Using FPGA Devices to Accelerate Tree-Based Genetic Programming: A Preliminary Exploration with Recent Technologies*

Christopher Crary[0000−0002−4953−9344], Wesley Piard[0000−0001−6581−2200], Greg Stitt[0000−0001−7159−7439], Caleb Bean[0000−0002−3635−0315], and Benjamin Hicks[0000−0001−7669−1598]

University of Florida, Gainesville, FL 32611, USA

Abstract. In this paper, we explore the prospect of accelerating tree-based genetic programming (TGP) by way of modern field-programmable gate array (FPGA) devices, which is motivated by the fact that FPGAs can sometimes leverage larger amounts of data/function parallelism, as well as better energy efficiency, when compared to general-purpose CPU/GPU systems. In our preliminary study, we introduce a fixed-depth, tree-based architecture capable of evaluating type-consistent primitives that can be fully unrolled and pipelined. The current primitive constraints preclude arbitrary control structures, but they allow for entire programs to be evaluated every clock cycle. Using a variety of floating-point primitives and random programs, we compare to the recent TensorGP tool executing on a modern 8nm GPU, and we show that our accelerator implemented on a 14nm FPGA achieves an average speedup of 43×. When compared to the popular baseline tool DEAP executing across all cores of a 2-socket, 28-core (56-thread), 14nm CPU server, our accelerator achieves an average speedup of 4,902×. Finally, when compared to the recent state-of-the-art tool Operon executing on the same 2-processor CPU system, our accelerator executes about 2.4× slower on average. Despite not achieving an average speedup over every tool tested, our single-FPGA accelerator is the fastest in several instances, and we describe five future extensions that could allow for a 32–144× speedup over our current design as well as allow for larger program depths/sizes. Overall, we estimate that a future version of our accelerator will constitute a state-of-the-art GP system for many applications.

Keywords: Tree-based genetic programming · Field-programmable gate arrays · Hardware acceleration

1 Introduction

During any given time, the development of AI has been constrained and influenced by the computing technologies available [9, 10, 27]. Nevertheless, novel

* This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1718033 and CCF-1909244.

applications of pre-existing technologies still happen, and they can drive research fields in whole new directions, occasionally prompting some form(s) of widespread adoption. For instance, the massive adoption of deep neural networks in the past decade was clearly enabled by developments regarding GPUs [9, 10, 27]. In this instance, computing advancements markedly extended the practical reach of neural networks, which then kickstarted a wave of popularity and research ventures. Importantly, in this domain and many others, the demands of ever-increasing performance and energy efficiency has now manifested into the broad development of *domain-specific hardware accelerators* [9]. However, in the wide-ranging domain of genetic programming (GP), there seems to exist only a few instances of specialized hardware accelerators [7, 8, 15, 16, 22], which is especially surprising given that GP is an "embarrassingly parallel" procedure.

Generally speaking, although general-purpose CPU/GPU systems can be made to effectively exploit some of the parallelism opportunities inherent to GP (e.g., by evaluating multiple data points, multiple operations, or multiple candidate solutions in parallel), the frequent, dynamic changes in control flow caused by GP (e.g., when evaluating different operations within a single program) generally limits how effective a general-purpose computing platform can perform [3, 4, 21]. Within this paper, we focus on the original tree-based GP (TGP) [12], and we explore how we may overcome the aforementioned limitations of CPU/GPU systems by way of an accelerator specialized to the evaluation phase of TGP, implemented with a modern field-programmable gate array (FPGA). In brief, FPGAs are programmable computing systems in which specialized digital circuitry can be synthesized from different levels of abstraction, without recourse to integrated circuit development.

Overall, as depicted in Fig. 1, our preliminary accelerator leverages a specialized, full tree of generic computing resources to compute any program relevant to a GP primitive set, as long as the depth of the program is not larger than the depth of the tree, the latter of which is defined by the user. By then pipelining the generic resources, the accelerator can generate an output for an entire program expression *every clock cycle* after some initial latency. To further increase throughput, the accelerator also dynamically compiles programs for the tree while evaluating, so that the tree may switch between programs *within a single clock cycle*. Importantly, such forms of parallelism have not been achieved via general-purpose CPU/GPU architectures.

We compare the performance of our architecture with the evaluation engines given by three actively maintained, open-source tree-based GP software tools: *DEAP* [6], *TensorGP* [1], and *Operon* [3].[1] From each tool, we use the evaluation engine—and no evolution engine—to execute a large set of randomly generated programs for various amounts of fitness cases (i.e., sample points), and we estimate evaluation performance in terms of *node evaluations per second* (*NEPS*). For each software-based tool, we utilize a dual-socket server populated with two 2.6 GHz (3.7 GHz Turbo), 14-core (28 thread), 14nm Intel Xeon Gold 6132 CPU packages, and we additionally use an 8nm Nvidia RTX 3080 GPU (10 GB)

[1] Software: https://github.com/christophercrary/conference-eurogp-2023.

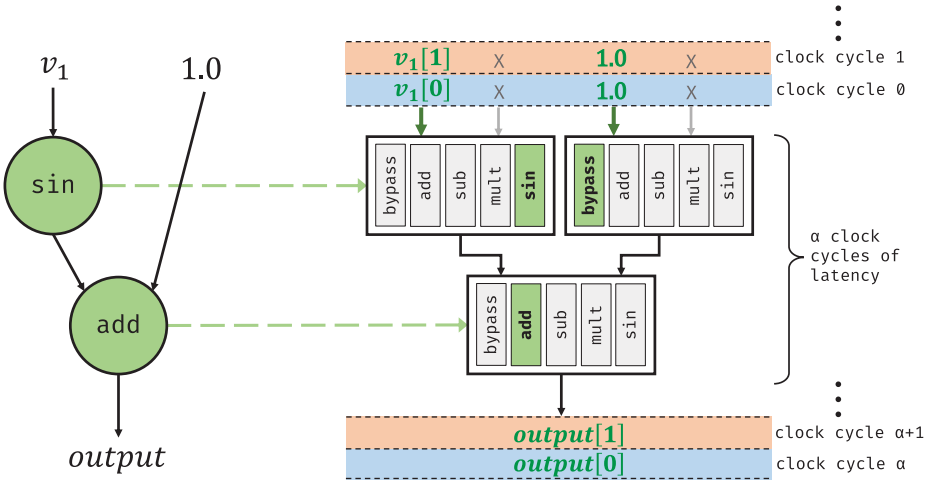


Fig. 1: A portrayal of how our GP accelerator can parallelize the evaluation of different data points and different solutions *every clock cycle* via a reconfigurable tree pipeline. Each node of the pipeline can perform any function within the GP primitive set, as well as a bypass, which allows for arbitrary program shapes.

for TensorGP. To implement our hardware accelerator, we utilize a 14nm Intel Stratix 10 SX 1SX280HN2F43E2VG FPGA provided by an Intel Programmable Acceleration Card (PAC) through the Intel FPGA DevCloud service. We compile the accelerator by way of Quartus Pro 19.2.0, Build 57.

When compared to DEAP [6], a popular baseline for GP software tools, our accelerator achieves an average speedup of 4,902×. Compared to TensorGP [1], a recent general-purpose GP software tool targeting both CPU and GPU systems, our architecture achieves an average speedup of 61.5× in regard to CPU execution and 43× in regard to GPU execution. Finally, when compared to Operon [3], a recent state-of-the-art GP tool tailored to symbolic regression [13], our single-FPGA accelerator executes about 2.4× slower on average when compared to the same 2-processor CPU system, although there are several instances in which our accelerator performs the fastest. Despite not achieving an average speedup over every tool tested, we describe five future extensions that could allow for a 32–144× speedup over our current design. Separately, we note that it has been widely shown that FPGAs can often provide power and energy improvements when compared to CPU/GPU systems, sometimes by multiple orders of magnitude [18, 20, 23, 24]. Although we do not provide power or energy estimates in this paper, if we can experience any of such improvements when compared to other GP tools, this should enable us to implement more energy-efficient (and, thus, potentially more cost-effective) GP systems than what has been presented in previous work [25]. Overall, we estimate that a future version of our accelerator will constitute a state-of-the-art GP system for many applications.

The remainder of the paper is organized as follows. Section 2 describes related work. Section 3 details our architecture. Section 4 describes our design of experiments. Section 5 presents results for our experiments. Section 6 discusses limitations of our current architecture and planned future extensions that should address the limitations and allow for state-of-the-art performance. Finally, Section 7 presents conclusions.

2 Related Work

In the context of CPU/GPU systems, there exist numerous works that discuss mechanisms for accelerating tree-based GP—see [5] for a recent review, as well as [1–4, 14, 21]—although there are comparatively few works that consider the use of FPGA devices [7, 22]. Compared to prior work [7, 22], our accelerator has several important contributions. Most significantly, our system dynamically compiles programs from a compressed prefix notation into configuration data for a reconfigurable pipeline, whereas previous work used a simpler, less flexible mechanism by which larger, fixed-size programs must be compiled. Ultimately, our compressed prefix notation allows for significantly reduced communication times as well as significantly reduced size requirements for on-chip RAM. Also, with the ability to dynamically compile arbitrary expressions directly on the target device, future extensions of our design can accelerate other GP stages without continued hardware/software communication. Besides dynamic compilation, we also explore the use of a higher-end FPGA device, multiple primitive sets, a range of fitness case amounts, different tree sizes, and 32-bit floating point, all while comparing to a range of modern GP tools.

Apart from tree-based GP, there exists some prior work on the FPGA acceleration of certain GP variants, e.g., Linear GP [15], Cartesian GP [16], and Geometric Semantic GP [8], although we note that the differences in evaluation schemes warrants a separate architecture dedicated to tree-based GP. Lastly, we note that the application area of evolvable hardware [28] has also leveraged FPGA devices, although this has been with the primary intention of evolving circuits, rather than accelerating the GP procedure via a single circuit.

3 Accelerator Architecture

In this section, we detail our accelerator architecture for the evaluation phase of tree-based GP. We focus on evaluation since it is the primary bottleneck for TGP. Eventually, we will investigate acceleration of the entire GP process.

The accelerator architecture currently consists of four major components, as shown in Figure 2. The *program memory* (Section 3.1) stores candidate program solutions, where each candidate is encoded in a language defined by the specification of a particular primitive set. The *program compiler* (Section 3.2) reads program expressions from program memory and dynamically compiles them into configuration information for the *program evaluator*, which we implement as a reconfigurable *function tree pipeline* (Section 3.3). This function tree pipeline

executes a compiled expression for all relevant fitness cases, resulting in a new output for the entire program *every clock cycle* after some initial latency. Finally, the *fitness evaluator* (Fig. 2d) compares the output of a current program to the relevant target data by way of some metric (*root-mean-square error* in this paper), which allows for other stages of GP to optimize the individual.

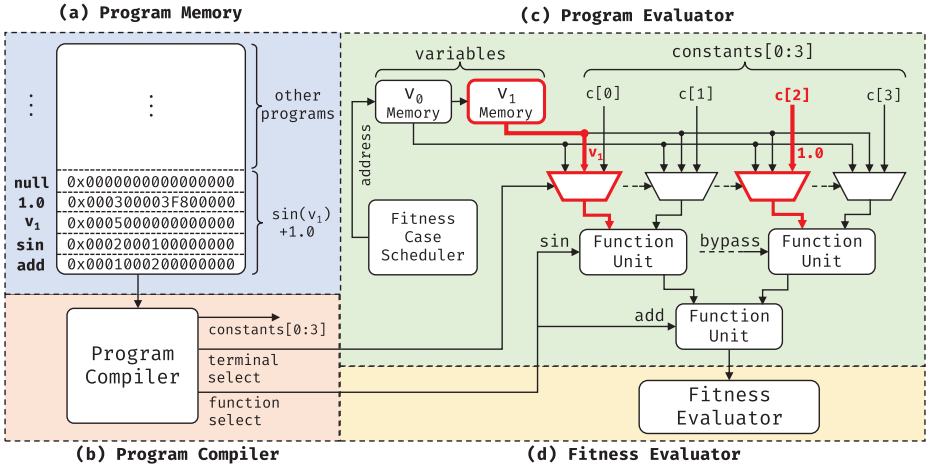


Fig. 2: High-level overview of the accelerator architecture. The accelerator stores programs (e.g., $\sin(v_1) + 1.0$) in (a) *program memory*, which are dynamically compiled by (b) *the program compiler* into configuration data for (c) *the program evaluator*. The program evaluator uses a reconfigurable function tree pipeline to execute a compiled expression for a set of fitness cases, resulting in a set of outputs to which (d) *the fitness evaluator* compares a set of desired outputs.

3.1 Program Memory

The architecture currently implements program memory with on-chip RAM resources and memory-mapped I/O. For a primitive set $P = F \cup V \cup C$, with function set F, variable terminal set V, and a set of 32-bit constant terminals C (e.g., all single-precision floating-point values), we define a 64-bit machine code for program nodes as follows:

1. The most-significant 16 bits of the machine code represent an opcode which specifies either the type of primitive or the *null word*, the latter of which is used to indicate the end of a program expression within memory. The null word is assigned opcode 0, each function is assigned an opcode in the range $[1, |F|]$, each constant is assigned opcode $|F| + 1$, and each variable is assigned an opcode in the range $[|F| + 1 + 1, |F| + 1 + |V|]$.
2. The least-significant 32 bits of the machine code specify a constant value, which is only relevant if the opcode indicates that the node is a constant.

3. The remaining 16 bits specify the depth of a node within the context of a program, which is relevant to the program compiler (Section 3.2).

We encode program expressions via a prefix (i.e., Polish) notation. In essence, such a representation flattens tree-based programs into a linear structure [19]. For example, Fig. 2a shows how our architecture could support the program $\sin(v_1) + 1.0$ by way of a simple primitive set consisting of addition (+), sine (sin), two variable terminals (v_0 and v_1), and the set of all single-precision floats.

3.2 Program Compiler

The program compiler reads program expressions from program memory and dynamically compiles them into configuration information for the program evaluator. Currently, the program compiler is implemented as a finite-state machine (FSM) that continually writes configuration information into a configuration buffer, which is omitted from Fig. 2 due to space constraints. Such buffering enables the program compiler to generate configurations for a program in advance while the program evaluator is processing fitness cases for an existing program.

The program compiler's configuration data contains three major components (Fig. 2b,c): 1) *function select* values that configure individual function units within the function tree of the program evaluator (Section 3.3), *terminal select* values that dictate whether a variable or constant terminal is connected to the corresponding function tree input, and 3) *constant* values that specify the bits of any constant terminals feeding into the tree.

To compile a program, the program compiler conducts a pre-order traversal on a model of the relevant function tree, so that compilation can happen in parallel to program evaluation. We determine a model for the tree at compile time, based on the specified depth and branching factor of the tree, the latter of which is determined by the maximum function arity of the chosen primitive set.

Ultimately, depending on the shapes/sizes of programs being compiled and the number of fitness cases that are to be streamed into a function tree, the cost of compiling a program may be completely amortized such that there is no dead cycles in between evaluating consecutive programs. Fortunately, for *any* function tree structure, there will always be some threshold for the number of fitness cases such that, for any number of fitness cases above this threshold, compilation will be completely amortized. Separately, since the program compiler FSM needs relatively few resources (currently, less than 2% of all area for our target device), we can extend our architecture to support multiple compiler instantiations. With this ability, multiple programs could be compiled in parallel—perhaps to effectively support multiple function trees, or perhaps to ensure that the cost of compiling a single tree can be completely amortized. For the experiments in this paper, we support the compilation of one program at a time, and we incorporate a multiple-buffering approach, following the above.

3.3 Program Evaluator

The program evaluator (Fig. 2c) is a reconfigurable *function tree* that serves two purposes: 1) provide configurable resources that enable the program compiler to implement arbitrary expressions specified by program memory; and 2) provide a pipeline that enables streaming of fitness case data such that program outputs can be computed *every clock cycle*.

The motivation behind the function tree is that, with tree-based GP, every program expression is represented as a tree. Therefore, if the accelerator provides a function tree containing pipelined generic resources capable of computing the functions of the relevant primitive set (i.e., a *function unit*), then the function tree can produce outputs for entire program expressions *every clock cycle* after some initial latency. In this paper, we consider a single tree structure, but we plan to support multiple tree structures in future work.

For the program evaluator, the user must specify the relevant primitive set and the depth of the underlying function tree, which define 1) the maximum function arity, 2) the operations supported by each function unit, and 3) the possible program shapes/sizes. A function tree with depth d can compute arbitrary programs that adhere to both 1) a maximum depth of $d + 1$—where the extra level accounts for terminal nodes—and 2) the syntax of the relevant primitive set. To be able to implement any program not represented by a full tree, a special bypass function is used to feed the leftmost input of a function unit directly to its output whenever that node within the tree is not to be used by a program. In regard to function primitives, we currently support any form of computation that can be unrolled and pipelined.

In addition to a function tree, the program evaluator also contains *variable memories*, which support variable terminals. The variable memories store fitness cases for every feature of the relevant training data. The particular data for each variable can be set at runtime, using memory-mapped I/O.

4 Design of Experiments

In this section, we detail our design of experiments, where the overall goal of these experiments was to compare our architecture (Section 3) with the core evaluation engines given by three tree-based GP software tools: *DEAP* [6], *TensorGP* [1], and *Operon* [3]. The computing technologies we used are listed in Section 1.

4.1 Comparison Metrics

We estimate and compare *median node evaluations per second* (*NEPS*)[2] values for each evaluation engine in the context of different combinations of program sizes, numbers of fitness cases, and primitive sets. For each software tool and for

[2] Frequently, the statistic of *GP operations per second* (*GPops*) is used when comparing the runtime performance of GP tools, but we use NEPS to emphasize that our runtimes do not include time taken for evolution.

each combination of parameters (detailed below), we conducted an experiment in which we evaluated 32 *program bins*, each consisting of 512 distinct random programs, and we estimated a median runtime for each program bin by measuring a certain number of evaluation runtimes and then taking the sample median. With a sample median runtime, we calculated an estimate for the true median NEPS value by dividing the total number of node evaluations for an experiment by the sample median runtime. Due to time constraints and significant performance differences between each of the GP tools, we used a different total number of executions for some tools when calculating sample median runtime. For Operon and TensorGP, the two fastest software tools, we ran the set of experiments 11 times. However, for DEAP, in which the set of experiments executed in about 44 hours (due to poor scaling at larger numbers of fitness cases), we ran each experiment just once. Running each DEAP experiment once seemed justified by the fact that any fluctuations in runtime due to other system processes were likely insignificant when compared to the processes used by the experiments, as indicated by the narrow $75^{\text{th}}/25^{\text{th}}$ percentile regions for the runtimes of TensorGP, given below. Lastly, we note that it was unnecessary for the accelerator experiments to be run more than once, since the circuitry created for the system had deterministic behavior.

4.2 Primitive Sets

Three distinct primitive sets were chosen. These primitive sets were inspired by recent work from Nicolau et al. [17], and, as such, were respectively named nicolau_a, nicolau_b, and nicolau_c. The first primitive set contained functions with the self-explanatory names add, sub, and mul, as well as a function by the name of aq, for "analytical quotient," defined by $\text{aq}(x_1, x_2) = x_1/\sqrt{1 + x_2^2}$, which is meant to behave similarly to divide, but without the asymptotic conditions at zero [17]. The second primitive set contained the same functions as the first, but also included sin and tanh. Lastly, the third primitive set contained the same functions as the second, but also included exp, log, and sqrt, where log and sqrt were "protected" in the typical GP sense [12, 19]. We chose these specific primitive sets since they are relevant to symbolic regression [12, 13, 19], our primary target domain.

For a primitive set containing function set F, $|F| - 1$ terminal variables and one ephemeral random constant were employed so that the program generator (Section 4.3) would consistently construct programs in which the proportion of functions/terminals was approximately 0.5, so that the average runtime of a particular primitive set was not dictated by having more of one primitive type.

4.3 Program Generation

For each primitive set, a set of 32 program bins was constructed, each containing 512 random programs with sizes in some fixed range, where the particular range was dependent on the bin and primitive set, as described further below. The maximum possible program depth/size was chosen to be the largest that the

target FPGA could support while also supporting up to 100,000 fitness cases for each of the relevant variable terminal memories (Section 3.3). To determine these values, the maximum possible function tree depth for each primitive set was manually determined through multiple hardware compilations—ultimately, depth values 8, 6, and 6 were respectively chosen for nicolau_a, nicolau_b, and nicolau_c. For a maximum possible function tree depth d, it was possible to support a program depth of up to $d + 1$ (Section 3.3), which corresponded to a maximum possible program size of $2^{d+2} - 1$, since every primitive set contained functions with arity of at most two. For a maximum size s, the range of program sizes $[1, s]$ was subdivided into 32 bins.

To randomly generate program expressions for each set of bins—which were kept the same for each GP tool—we utilized DEAP [6]. We chose DEAP for this task because it was simple to extend. DEAP offered, by default, several classic GP program initialization algorithms: *full*, *grow*, and *ramped half-and-half* [12, 19]. Unfortunately, via the original version of these algorithms, the size of a generated program was completely random beyond a specified depth constraint, which made it too cumbersome to generate 512 distinct random programs for the bin structures established above. To circumvent this issue, we created a modified version of the *grow* method that allowed for the specification of a minimum/maximum program size, from which a random value was chosen in a uniform manner. Overall, choosing 512 distinct random programs for each bin structure meant that 16,384 programs were used to evaluate each of the three primitive sets, which corresponded to a total of 49,152 random programs.

4.4 Fitness Cases

For each primitive set, we used five amounts of fitness cases: 10, 100, 1,000, 10,000, and 100,000. For each number of fitness cases, we randomly generated input/target data in the range [0, 1), and we used the same data for each of the evaluation engines. We note that using random data should elucidate the fact that our performance results are relevant to *any* GP application that can utilize the 1) chosen primitive sets, 2) maximum number of variables, and 3) maximum number of fitness cases, which, as shown in [13], allows for many.

5 Results

Figure 3 compares the performance of each evaluation engine in terms of sample median NEPS values, for six of the fifteen combinations of primitive set and number of fitness cases. For each combination, we plot results for five GP tool setups: 1) DEAP, 2) TensorGP with CPU, 3) TensorGP with GPU, 4) Operon, and 5) our FPGA-based hardware accelerator. More specifically, for each plot representing a GP tool, a sample median NEPS value is marked for each program bin containing 512 programs, with the particular number of fitness cases used for each program changing between sub-figures. In addition, for the tools in which experiments were run more than once (i.e., TensorGP and Operon), the 75^{th} and

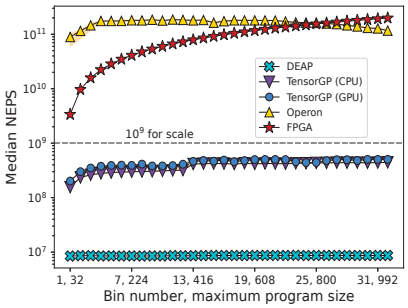
25^{th} percentiles for runtime are plotted above/below each sample point; only a few of such percentile regions are noticeable, meaning that most runtimes vary little between multiple runs. Overall, due to space constraints, we include plots for just six experiments, but we chose these six in particular since they best conveyed the most important general trends for our accelerator, detailed below.

Overall, our accelerator mostly performed second-best behind Operon, but in several instances our accelerator obtained the highest performance, e.g., for the larger programs and larger number of fitness cases with the nicolau_a primitive set (Fig. 3a,b), and for the smaller programs and smaller numbers of fitness cases across all primitive sets (e.g., Fig. 3c,e). In some other instances, our accelerator performed very similarly to Operon, e.g., for the medium-sized programs and medium-sized numbers of fitness cases with nicolau_b (Fig. 3d). In general, the speedups we achieved stemmed from the fact that our accelerator had *constant throughput* once programs were compiled for the program tree. For larger numbers of fitness cases (e.g., 10K and 100K), compilation was completely amortized after the first program (Section 3.2), which allowed for maximal throughput. Interestingly, although the program tree structures for primitive sets nicolau_b and nicolau_c utilized the same depths/sizes—which should potentially allow for identical runtime—the hardware synthesis tool had to utilize a lower clock frequency for nicolau_c in order to support more complex primitives, which allowed for nicolau_b to have better performance. A similar discrepancy in clock frequency also explains why Fig. 3a lists better performance than Fig. 3b.

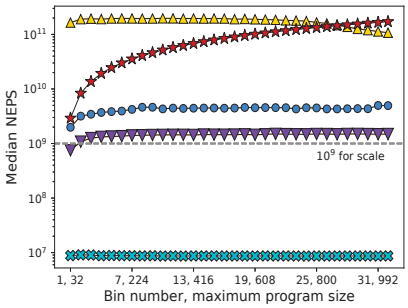
Average FPGA Speedup				
Fitness Case Threshold (≤)	Tool			
	DEAP	TensorGP (CPU)	TensorGP (GPU)	Operon
10	569×	1210×	1408×	0.375×
100	741×	1197×	1384×	0.357×
1,000	2372×	1039×	1123×	0.290×
10,000	4611×	312.0×	318.5×	0.432×
100,000	4902×	61.5×	43.0×	0.423×

Table 1: Average NEPS speedups for various fitness case thresholds. For a given threshold value, the average is calculated from all results regarding thresholds less than or equal to this value. The last row represents an overall average.

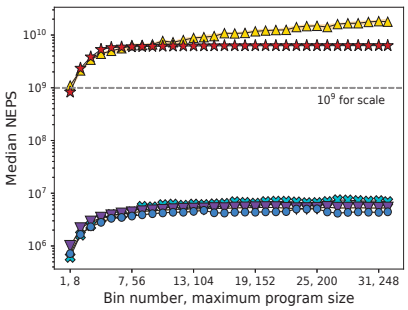
Beyond Operon, our accelerator was able to consistently outperform a modern GPU system running TensorGP, where our results for TensorGP align with results previously listed [1]. Interestingly, DEAP sometimes performed better than TensorGP for the smallest number of fitness cases (e.g., Fig. 3e), although TensorGP scaled much better with larger numbers of fitness cases. All in all, we



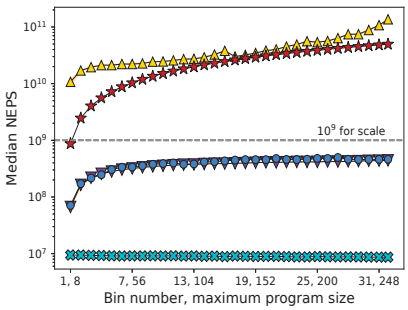
(a) Primitive set nicolau_a, 10K fitness cases. **Max value: 199 billion NEPS**.



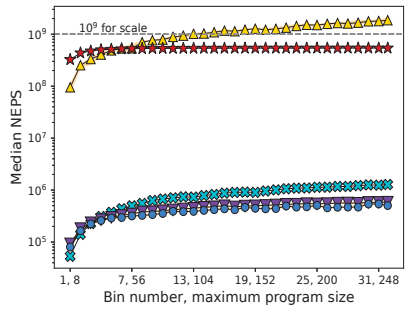
(b) Primitive set nicolau_a, 100K fitness cases. Max value: 197 billion NEPS.



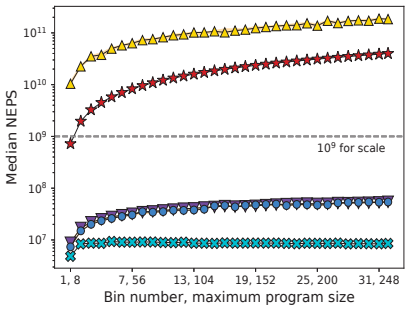
(c) Primitive set nicolau_b, 100 fitness cases. Max value: 18.3 billion NEPS.



(d) Primitive set nicolau_b, 10K fitness cases. Max value: 136 billion NEPS.



(e) Primitive set nicolau_c, 10 fitness cases. Max value: 1.86 billion NEPS.



(f) Primitive set nicolau_c, 1K fitness cases. Max value: 188 billion NEPS.

Fig. 3: Sample median node evaluations per second (NEPS) vs. program bin number and maximum program size, for six different combinations of primitive set and number of fitness cases. For Operon and TensorGP plots, the $75^{\text{th}}/25^{\text{th}}$ percentiles for runtime are plotted above/below each sample point, which are noticeable in only a few instances; for example, see the first bin of (a). Note that the legend from (a) applies to all sub-figures. Also, note the use of a log scale.

note from the given plots that a single, pipelined program tree by way of our accelerator could keep up with and sometimes outpace a two-socket, 28-core, 56-thread CPU system running the state-of-the-art Operon tool, and we consistently outperformed a modern GPU system running the recent, high-performance TensorGP tool. On average, across all fifteen experiments and all random programs, the FPGA was 4,902× faster than DEAP, 61.5× faster than TensorGP executing with the CPU, 43× faster than TensorGP executing with the GPU, and 2.4× slower than Operon. For some more specific trends in regard to number of fitness cases, Table 1 presents an average NEPS speedup for the FPGA in the context of all experiments with 1) 10 fitness cases, 2) 100 fitness cases or less, 3) 1,000 fitness cases or less, 4) 10,000 fitness cases or less, and 5) 100,000 fitness cases or less, where the values provided by 5) are used to represent an overall average for the conducted experiments, already listed above. To calculate one of these averages for a particular tool and fitness case threshold, we first divided a sum of node evaluations by a sum of median runtimes, with the values in each sum stemming from all experiments regarding the particular tool and fitness case threshold. Then, to compute the relevant speedup, we divided a corresponding average for the FPGA by the average computed for the relevant tool. Note that using the median runtime from each bin in these calculations was appropriate, given that we wanted to establish a typical runtime value for each bin of each experiment. For more details, please refer to our code.

6 Current Limitations and Potential Optimizations

Below, we list three limitations of our initial accelerator architecture, and then we present five potential optimization strategies that could alleviate the three limitations and allow for an updated accelerator to achieve a speedup over our current design by 32–144× as well as support for larger program depths/sizes.

6.1 Current Limitations

Comprehensive Support For a function tree (Section 3.3) to be able to support arbitrary programs, *every* function unit must support *all* function primitives defined by the primitive set. Therefore, depending on the number of function primitives and the types of low-level device resources utilized for these primitives, the maximum depth/size of function trees—and, thus, programs—can be restricted. For our experiments that utilized primitives relying on floating-point operations, we were ultimately constrained by the number of floating-point DSP and embedded memory resources available within the target FPGA device.

Exponential Growth For an m-ary function tree (where m is the maximum function arity of the primitive set) with $m > 1$, the amount of area needed to implement the tree grows exponentially with increasing tree depth. Namely, for $m > 1$ and a function tree depth of d, $\frac{1-m^{d+1}}{1-m}$ generic function units are needed for the tree, which can prevent up to $\frac{100}{m}\%$ of some device resource(s) from being used when maximizing d. (For $m = 1$, only $d + 1$ function units are needed.)

Low Resource Utilization If each function unit in the tree is capable of computing *every* function primitive, then for $|F|$ function primitives, the utilization of each function unit in terms of these high-level primitives is $\frac{1}{|F|}$. The utilization of low-level device primitives (e.g., floating-point DSPs) can be significantly lower, depending on the function primitive implementations.

6.2 Potential Optimizations

We note that the following five optimizations are independent from one another[3], and, thus, if all could be achieved simultaneously, a speedup between $2{\cdot}2{\cdot}2{\cdot}2{\cdot}2 = 32\times$ and $2 \cdot 6 \cdot 3 \cdot 2 \cdot 2 = 144\times$ could be achieved over our current accelerator.

Use Compacted Trees To be able to more effectively leverage device resources as well as support larger program depths/sizes, we plan to explore various "compacted tree" architectures. Ideally, such an architecture would allow for the use of all resources that are currently unused due to exponential growth in area—either through the use of a single, more efficient compute engine or through multiple compute engines—and such an architecture would also offer native support for larger depths/sizes. One option may be to construct a unified parallel/sequential tree structure, similar to what has been developed for tree-based accumulators [26]. Another possibility may be to design a linear architecture that natively handles flattened tree (e.g., prefix/postfix) representations. If either option could result in a fully-pipelined architecture, the latter may be able to more effectively map flattened programs onto function unit resources, but such an architecture would seemingly require state memory (e.g., a temporary stack) to be included in the pipeline in order to maximize throughput, which would likely infer significantly more memory resources than a spatially parallel tree representation.

For the current study, if we could leverage all resources not currently used, we could improve upon our performance results by upwards of $2\times$ (Section 6.1).

Multiplex Function Unit Resources Function unit primitives experience poor utilization due to the fact that they are implemented with independent IP blocks. This issue could be improved upon by implementing a function unit via a single IP block that multiplexes a minimal amount of some devices resource(s), e.g., floating-point DSPs. Such an "overlay" could free up a significant amount of resources, allowing for further parallelization of program evaluation. For example, in the context of the most complex primitive set used for this paper, nicolau_c, the most expensive primitive was tanh, which utilized $\frac{13}{42} \approx 31\%$ of all DSPs allocated for each function unit. Thus, with an appropriate overlay, $\frac{29}{42} \approx 69\%$ of all DSPs for function units could be recovered. Carrying out a similar process for all primitive sets used in this paper, about 50% of all DSPs

[3] A possible exception could occur when dealing with timing optimization, since the resulting clock frequency may unexpectedly get better or worse with design changes.

utilized for function units could be recovered on average, which could translate into an average speedup by up to 2×. However, in general, for a primitive set containing $|F|$ functions, an optimal overlay could allow for up to an $|F|\times$ speedup if all functions were to utilize the same amount of low-level resources. Therefore, in our case, where we currently utilize an average of approximately six functions, we estimate that we could achieve a speedup of 2–6× by using optimized (or alternate) functions.

Design for Higher Clock Frequencies For our accelerator, throughput (i.e., performance) is directly proportional to clock frequency. With modern FPGAs, it is not uncommon for designs to achieve clock frequencies in the range 400-850 MHz after optimizing for timing [18, 23, 24]. Our current accelerator has not been fully optimized, and, as such, we achieved an average clock frequency of 178 MHz across the fifteen hardware compilations performed for this paper. We estimate that we can achieve up to a 2–3× higher average clock frequency once we further optimize for timing (and potentially move to a newer device), which would allow for an average speedup over our current design by up to 2–3×.

Use a Higher-End FPGA Device With a more modern, higher-end FPGA implemented on newer process-node technology (e.g., [11]), we should be able to support at least 2× more floating-point DSP resources and 1.5× more embedded memory resources, in addition to higher clock frequencies. With 2× more DSP resources, we expect that we can further parallelize our current floating-point computations by up to 2×, which should allow for up to a 2× speedup.

Double-buffer GP Runs When our accelerator enters the context of a full GP system, including evolution, we expect that we can execute two GP runs simultaneously, by evolving one population whilst evaluating another. Such an optimization would generally not make sense for a typical GP system (with exception to possibly a combined CPU/GPU system), since any additional compute cores would likely be used to further parallelize program evaluation. If the total time taken for evolution and device communication can be less than the total time taken for evaluation, then this optimization should allow our accelerator to achieve an additional speedup by up to 2×.

7 Conclusion

In this paper, we leveraged a modern FPGA device to implement a hardware accelerator that more closely aligns with the computing model of tree-based GP when compared to CPU/GPU solutions. Specifically, the presented architecture dynamically compiles program trees onto a reconfigurable function tree pipeline that can generate outputs for entire program expressions every clock cycle and transition between separate programs within a single cycle.

We showed that our accelerator on a 14nm FPGA achieves an average speedup of 43× when compared to a recent open-source GPU solution implemented on 8nm process-node technology, and an average speedup of 4,902× when compared to a popular baseline GP software tool running parallelized across all cores of a 2-socket, 28-core (56-thread), 14nm CPU server. Despite our single-FPGA accelerator being 2.4× slower on average when compared to a recent state-of-the-art GP software tool executing on the same 2-processor CPU system, we described future extensions that could provide a 32–144× speedup over our current design.

References

1. Baeta, F., Correia, J., Martins, T., Machado, P.: Exploring genetic programming in TensorFlow with TensorGP. SN Computer Science **3**(2), 154 (2022). https://doi.org/10.1007/s42979-021-01006-8
2. Banzhaf, W., Harding, S., Langdon, W.B., Wilson, G.: Accelerating Genetic Programming through Graphics Processing Units., pp. 1–19. Springer US, Boston, MA (2009). https://doi.org/10.1007/978-0-387-87623-8_15
3. Burlacu, B., Kronberger, G., Kommenda, M.: Operon C++: An efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion. p. 1562–1570. GECCO '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3377929.3398099
4. Chitty, D.M.: Fast parallel genetic programming: multi-core CPU versus many-core GPU. Soft Computing **16**(10), 1795–1814 (2012). https://doi.org/10.1007/s00500-012-0862-0
5. Chitty, D.M.: Faster GPU-based genetic programming using a two-dimensional stack. Soft Computing **21**(14), 3859–3878 (2017). https://doi.org/10.1007/s00500-016-2034-0
6. Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: DEAP: Evolutionary algorithms made easy. J. Mach. Learn. Res. **13**(1), 2171–2175 (2012)
7. Funie, A.I., Grigoras, P., Burovskiy, P., Luk, W., Salmon, M.: Run-time reconfigurable acceleration for genetic programming fitness evaluation in trading strategies. J. Signal Process. Syst. **90**(1), 39–52 (2018). https://doi.org/10.1007/s11265-017-1244-8
8. Goribar-Jimenez, C., Maldonado, Y., Trujillo, L., Castelli, M., Gonçalves, I., Vanneschi, L.: Towards the development of a complete GP system on an FPGA using geometric semantic operators. In: 2017 IEEE Congress on Evolutionary Computation (CEC). pp. 1932–1939 (2017). https://doi.org/10.1109/CEC.2017.7969537
9. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edn. (2017)
10. Hooker, S.: The hardware lottery. Commun. ACM **64**(12), 58–65 (2021). https://doi.org/10.1145/3467017
11. Intel: Intel Agilex™ M-Series FPGA and SoC FPGA Product Table, 2015 [Online], https://cdrdv2.intel.com/v1/dl/getContent/721636
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)

13. La Cava, W., et al.: Contemporary symbolic regression methods and their relative performance. In: Vanschoren, J., Yeung, S. (eds.) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks. vol. 1 (2021)
14. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: European Conference on Genetic Programming. pp. 73–85. Springer (2008). https://doi.org/10.1007/978-3-540-78671-9_7
15. Martin, P.: A hardware implementation of a genetic programming system using FPGAs and Handel-C. Genetic Programming and Evolvable Machines **2**(4), 317–343 (2001). https://doi.org/10.1023/A:1012942304464
16. Miller, J.F.: Cartesian genetic programming: its status and future. Genetic Programming and Evolvable Machines **21**(1), 129–168 (2020). https://doi.org/10.1007/s10710-019-09360-6
17. Nicolau, M., Agapitos, A.: Choosing function sets with better generalisation performance for symbolic regression models. Genetic Programming and Evolvable Machines **22**(1), 73–100 (2021). https://doi.org/10.1007/s10710-020-09391-4
18. Nurvitadhi, E., et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 5–14. FPGA '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3020078.3021740
19. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
20. Putnam, A., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro **35**(3), 10–22 (2015). https://doi.org/10.1109/MM.2015.42
21. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines **10**(4), 447–471 (2009). https://doi.org/10.1007/s10710-009-9092-3
22. Sidhu, R.P.S., Mei, A., Prasanna, V.K.: Genetic programming using self-reconfigurable FPGAs. In: Lysaght, P., Irvine, J., Hartenstein, R. (eds.) Field Programmable Logic and Applications. pp. 301–312. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/978-3-540-48302-1_31
23. Stitt, G., Gupta, A., Emas, M.N., Wilson, D., Baylis, A.: Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. p. 173–182. FPGA '18, Association for Computing Machinery (2018). https://doi.org/10.1145/3174243.3174262
24. Tan, T., Nurvitadhi, E., Shih, D., Chiou, D.: Evaluating the highly-pipelined Intel Stratix 10 FPGA architecture using open-source benchmarks. In: 2018 International Conference on Field-Programmable Technology (FPT). pp. 206–213 (2018). https://doi.org/10.1109/FPT.2018.00038
25. Veeramachaneni, K., Arnaldo, I., Derby, O., O'Reilly, U.M.: FlexGP - Cloud-based ensemble learning with genetic programming for large regression problems. J Grid Computing **13**(3), 391–407 (2015). https://doi.org/10.1007/s10723-014-9320-9
26. Wilson, D., Stitt, G.: The unified accumulator architecture: A configurable, portable, and extensible floating-point accumulator. ACM Trans. Reconfigurable Technol. Syst. **9**(3) (2016). https://doi.org/10.1145/2809432
27. Wright, L.G., et al.: Deep physical neural networks trained with backpropagation. Nature **601**(7894), 549–555 (2022). https://doi.org/10.1038/s41586-021-04223-6
28. Yao, X.: Following the path of evolvable hardware. Commun. ACM **42**(4), 46–49 (1999). https://doi.org/10.1145/299157.299169