GEVO-ML: A Proposal for Optimizing ML Code with Evolutionary Computation

Jhe-Yu Liou
ASU
jhe-yu.liou@asu.edu

Xiaodong Wang
Facebook
xdwang@fb.com

Stephanie Forrest
ASU
steph@asu.edu

Carole-Jean Wu
ASU/Facebook
carole-jean.wu@asu.edu

ABSTRACT

Parallel accelerators, such as GPUs, are a key enabler of large-scale Machine Learning (ML) applications. However, programmers often lack detailed knowledge of the underlying architecture and fail to fully leverage their computational power. This paper proposes GEVO-ML, a tool for automatically discovering optimization opportunities and tuning the performance of ML kernels. GEVO-ML extends earlier work on GEVO (Gpu optimization using EVOlutionary computation) by focusing directly on ML frameworks, intermediate languages, and target architectures. It retains the multi-objective evolutionary search developed for GEVO, which searches for edits to GPU code compiled to LLVM-IR and improves performance on desired criteria while retaining required functionality. In earlier work, we studied some ML workloads in GPU settings and found that GEVO could improve kernel speeds by factors ranging from 1.7X to 2.9X, even with access to only a small portion of the overall ML framework. This workshop paper examines the limitations and constraints of GEVO for ML workloads and discusses our GEVO-ML design, which we are currently implementing.

CCS CONCEPTS

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

KEYWORDS

Genetic Improvement, Multi-objective Evolutionary Computation, Machine Learning

ACM Reference Format:
Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO-ML: A Proposal for Optimizing ML Code with Evolutionary Computation. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3377929.3398139

1 INTRODUCTION

Machine learning (ML) applications are being deployed at unprecedented scales across a wide variety of domains. These applications are enabled by advances in many research domains, from hardware accelerators like Graphic Processing Unit (GPU) and Tensor Processing Unit (TPU), framework infrastructures like PyTorch [47] or TensorFlow [2], improved ML algorithms and network architectures. To achieve better performance, meaning faster training/inference time and/or higher model accuracy, application developers fine-tune many of these design components to achieve the best configuration, and they often use evolutionary computation (EC) to find these improvements [64]. For example, in network architecture, developers have to decide how many layers will be in the network, how many neurons will be in each layer, etc [65]. Similarly, in Support Vector Machine (SVM), the designer must select a cost value [25]. And finally, at the framework level, developers have to select the operators to be used in the model they want and allocate their available system resources such as CPU or GPU for running the models.

As the scope of artificial neural network has grown, the raw number of tunable knobs has exploded, and consequently, most designers use empirical methods to identify a set of parameters that works well for their situation. Many research projects also use automated parameter tuning methods, or hyperparameter search, to find model parameters [7, 66]. These include simple grid search [32], random search [7], reinforcement learning [6, 74], evolutionary computation [64], and gradient descent [40]. At the framework level, developers have introduced ML compilers, which find optimizations such as operator fusion [4], or at code generation time can determine the degree of loop unrolling or loop tiling based on the hardware platform characteristics [12, 50]. These various searches and optimizations occur at different levels and are usually performed separately, as developers are not always the master of all worlds. In addition, there are a number of other framework-level features, such as threading libraries and scheduling policies that can be used to further optimize ML training and inference execution time [3, 20, 69].

Typically, developers leave low-level code optimization to compilers, which for most applications are hard to beat. However, codes running on GPUs often have inefficiencies that arise because of interactions between the application and the underlying architecture. Unless the developer has unusually detailed knowledge of the architecture, it is challenging to uncover these additional optimization opportunities. In earlier work, we showed that EC search can find many interesting optimization opportunities, yielding an average of 49% speedup on common parallel benchmarks [39]. These speedups are achieved by relaxing the usual compiler restriction that preserves exact program semantics. This approach is thus well-suited for approximate computing applications such as ML. Thus, we propose to use EC to optimize the GPU kernels that implement ML algorithms, by extending our earlier work on a tool called GEVO (**G**PU **EVO**lution) [37–39]. GEVO can be thought of as a compiler

GECCO '20 Companion, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7127-8/20/07…$15.00
https://doi.org/10.1145/3377929.3398139

post-pass performance tuning framework, which 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 intermediate representation (LLVM-IR).

Although the GEVO approach is quite general in the sense that it can be applied to any GPU program represented in LLVM-IR, it is well-suited to ML workloads because they are computationally intensive, and by design they are error-tolerant. GEVO implements multi-objective search, so it can help manage tradeoffs between model accuracy and training time [24, 26, 53, 55, 73].

In this workshop paper, we first review related research and describe GEVO's design and implementation. Next, we outline the proposed GEVO-ML design, highlighting opportunities for further optimizations and discussing various implementation strategies and constraints. Finally, we summarize some of the ML optimizations that GEVO has already discovered [39] to provide context and motivation for the current work.

2 RELATED WORK

Over the past decade, Evolutionary Computation (EC) methods have been applied to the task of improving computer programs, particularly the task of automatically repairing bugs in legacy software [15, 19, 21, 35, 70], with some industrial applications [17, 23]. Typically, the repair methods operate at the source-code level using abstract syntax trees, but similar methods have also been used to repair assembly programs [57] and even object code [60].

A surprising feature of software was revealed by these projects: Random mutations of code often have no observable functional effect on program behavior [5, 8, 22, 59, 67]. These *neutral* mutations are plentiful, occurring $20-40\%$ of the time, even when mutations are focused only on the sections of code covered by the tests. Some neutral mutations are semantically equivalent to the original, like *equivalent mutations* in mutation testing [42], but many others produce semantically similar programs, some of which may repair the bug and others which can satisfy required functionality in slightly different ways from the original. These results led researchers to consider optimizing non-functional properties of software such as energy by finding neutral mutations that satisfy required functionality and improve the non-functional property [58]. GEVO follows in this tradition, using multi-objective search to find improvements of non-functional software properties, in particular, focusing on reducing run-time. This approach is a form of super-optimization, one which can scale to much larger code sizes than existing super-optimization methods [43].

GEVO operates on GPU code, and there has been some prior research applying EC to GPU kernels. For example, Sitthi-Amorn et al. optimized a graphics shader program, beginning with a basic lighting algorithm and evolving it into a form resembling an advanced algorithm [61]. Langdon et al. tackled CUDA runtimes for two target programs [30, 31] by representing the program object as a custom-designed, line-based Backus Normal Form (BNF) grammar. In contrast, GEVO applies to any CUDA program with minimal manual intervention and uses modern Clang/LLVM tooling.

Clang/LLVM is a popular compiler infrastructure, but there is only one earlier work we are aware of that has attempted to apply EC to programs in the LLVM intermediate representation (LLVM-IR) [56]. LLVM-IR is challenging because random mutations often break the Single Static Assignment discipline and must be repaired. Schulte's work gave a proof-of-concept for how to approach this challenge, and GEVO extends the basic approach to provide a robust implementation. To summarize, GEVO modifies CUDA programs in the LLVM-IR, as shown in the left half of Figure 1. This avoids developing novel parsing and syntax manipulating infrastructure of some earlier work, but requires special handling of mutation and recombination.

While GEVO can be applied to any computer program, Machine Learning (ML) is an appealing target for several reasons: ML applications have become ubiquitous, they have high computational cost, and they often run on specialized hardware such as GPU or TPU. Of course, the field of neuro-evolution focused on evolving neural networks for many years [45], typically using evolution to improve network architecture [51, 64], module design [63, 68], hyperparameters[52], and to learn weights[46]. GEVO adds to these capabilities by finding optimizations in the low-level codes that implement NN algorithms. And, in some cases it finds synergies between low-level optimizations and modifications at higher levels of the framework.

Today's ML programming frameworks represent deep learning models as computational graphs of various types of operators. Domain-specific compilers identify optimization opportunities such as operator fusion or tiling. Examples include: XLA [1] for TensorFlow [2]; Glow [54] for PyTorch [47]; and TVM [12] for MXnet [11]. Further optimizations can be achieved by translating high-level NN operators to machine-specific implementations using optimized libraries. These optimization techniques preserve the functional behavior of a given neural network. A recent proposal, called TASO [27], uses superoptimization methods to optimize the computational graph of a deep neural network. Essentially TASO enumerates the possible combinations of operator implementations and selects the graph implementation that minimizes runtime. A SAT solver is used to ensure that the original graph's functionality is preserved. Although promising, this approach currently does not scale well beyond small graphs comprised of more than four operators. Our approach to optimizing NNs complements this earlier work, finding joint optimization opportunities that involve (1) better-performing operator implementations and (2) modifying neural network architectures.

3 BACKGROUND: GEVO

GEVO is an EC tool for automatically improving kernel implementations targeting GPUs [37–39]. GEVO enables GPU code optimization using EVOlutionary computation. In this section, we briefly review GEVO's design, highlighting its representation, genetic operators, fitness function, and selection method. A complete description of the most recent version appears in [39].

When CUDA code is compiled, the kernels that will run on the GPU are separated from the host code and compiled into the intermediate representation (LLVM-IR). GEVO operates on the IR, applying genetic operators to transform the program and passing the transformed code back to clang to complete the compilation process. The binary is then run on a set of user-defined test cases to evaluate functionality and performance.

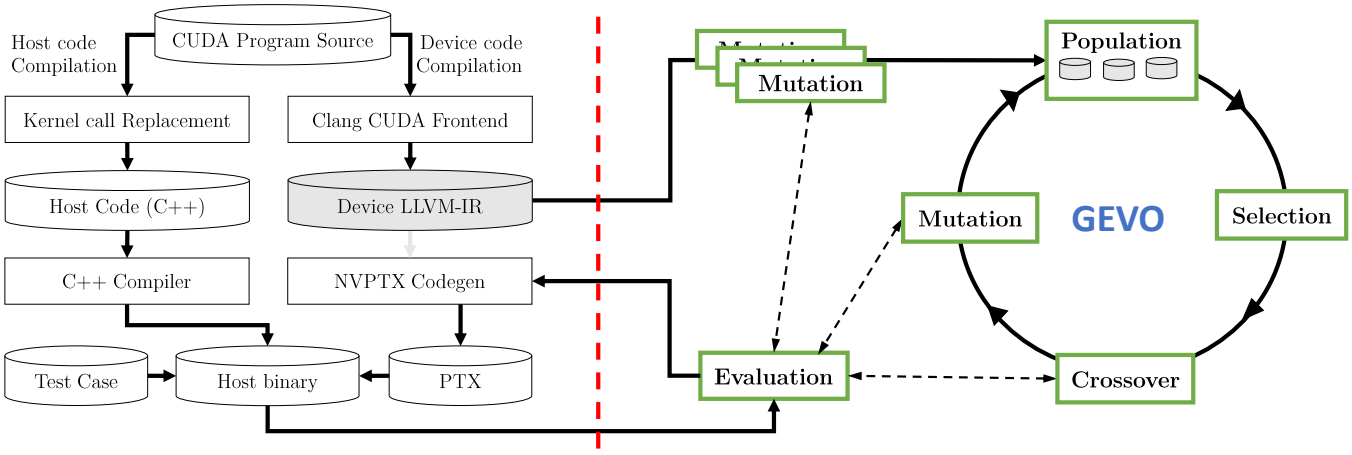

Figure 1: GEVO in the LLVM/Clang CUDA compilation flow. The left half is the CUDA code compilation process in clang/llvm. The right half shows how GEVO searches the code optimization for the device code from the compilation, and how the modified device code is evaluated.

The population is initialized with PopSize individuals that are formed by taking the original program, making copies and applying random mutations to each. By default, three mutations are applied to each individual in the initial generation, providing diversity. The search then uses multi-objective GEVO and 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 GPU optimization.

Representation. Each individual in GEVO has two representations, one of which is the code itself (the *program-based* representation) and one which is simply a list of the edits (mutations) to be applied to the program (the *patch-based* representation) [35]. This design decision relates to the many data dependencies built into the LLVM-IR, which complicates mutation (see below) and requires certain repair processes each time a mutation is applied. These are expensive, so it is more efficient to use the patch-based representation for crossover. As the number of mutations applied to any kernel variant (individual) is typically low, and because the kernels themselves tend to be small, this design choice does not greatly impact the memory requirement of GEVO.

Mutation. GEVO uses several mutation operators, each of which modifies a line of LLVM-IR code: Copy (copy an instruction from one location to another location); Delete (move an instruction from one location to another); Replace (one instruction or operand with another); and Swap (instructions). The LLVM-IR uses Static Single Assignment (SSA), which means that a variable can be assigned only once at the time it is created. The mutation operators are highly likely to create invalid programs by violating this restriction. To address this problem, GEVO has a fairly complex 'repair' step built into its mutation operation, described in detail in [39]. As one example, a Copy mutation can lead to a type mismatch in one of the copied instruction's operands. To repair this, GEVO first looks for another in-scope variable of the proper type and substitutes that if one is found, or it simply replaces the offending variable with a constant, e.g., 1.0. Unlike most EC applications, a significant portion of GEVO's search budget is spent finding valid mutations.

When the mutation operator is invoked, one mutation type is selected randomly (with equal probability) and applied (with repairs) as an edit to generate a new kernel variant. After each mutation is applied, GEVO immediately evaluates the individual to check if it still passes all the test cases. If it fails, another mutation is tried, and this process continues until a valid kernel variant is found.

Crossover: GEVO uses the patch-based representation (list of edits) for crossover, because it is highly likely that recombining two random program slices would require additional repairs to create a valid individual. GEVO uses one-point messy crossover, which combines shuffle [9] and variable-length [36] crossover operations. GEVO begins with two randomly selected individuals, concatenates the two lists of mutations (edits) in the patch representation; shuffles the sequence; and then randomly selects a location to cut the list back into two. GEVO then reapplies each patch in sequence to the original GPU kernel, and generating two new individuals. Although unusual, this strategy produces a wide diversity of recombinations from a minimal number of mutations, since mutations are relatively expensive. Each new individual is then evaluated to test if the new combination of edits is valid, and we find that about 80% of the time they are. If not, GEVO repeats the process until it finds a successful recombination.

Fitness Evaluation. For most applications, GEVO requires that all individuals produce identical output as the test cases, and only individuals that meet that criterion have fitness assessed according to the non-functional objective(s) selected by the user. Many applications, however, including ML, can tolerate some error in the output. For these *approximate* applications, GEVO requires only

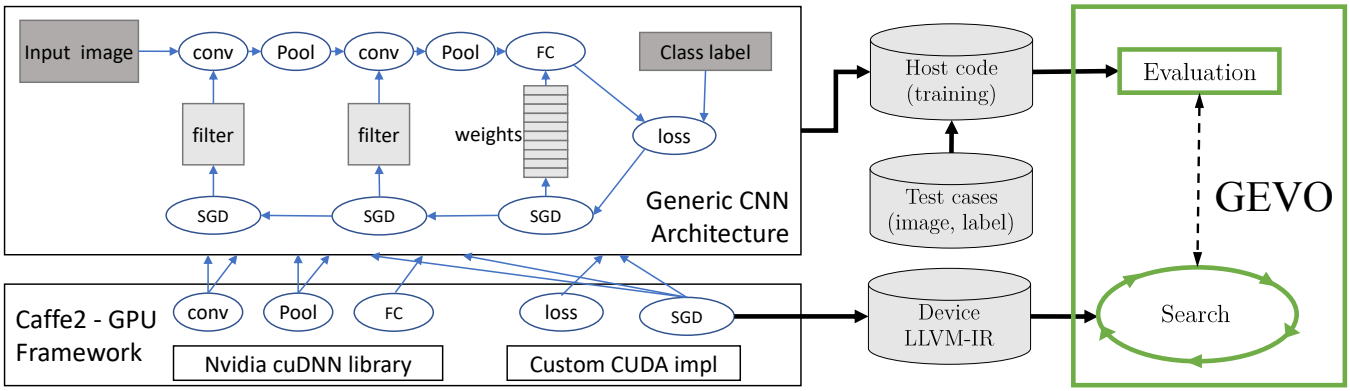

Figure 2: How GEVO searches for the optimization with a generic CNN model built on top of the Caffe2 framework

that the morphed program executes successfully, and the output error becomes one of the optimization objectives.

For ML, the two objectives become reduced kernel execution time and ML model error, i.e., $argmin(time, error)$. Given an ML model that has been modified by GEVO, fitness is evaluated by retraining the model on a given training dataset and recording the training time and model error. At the end of the search, the most fit individual is evaluated against the testing dataset and that value is reported.

Selection. GEVO uses the Non-dominated Sorting Genetic Algorithm (NSGA-II) [14] to select individuals according to multi-objective fitness criteria and reports the pareto frontier of individuals that best satisfy the two objectives. GEVO retains the top quarter of the population at time t and copies it unchanged to the population at time $t + 1$. It then chooses the remaining 3/4 of the population using tournament selection.

Implementation. We developed GEVO using DEAP [13], by implementing the genetic operators described above in C++. We instrumented the LLVM compiler (LLVM 8.0) so the mutation operations, written in C++ appear as a LLVM pass. All GEVO experiments to date have been run on NVIDIA Tesla p100 GPUs, under CUDA 9.2 and NVIDIA GPU driver 410. The Nvidia profiler (nvprof) collects kernel execution time, that is, 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%.

4 GEVO-ML

As aforementioned, GEVO optimizes runtime and minimizes error of LLVM-IR codes running on GPUs [37, 39]. Previous experimental evaluation focused primarily on the general-purpose GPU code, but included experiments on ML workloads (summarized in Section 5). These early results were encouraging, but they revealed several limitations of GEVO with respect to optimizing ML workloads. In the following, we discuss these limitations and how they are addressed in the design of GEVO-ML, an EC framework for enhancing the performance of ML code.

In ML models, there may be diverse optimization opportunities for the same operator, depending on the context in which the operator is executed. That is, in principle, GEVO could optimize ML

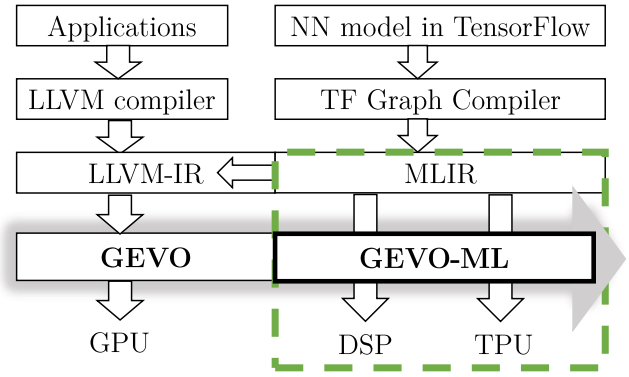

Figure 3: The current GEVO system stack in application compilation and execution, and our plan to extend our design to other accelerators like DSP and TPU through integrating deeply with ML framework with mutation on MLIR.

operators that appear in different layers of the ML model, each of which has a different purpose or configuration. For example, in Figure 2, the Stochastic Gradient Descent (SGD) operator is used to update the parameters in several ways. The convolution layers can each have a different filter size, producing a different number of weights for SGD to update. The current workflow of GEVO prevents it from discovering these nuanced opportunities because it doesn't distinguish the different contexts in which the operator is used. With access to these contexts, GEVO-ML can, for example, modify how many parameters are passed into SGD, but the degree of weight pruning is likely different in the different cases because the number of parameters is not identical in the different layers.

A second issue concerns the hardware for accelerating ML workloads. Many new hardware platforms are emerging, particularly for the edge. For instance, the digital signal processors (DSPs) shipped on many mobile devices can be used to accelerate ML applications [49, 72], especially when the GPU is busy with other tasks such as rendering the device screen. A second example is the Tensor Processing Unit (TPU), an accelerator dedicated to ML workloads [28]. The challenge here, given a neural network model composed of a set of operators, is to generate optimized code for each operator and to

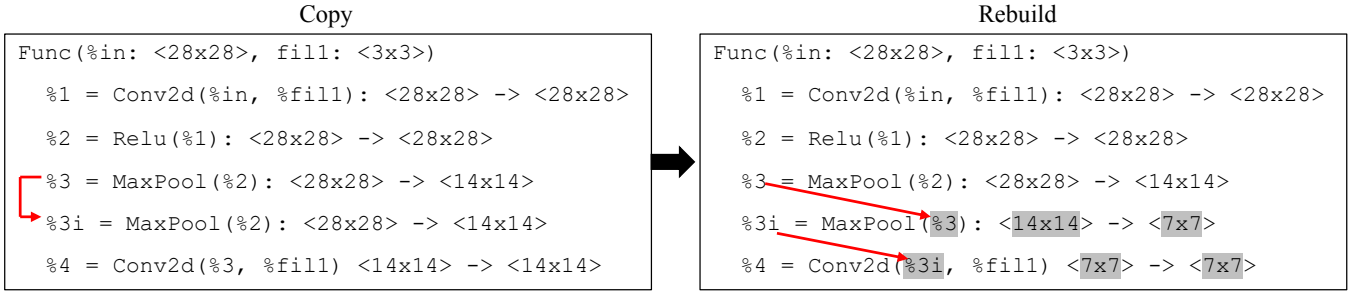

Figure 4: A mutation example in a MLIR program representing part of a neural network model in TensorFlow. Each MLIR instruction also specifies the shape of input/output tensor such as <7x7>. In this example, instruction 3 is copied and inserted right after itself. Besides rerouting the operand, the tensor shape must also be configured as the instruction sequence changes.

schedule the execution order of the operators for different hardware architectures. The proposed design GEVO-ML, then, must accommodate a much broader set of target architectures. It is currently unknown which, if any, optimizations that GEVO-ML finds for one architecture will generalize to others. A more practical problem is that, for today's ML workloads, most operators are implemented on Nvidia cuDNN library. These operators cannot be optimized by GEVO because their source codes are not available. Although 'just an implementation' issue, this restriction severely limited our earlier experiments.

To address these concerns, GEVO-ML is designed to be tightly integrated with ML frameworks so that it can optimize ML models directly. Figure 3 illustrates the overall design, which will leverage the Multi-Level IR (MLIR) [33]. MLIR provides a formal representation for machine learning models so that existing compiler optimizations can be applied [12, 50]. This makes MLIR an ideal target for GEVO-ML. The design of MLIR has many features shared or inherited from LLVM-IR. In fact, it has recently been incorporated into LLVM as a sub-project for domain-specific intermediate representation [41], with the functionality to convert MLIR to LLVM-IR. GEVO-ML will extend mutation operations to support MLIR.

Figure 4 shows an example for the Copy mutation, as it can be implemented in MLIR representing a TensorFlow model. This will allow GEVO-ML to discover optimizations in both the model structure and its underlying implementation. Why might this be useful? As just one example, Figure 5 illustrates an optimization that GEVO found in our preliminary work which leverages a synergy between changes at the model layer and the implementation to improve performance. Finally, as a proof-of-concept prototype, we plan to integrate GEVO-ML into TensorFlow for two reasons. First, TensorFlow is currently the only ML framework that supports MLIR. In addition, multiple platforms are supported by TensorFlow, including GPU, DSP and TPU. This broadens the application of the proposed GEVO-ML.

Figure 2 illustrates the current approach of GEVO to optimizing NNs in Caffe2. As the figure shows, the only input to the search (mapped through the LLVM-IR box) is the SGD kernel, and the rest of the NN model is used only for the purpose of evaluation. Under GEVO-ML, however, the input to the search will be the entire neural network architecture as represented in MLIR.

The GEVO-ML design allows us to address three other shortcomings of GEVO.

- **Operator fusion:** Today's ML frameworks typically have a set of rules for optimizing a given model by combining multiple operators into a single one [4]. This reduces the overhead incurred by operator invocations to the dedicated hardware and the overhead required to transfer intermediate results between the operators. In principle, the existing approach of GEVO could discover a partial form of this optimization by moving the internal implementation of one operator to another operator, but it cannot completely eliminate the extra invocation. In GEVO-ML, the entire model architecture is exposed to the evolutionary process, so GEVO-ML can remove the operators after merging.
- **Optimization during training:** By integrating with the ML framework, GEVO-ML can be configured as part of the training process. Under this scenario, code is modified on a per-epoch basis or on a per-batch evaluation similarly to how weights are updated or how learning rates can be scheduled differently for different epochs [62].
- **Automation:** GEVO had to be manually configured to find optimizations in the most important operators used in the model. Because GEVO-ML has access to the entire model architecture in the MLIR representation, the evolutionary process of the neural network implementation in the MLIR representation as a whole can be automated naturally.

5 WHY ARE WE OPTIMISTIC ABOUT GEVO-ML?

Why do we think that a low-level approach like GEVO-ML might be able to find additional improvements in already highly optimized NN codes? Why do we think that GEVO-ML is likely synergistic optimizations that combine low-level code changes with higher-level algorithm or hyperparameter changes? In this section, we briefly review our earlier ML results. Refer to [39] for detailed descriptions of the experimental procedures and evaluations.

Support Vector Machines (SVM). In one set of experiments, we targeted the supervised ML framework, ThunderSVM [71], which is an SVM library that is fully open-sourced and optimized for GPU implementation. We used two popular datasets taken from [10],

handwriting recognition using MNIST [34] and income prediction using a9a [16, 48]. In our setup, we gave GEVO a 48-hour search budget and asked it to optimize the `c_smo_solve` kernel for both training time and inference prediction accuracy of the trained model, using two-fold cross validation. We rejected any solutions that introduced greater than 1% additional error beyond that achieved by the baseline implementation.

Although GEVO reported pareto optimal values for several runtime/error tradeoffs, the best combined improvement for MNIST ran 3.24X faster than the baseline with a slight improvement in accuracy. Similarly, the income prediction (a9a)'s experiment led to a 2.93X speedup with a very slight improvement in model accuracy. These results were surprising because we expected to find tradeoffs between speed and accuracy. Because GEVO was only optimizing one component of the model, these results translated to an overall improvement in model training speed of 50% and 24.8% respectively. When we tested the models that were learned using GEVO-optimized code on their official test datasets, accuracy was slightly improved, ranging from 98.37% to 98.5% (MNIST) and from 84.59% to 84.64% (a9a).

We hypothesize that an SVM that is optimized for training a specific dataset might 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. To address this, we also tested the SVM optimized for the MNIST common dataset (60,000 samples) by using it to train the large MNIST dataset (8,000,000 handwriting samples). Using 10-fold cross validation, we found that accuracy from the unmodified ThunderSVM was 100% and our GEVO-optimized training model produced a network that achieved 99.997% accuracy.

How did GEVO achieve these optimizations? In essence, GEVO discovered that for the MNIST dataset it could relax the convergence condition in the SVM solver. Surprisingly, for MNIST this change actually improves model accuracy as well as performance, perhaps by avoiding overfitting.

ResNet18. Although many deep learning frameworks, like TensorFlow [2], PyTorch [47], and Caffe2 [18], rely on closed source libraries, and are thus unavailable to GEVO, Caffe2 contains a module that is custom implemented as a CUDA kernel and open sourced within Caffe2 source repository. Thus, we were able to use GEVO to optimize stochastic gradient descent with the momentum (momentumSGD) code in Caffe2. We used an 18-layer residual neural network (referred to as ResNet18) for image classification on the CIFAR-10 dataset [29], which contains 50000 training and 10000 testing images. MomentumSGD updates the weights and bias for different layers of the neural network by evaluating the difference between the true label and predicted label. This experiment was computationally intensive, so we only trained the model for three epochs, and even then, GEVO was only able to run for 20 generations when we allowed up to 10% error. Even with these restrictions, GEVO found a kernel that was 1.79X faster than the original one. Since the kernel constitutes less than 1% of the entire training time, these gains don't immediately translate into impressive overall improvements in model training time. Of more interest, however, are the optimizations that GEVO found, as Figure 5 shows. In this

```
1  /*    N = number of parameters
2   * m[i] = momentum
3   * g[i] = gradient
4   * BETA = momentum decay rate
5   *   LR = learning rate
6   */
7  for (i=tid; i<N; i+= GRID_SIZE N) {
8    float mi = m[i];
9    float mi_new = BETA*mi + LR*g[i];
10   m[i] = mi_new LR*g[i];
11   g[i] = (1+BETA)*mi_new - BETA*mi;
12
13 - if (param)
14     param[i] -= g[i];
15 }
```

Figure 5: Code snippet from the Caffe2 momentumSGD operator illustrates two optimizations discovered by GEVO.

example, three changes are responsible for the accuracy and performance improvements: (1) Terminate the loop which updates the parameters early, similar to weight pruning [44]. (2) Change the algorithm of momentum by preserving the most recent momentum but discarding the rest. (3) A low level code optimization through removing the unnecessary branch condition.

We would like to point this out again, as Figure 2 shows. The SGD kernel can be used in various layers with different number of parameters needed to be updated. In fact, as there are 18 layers in ResNet18, 18 instances of momentumSGD are invoked for updating parameters of the layers. Different degrees of weight pruning might further improve the model accuracy and performance, which leads to GEVO-ML design.

6 CONCLUSION

Deep learning applications today are often developed using a complex deep learning framework, which is compiled to run on complex proprietary architectures that lack transparency. This often leads to unanticipated interactions with runtime environments and workloads. GEVO-ML contributes a new dimension to neuro-evolution by tackling this complexity in a general way. However, there are many complexities in the emerging ML framework implementations which complicate the implementation of GEVO-ML. The paper argues, however, that these are tractable.

In our earlier work with GEVO, we showed that EC can find application-specific, architecture-specific, and dataset-specific optimizations, and in some cases optimizations at different layers combine in synergistic ways. Sometimes the optimizations are as simple as removing a redundant synchronization call, and sometimes they exploit nuanced interactions between an algorithm and the dataset it runs on, as we saw in the SVM example. By moving to a general MLIR code representation, where GEVO-ML will have access to the full ML framework, we look forward to harnessing the power of EC to find even more impressive performance improvements in the future.

ACKNOWLEDGMENTS

We thank F. Esponda, W. Weimer, and E. Schulte for many insights, code and helpful comments. The authors gratefully acknowledge the partial support of the National Science Foundation (CCF-1618039, SHF-1652132, and CCF 1908633); DARPA (FA8750-15-C-0118); AFRL (FA8750-19-1-0501); and the Santa Fe Institute for Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu at ASU.

REFERENCES

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

[2] 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 *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*.

[3] P Anju. 2018. Tips to Improve Performance for Popular Deep Learning Frameworks on CPUs. *Intel Developer Zone* (2018).

[4] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On Optimizing Machine Learning Workloads via Kernel Fusion. In *Proceedings of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 173–182. https://doi.org/10.1145/2688500.2688521

[5] 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).

[6] 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).

[7] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[8] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement. In *Proc. of the 17th Annual Conf. on Genetic and Evolutionary Computation*.

[9] Forbes J Burkowski. 1999. Shuffle crossover and mutual information. In *Proc. of the 1999 Congress on Evolutionary Computation-CEC99*.

[10] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* (2011).

[11] 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).

[12] 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 *Proc. of 13th {USENIX} Symp. on Operating Systems Design and Implementation*.

[13] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: A Python Framework for Evolutionary Algorithms. In *Proc. of the 14th Annual Conf. Companion on Genetic and Evolutionary Computation*.

[14] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* (2002).

[15] Vidroha Debroy and W Eric Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Proc. of 3rd Intl. Conf. on Software Testing, Verification and Validation*.

[16] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[17] Facebook. 2018. 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).

[18] Facebook. 2019. Caffe2. (2019). https://caffe2.ai/.

[19] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proc. of the 11th Annual Conf. on Genetic and Evolutionary Computation*.

[20] Google. 2019. TensorFlow Performance Guide. https://docs.w3cub.com/tensorflow~guide/performance/performance_guide/#general_best_practices. (2019). TensorFlow Documentation.

[21] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* (2012).

[22] 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 *Proc. of the Genetic and Evolutionary Computation Conf. Companion*.

[23] 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 *Proc. of the Genetic and Evolutionary Computation Conf. Companion*.

[24] 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 *Proc. of IEEE Intl. Symp. on High Performance Computer Architecture*.

[25] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. 2003. A practical guide to support vector classification. (2003).

[26] 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 *Proc. of the IEEE Conf. on computer vision and pattern recognition*.

[27] 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 *Proc. of the 27th ACM Symp. on Operating Systems Principles (SOSP '19)*.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[29] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

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

[31] William B. Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *Proc. of the 17th Annual Conf. on Genetic and Evolutionary Computation*.

[32] 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 *Proc. of the 24th Intl. Conf. on Machine Learning*.

[33] Chris Lattner and Jacques Pienaar. 2019. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law. (2019).

[34] Yann Le Cun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. of the IEEE* (1998).

[35] 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 *Proc. of the 34th Int. Conf. on Software Engineering*.

[36] C.-Y. Lee and E. K. Antonsson. 2000. Variable Length Genomes for Evolutionary Algorithms. In *Proc. of 2nd Annual Conf. on the Genetic and Evolutionary Computation Conf.*

[37] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Genetic Improvement of GPU Code. In *Proc. of the 6th Intl. Workshop on Genetic Improvement (GI '19)*. Best paper award.

[38] Jhe-Yu Liou, Stephanie Forrest, and Carole-Jean Wu. 2019. Uncovering Performance Opportunities by Relaxing Program Semantics of GPGPU Kernels. Wild and Crazy Idea session at the 24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. (2019).

[39] Jhe-Yu Liou, Xiaodong Wang, Stephanie Forrest, and Carole-Jean Wu. 2020. GEVO: GPU Code Optimization using EvolutionaryComputation. (2020). arXiv:cs.NE/2004.08140

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

[41] LLVM. 2020. Multi-Level IR Compiler Framework. (2020). https://mlir.llvm.org/.

[42] 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 Transactions on Software Engineering* (2014).

[43] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proc. of the 2nd Intl. Conf. on Architectual Support for Programming Languages and Operating Systems*.

[44] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *Proc. of Intl. Conf. on Learning Representations*.

[45] David J Montana and Lawrence Davis. 1989. Training Feedforward Neural Networks Using Genetic Algorithms.. In *IJCAI*.

[46] Gregory Morse and Kenneth O. Stanley. 2016. Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16)*. Association for Computing Machinery, New York, NY, USA, 477–484. https://doi.org/10.1145/2908812.2908916

[47] 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. (2017).

[48] John C. Platt. 1999. Advances in Kernel Methods. Chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization.

[49] Qualcomm. 2016. Snapdragon Neural Processing Engine SDK. (2016). https://developer.qualcomm.com/docs/snpe/overview.html.

[50] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* (2013).

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

[52] 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 *Proc. of the 34th Intl. Conf. on Machine Learning-Volume 70*. JMLR. org.

[53] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*.

[54] 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).

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

[56] Eric Schulte. 2014. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution.* Ph.D. Dissertation. University of New Mexico, Albuquerque, USA.

[57] Eric Schulte, Jonathan DiLorenzo, Stephanie Forrest, and Westley Weimer. 2013. 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*.

[58] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. 2014. Post-compiler Software Optimization for Reducing Energy. In *Proc. of the 19th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[59] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software Mutational Robustness. *Genetic Programming and Evolvable Machines* (2014).

[60] 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 *Proc. of the 1st Genetic Improvement Workshop*.

[61] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic Programming for Shader Simplification. In *Proc. of the 2011 SIGGRAPH Asia Conf.*

[62] Leslie N Smith. 2017. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 464–472.

[63] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life* (2009).

[64] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127.

[65] D Stathakis. 2009. How many hidden layers and nodes? *International Journal of Remote Sensing* 30, 8 (2009), 2133–2147.

[66] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proc. of the 19th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD '13)*.

[67] Nadarajen Veerapen, Fabio Daolio, and Gabriela Ochoa. 2017. Modelling genetic improvement landscapes with local optima networks. In *Proc. of the Genetic and Evolutionary Computation Conf. Companion*.

[68] Phillip Verbancsics and Kenneth O Stanley. 2011. Constraining connectivity to encourage modularity in HyperNEAT. In *Proc. of the 13th annual Conf. on Genetic and evolutionary computation*. ACM.

[69] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. 2019. Exploiting Parallelism Opportunities with Deep Learning Frameworks. *arXiv preprint arXiv:1908.04705* (2019).

[70] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proc. of the 31st Intl. Conf. on Software Engineering*.

[71] Zeyi Wen, Jiashuai Shi, Qinbin Li, Bingsheng He, and Jian Chen. 2018. ThunderSVM: A Fast SVM Library on GPUs and CPUs. *Journal of Machine Learning Research* (2018).

[72] Carole-Jean Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344.

[73] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *Advances in neural information processing systems*. 685–693.

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