mutation rate of 30% (i.e., every individual has 30% chance to get one mutation). The 48-hour budget for each GEVO run translates into a variable number of generations, 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 20 to over 100. For example, for the NN benchmark, GEVO spent the majority of its time searching for valid mutations and was able to complete only 20 generations within 48 hours.

B. Benchmarks

To assess the general applicability of GEVO, we first evaluate GEVO with the Rodinia benchmark suite [40]. Rodinia covers a wide range of general-purpose deterministic workloads for heterogeneous computing, representing diverse parallel communication patterns, synchronization techniques and power consumption. Table I summarizes the Rodinia applications.

We validate of 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 Rodinia built-in input generator. Depend on benchmark, each input set contains from tens of thousands to millions of input values. Each test is running under the original, unmodified kernel and using its output 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 that exceed the 1% error tolerance (GEVO-mO). After evolution, we validate the best kernel variant found by GEVO with an unused test having input data generated through the above process, to make sure the GEVO does not overfit the kernel to the existed test cases during the evolution.

In addition to the general-purpose GPU workloads, we consider a machine learning program. Because machine learning (ML) algorithms are intrinsically error-tolerant and there is significant time overhead for training (e.g., the training time of state-of-the-art language translation models is on the order of days [41]), ML models are a natural application for exploring accuracy/efficiency tradeoffs. Because GEVO searches the optimization space at the granularity of instructions, it requires full

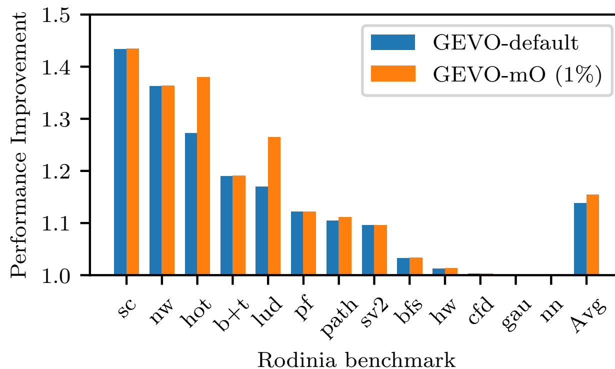

Fig. 5: Normalized performance improvement over the default baseline with full compilation optimization for GEVO-default and GEVO-mo (1%) in the Rodinia Benchmark. (For example, the 1.38X improvement in hot reduces runtime from 1.07 seconds to 1.07/1.38 = 0.77 seconds.)

access to the intermediate representation and the corresponding optimized library implementations. This consideration ruled out using the NVIDIA cuDNN-based ML framework and led us to a supervised ML framework (ThunderSVM) [42]. ThunderSVM is a support vector machine (SVM) library with an open-source, optimized implementation for GPUs.

We evaluate GEVO on ThunderSVM using two datasets: handwriting recognition using MNIST [2] and income prediction using a9a[3]. We downloaded the datasets from libsvm [43]'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, MNIST large dataset, which contains 8,000,000 image samples, is solely used as a 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 V-B. Earlier work used evolutionary computation to evolve neural network architectures [44], [45], but this is the first work we know of to evolve the code that implements the neural network itself.

C. Error Metric

For the Rodinia benchmarks, error represents the maximum difference between outputs 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 the SVM, we consider the runtime to train the model and the accuracy of the trained model's performance on its testing data. We use two-fold cross validation on the training dataset to report the error for GEVO to optimize. The testing dataset is used only for reporting the testing error. Similar to the Rodinia benchmark suite, kernel variants are rejected if training error rates exceed the training error from original kernel plus 1%.

V. Evaluation Results and Analysis

We first present the empirical results across all workloads in the first 2 subsections, for the Rodinia benchmark and

Machine learning workloads. To understand the types of kernel optimization techniques discovered by GEVO, Section V-A takes a closer look at some recurring optimizations found by GEVO, for the Rodinia applications, and Section V-B considers the machine learning workloads. Section V-C, uses `Hotspot` as an example to show how GEVO explores the optimization space and converges to an optimized kernel variant that satisfies the fitness function.

A. Rodinia Benchmark

Figure 5 reports the overall performance improvement for GEVO-default, GEVO-mO (1%), by comparing to the default baseline with full compilation optimization for the Rodinia benchmarks. GEVO-default improves the performance of the Rodinia benchmark suite by an average of 13.87% and by as much as 43.41% for `sc`. When relaxing the accuracy constraint on kernel outputs, GEVO-mO scavenges additional improvements, reducing run-time by an average of 15.47% over the baseline. As figure 5 shows, there is significant variability in the achieved improvements among programs, including three, `cfd` ,`gau` and `nn`, for which we found no improvements. There are several possible explanations for this variability, e.g., it could be a feature of the program itself or perhaps the program was perfectly optimized by the original programmer, but this variability is consistent with results reported using similar methods on assembly programs to reduce energy, e.g., [20].

GEVO discovered many different optimizations in the evolved Rodinia Benchmarks. We categorize them into 2 types, which are architecture-, and application-specific optimization.

1) Architecture-specific optimization:

- Removing synchronization primitives: This was the most common optimization found by GEVO. Although in general it is risky to remove synchronizations, some syncthread() calls in CUDA can be removed because of the thread scheduler in the GPU hardware provides redundant synchronizations under particular memory access patterns. We find this optimization in `hot`, `lud`, and `nw`.
- Removing redundant stores: In `lud`, GEVO removed a store instruction which updates a value in shared memory that is later read by other threads. This change doesn't alter the program behavior, because in this case the store instruction is not protected by a synchronization primitive. We speculate that the GPU implicitly changes the load-store order by issuing the load instruction without waiting for the outstanding store instruction to complete, rendering the store instruction redundant.

2) Application-specific optimizations:

- Removing conditional execution: GEVO could eliminate code blocks in the conditional path entirely if the input space does not touch that portion of the kernel, which it found for `hot`, `lud` and `pf`.
- Loop perforation: Loop perforation is a technique from approximate computing that skips iterations of a loop based on the *skip factor* [46]. GEVO discovers similar optimizations, for example, when loops have been unrolled heavily post-compilation. GEVO then removes some part(s) of the unrolled loop while optimizing the fitness function. We observed this behavior in `sc`, `lud`, and `hot`.
- Memoization: Memoization stores results of expensive function calls and returns the stored value without re-computation when the same inputs reappear. GEVO sometimes identifies similar memoization opportunities by eliminating unneeded instructions and using stored results directly, e.g., in the HotSpot temperature modeling tool (`hot`). HotSpot performs some pre-processing based on the physical dimension of the processor chip. Since the shape of simulated chips is the same across all loop iterations, GEVO discovers memoization opportunities to reuse the pre-processing results of the x-dimension for the y-dimension.

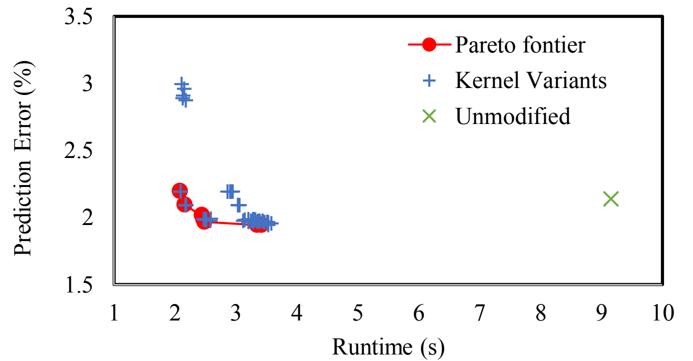

Fig. 6: The Pareto-frontiers and other kernel variant measurements for the handwriting recognition (ThunderSVM with MNIST).

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.

B. ThunderSVM

Machine learning (ML) is a popular class of intrinsically error-tolerant applications, which can consume large computational resources, and is particularly suitable for the GEVO approach. We consider one ML example and use it to illustrate how performance and accuracy can be co-optimized. Although earlier work examined accuracy/energy tradeoffs, we are not aware of any earlier work targeting genetic improvement of ML LLVM-IR kernels.

Figure 6 shows the Pareto frontier for `MNIST`. The x-axis represents the measured training time and the y-axis represents the testing inference prediction error in %. We report results for each kernel variant in GEVO's final generation, relative to the original unmodified kernel. Figure 6 also shows how GEVO-mO navigates away from the original, sub-optimal, kernel implementation and explores the better performing part of the search space.

Through manually selecting the kernel variant in the Pareto frontier representing the best combined improvement, we find that GEVO-mO achieves 3.24X and 2.93X performance improvement for handwriting recognition (MNIST) and income

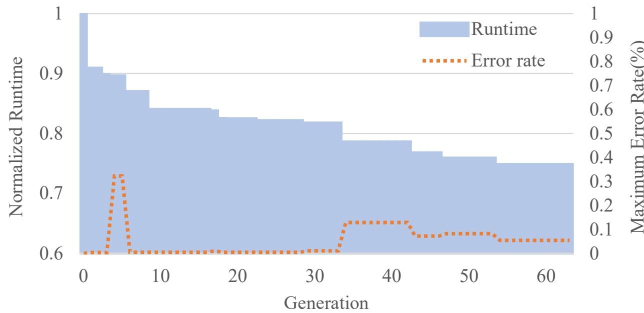

Fig. 7: Temporal evolution of a `hot` kernel variant.

prediction(a9a)'s kernel performance, which increases overall model training speed by 50% and 24.8% respectively, with accuracy on the training set improving from 97.86% to 98.03% (MNIST) and from 84.61% to 84.65% (a9a). Next, we test the trained GEVO-optimized model on the official testing dataset, where we find accuracy is improved slightly, from 98.37% to 98.5% (MNIST) and from 84.59% to 84.64% (a9a).

Finally, we consider whether the ThunderSVM 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 thunderSVM optimized for the MNIST common dataset (60,000 samples) by using it to train the large MNIST dataset (8,000,000 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 1182 and 121 minutes.

When we applied GEVO to SVM for MNIST, it found some epistatic (interacting) mutations. Our preliminary analysis of these mutations suggests that GEVO's optimization changed the termination condition of a while loop, causing it to exit sooner. This loop implements a 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 reduce the solution correctness. However, for MINST, this change actually improves model accuracy. We leave further analysis of this result for future work.

C. Temporal Evolution of `hot`

Figure 7 show the evolution of a GEVO optimization for `hot`. 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 (*compensatory evolution*). 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 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.

VI. Discussion and Conclusion

This paper presents a Genetic Programming approach using population-based search to find optimizations of GPGPU kernels that fulfill required specifications of program behavior. The proposed approach trades off absolute program semantics for other important non-functional design aspects. We 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 absolute correctness requirements (e.g., avionics software or some systems programs), is suitable for many other applications, including the important class of machine learning codes. Our results show that GEVO explores the optimization search space for the handwriting recognition and the income prediction machine learning workloads, finding multiple points along the Pareto frontier that typically trades off performance and accuracy. In some cases, however, GEVO finds a significant 3.28x speedup of the handwriting recognition kernel (ThunderSVM with `MNIST` with modest improved prediction accuracy. This translates to 50% training time reduction with 0.17% improvement on the prediction accuracy, reflecting absolute improvements in both dimensions.

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 these results. GEVO found both application-specific architecture-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. 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. 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 attack. We hope the idea and insights presented in this paper inspire other applications and optimizations of software and systems in the years to come.

Acknowledgements

We thank F. Esponda, W. Weimer, and E. Schulte for many insights, code and helpful comments. This work is supported in

part by the National Science Foundation under CCF-1618039 and SHF-1652132; and AFRL FA8750-17-S-7007.

References

[1] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *Proc. of ACM SIGARCH Computer Architecture News*, 2013.

[2] Y. Le Cun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. of the IEEE*, 1998.

[3] J. C. Platt, “Advances in kernel methods,” B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds., 1999, ch. Fast Training of Support Vector Machines Using Sequential Minimal Optimization.

[4] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” in *Proc. of the 34th Int. Conf. on Software Engineering*, 2012.

[5] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *Proc. of 3rd Intl. Conf. on Software Testing, Verification and Validation*, 2010.

[6] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Transactions on Software Engineering*, 2012.

[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *Proc. of the 31st Intl. Conf. on Software Engineering*, 2009.

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

[9] Facebook, “Finding and fixing software bugs automatically with sapfix and sapienz,” https://code.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/, 2018.

[10] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, “Fixing bugs in your sleep: How genetic improvement became an overnight success,” in *Proc. of the Genetic and Evolutionary Computation Conference Companion*, 2017.

[11] E. Schulte, J. DiLorenzo, S. Forrest, and W. Weimer, “Automated repair of binary and assembly programs for cooperating embedded devices,” in *Proc. of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[12] E. M. Schulte, W. Weimer, and S. Forrest, “Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair,” in *Proc. of the 1st Genetic Improvement Workshop*, 2015.

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

[14] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, “Automatic software diversity in the light of test suites,” *arXiv preprint arXiv:1509.00144*, 2015.

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

[16] S. O. Haraldsson, J. R. Woodward, Alexander, E. Brownlee, A. V. Smith, and V. Gudnason, “Genetic improvement of runtime and its fitness landscape in a bioinformatics application,” in *Proc. of the Genetic and Evolutionary Computation Conference Companion*, 2017.

[17] N. Veerapen, F. Daolio, and G. Ochoa, “Modelling genetic improvement landscapes with local optima networks,” in *Proc. of the Genetic and Evolutionary Computation Conference Companion*, 2017.

[18] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, “Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation,” *IEEE Transactions on Software Engineering*, 2014.

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

[20] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, “Post-compiler software optimization for reducing energy,” in *Proc. of the 19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[21] N. Burles, E. Bowles, A. E. Brownlee, Z. A. Kocsis, J. Swan, and N. Veerapen, “Object-oriented genetic improvement for improved energy consumption in google guava,” in *Proc. of Intl. Symposium on Search Based Software Engineering*, 2015.

[22] I. Manotas, L. Pollock, and J. Clause, “Seeds: a software engineer’s energy-optimization decision support framework,” in *Proc. of the 36th Intl. Conf. on Software Engineering*, 2014.

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

[24] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, “Automatically exploring tradeoffs between software output fidelity and energy costs,” *ACM Transactions on Software Engineering*, p. to appear, 2017.

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

[26] W. B. Langdon and M. Harman, “Evolving a cuda kernel from an nvidia template,” in *Proc. of IEEE Congress on Evolutionary Computation*, 2010.

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

[28] W. B. Langdon and M. Harman, “Genetically improved cuda c++ software,” in *Proc. of 17th European Conference on Genetic Programming*, 2014.

[29] W. B. Langdon and M. Harman, “Grow and graft a better cuda pknotsrg for rna pseudoknot free energy calculation,” in *Proc. of the Companion Publication of the 17th Annual Conference on Genetic and Evolutionary Computation*, 2015.

[30] E. Schulte, “Neutral networks of real-world programs and their application to automated software evolution,” Ph.D. dissertation, University of New Mexico, Albuquerque, USA, 2014.

[31] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, “Gpucc: An open-source gpgpu compiler,” in *Proc. of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO ’16, 2016.

[32] H. Massalin, “Superoptimizer: A look at the smallest program,” in *Proc. of the 2nd Intl. Conf. on Architectual Support for Programming Languages and Operating Systems*, 1987.

[33] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic optimization of floating-point programs with tunable precision,” *ACM SIGPLAN Notices*, 2014.

[34] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, “Conditionally correct superoptimization,” in *Proc. of ACM SIGPLAN Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[35] B. Churchill, R. Sharma, J. Bastien, and A. Aiken, “Sound loop superoptimization for google native client,” *SIGARCH Comput. Archit. News*, 2017.

[36] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Transactions on Evolutionary Computation*, 2002.

[37] F. J. Burkowski, “Shuffle crossover and mutual information,” in *Proc. of the 1999 Congress on Evolutionary Computation-CEC99*, 1999.

[38] C.-Y. Lee and E. K. Antonsson, “Variable length genomes for evolutionary algorithms,” in *Proc. of 2nd Annual Conf. on the Genetic and Evolutionary Computation Conference*, 2000.

[39] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, “Deap: A python framework for evolutionary algorithms,” in *Proc. of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, 2012.

[40] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proc. of IEEE Intl. Symposium on Workload Characterization*, 2009.

[41] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, “Applied machine learning at facebook: A datacenter infrastructure perspective,” in *Proc. of IEEE Intl. Symposium on High Performance Computer Architecture*, 2018.

[42] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, “ThunderSVM: A fast SVM library on GPUs and CPUs,” *Journal of Machine Learning Research*, 2018.

[43] C.-C. Chang and C.-J. Lin, “Libsvm: A library for support vector machines,” *ACM Trans. Intell. Syst. Technol.*, 2011.

[44] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, “Large-scale evolution of image classifiers,” *arXiv preprint arXiv:1703.01041*, 2017.

[45] B. Zoph and Q. Le, “Neural architecture search with reinforcement learning,” in *Proc. of Intl. Conf. on Learning Representations*, 2017.

[46] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, “Managing performance vs. accuracy trade-offs with loop perforation,” in *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.