Genetic Programming and Evolvable Machines (2025) 26:8
https://doi.org/10.1007/s10710-024-09505-2

Using FPGA devices to accelerate the evaluation phase of tree-based genetic programming: an extended analysis

Christopher Crary[1] · Wesley Piard[1] · Greg Stitt[1] · Benjamin Hicks[1] · Caleb Bean[1] · Bogdan Burlacu[2] · Wolfgang Banzhaf[3]

Received: 1 November 2023 / Revised: 17 December 2024 / Accepted: 18 December 2024 /
Published online: 7 January 2025

Abstract

This paper establishes the potential of accelerating the evaluation phase of tree-based genetic programming through contemporary field-programmable gate array (FPGA) technology. This exploration stems from the fact that FPGAs can sometimes leverage increased levels of both data and function parallelism, as well as superior power/energy efficiency, when compared to general-purpose CPU/GPU systems. In this investigation, we introduce a fixed-depth, tree-based architecture that can fully parallelize tree evaluation for type-consistent primitives that are unrolled and pipelined. We show that our accelerator on a 14nm FPGA achieves an average speedup of 43× when compared to a recent open-source GPU solution, TensorGP, implemented on 8nm process-node technology, and an average speedup of 4,902× when compared to a popular baseline GP software tool, DEAP, 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 the recent state-of-the-art Operon tool executing on the same 2-processor, 28-core CPU system, we show that this single-FPGA system is 1.4× better than Operon in terms of performance-per-watt. Importantly, we also describe six future extensions that could provide at least a 64–192× speedup over our current design. Therefore, our initial results provide considerable motivation for the continued exploration of FPGA-based GP systems. Overall, any success in significantly improving runtime and energy efficiency could potentially enable novel research efforts through faster and/or less costly GP runs, similar to how GPUs unlocked the power of deep learning during the past fifteen years.

Keywords Tree-based genetic programming · Field-programmable gate array · Domain-specific architecture · Hardware acceleration

Extended author information available on the last page of the article

1 Introduction

Throughout history, the evolution of AI has been heavily constrained and influenced by the computing technologies that are available [44, 47]. With such constraints, there is continual motivation to develop clever applications of existing technologies. Notably, resulting solutions can sometimes affect the trajectories of science and life in profound ways. For example, the application and development of GPU technology for neural networks significantly expanded the utility of this form of AI [44, 47], which then drove substantial research initiatives [92], transformed Nvidia into the most valuable public company [79], and enabled the creation of applications and systems that have greatly influenced the world at large [81, 98, 116].

Of course, some other challenge or trade-off is eventually reached. In the case of neural networks, recent work has highlighted serious scaling challenges for energy consumption and various other costs [2, 15, 72, 103, 116], which has kick-started or revived the exploration of many other learning systems. Importantly, one domain that is promising in this regard is genetic programming (GP) [13, 59, 89], for which it has been widely shown that the pairing of evolutionary search with alternative model structures (e.g., trees, assembly languages, tangled program graphs, etc.) can sometimes allow for more compact solutions and enhanced efficiency during inference [57, 64, 89]. However, training often remains complex with current GP techniques, which motivates improvements to the computational efficiency of such procedures [19, 27, 89].

In general, there are various ways to improve the computational efficiency of GP training, and they normally involve either increasing performance (i.e., throughput) or enhancing energy efficiency, for which there are at least four key benefits: (1) with increased performance, useful solutions can potentially be found in a shorter amount of time; (2) with improved energy efficiency, there is the potential for lower operational costs, which (3) can allow for more cost-effective multi-computer GP systems, in turn allowing for higher performance; and (4) with either improved performance or improved energy efficiency, better solutions can potentially be found when allowing the system to consume a similar amount of runtime/energy. Notably, in regard to this last point, consuming a similar amount of runtime/energy does not necessarily mean conducting a single evolutionary run for more generations; it could also mean more hyperparameter tuning, larger statistical studies, etc., which are frequently useful.

Notably, in many application domains, the demands of ever-increasing performance and energy efficiency has now led to the broad development of *domain-specific architectures* [44]. However, in the context of genetic programming, there exist only a few instances of domain-specific architectures (Sect. 3), which is particularly surprising given that GP is often an "embarrassingly parallel" procedure [89]. Although general-purpose CPU/GPU systems can be configured to implement the multiple-program, multiple-data model of GP, it is challenging to fully exploit the inherent parallelism (Sect. 3). For CPUs, the use of multiple cores/threads is relatively straightforward, but it is often difficult or prohibitively

expensive (in terms of power and other costs) to continually scale up [44]. And for GPUs, which offer numerous simpler cores and have been highly successful in accelerating other forms of machine learning, the need for conditional program execution (e.g., to decide which function primitive to execute) and large cache sizes generally limits acceleration capabilities [23, 27, 94]. To address such limitations of CPU/GPU systems, this paper introduces a specialized accelerator for the evaluation phase of tree-based GP, implemented using contemporary field-programmable gate array (FPGA) technology as described in Sect. 4. We provide a more detailed description of FPGA devices in Sect. 2.3, but in brief, FPGAs are programmable computing systems that enable the synthesis of specialized digital circuitry from various levels of abstraction, without recourse to integrated circuit development.

We compare the performance of our architecture with the evaluation engines given by three actively maintained, open-source tree-based GP software tools: *DEAP* [29], *TensorGP* [12], and *Operon* [19]. 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., data points), and we estimate evaluation performance in terms of *nodes per second* (*NPS*), which is similar to the conventional *GP operations per second (GPops/s)*, as described in Sect. 5.1. For each CPU-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) for TensorGP. For our accelerator, we use a 14nm Intel Stratix 10 SX 1SX280HN2F43E2VG FPGA provided by an Intel Programmable Acceleration Card D5005. We use the VHDL programming language to specify our designs [115], and we compile our accelerator by way of Quartus Pro 19.2.0, Build 57.

When compared to DEAP [29], a popular baseline for GP software tools, our accelerator achieves an average speedup of 4,902×. Compared to TensorGP [12], a recent general-purpose GP software tool targeting both CPU and GPU systems, our architecture achieves an average speedup of 62× in regard to CPU execution and 43× in regard to GPU execution. Finally, when compared to Operon [19], a recent state-of-the-art GP tool tailored to symbolic regression [64], our single-FPGA accelerator executes 0.42× faster (i.e., 2.4× slower) on average when compared to the same *2-processor* CPU system, although there are multiple instances in which our accelerator performs fastest. Importantly, when we additionally consider energy consumption, we showcase that the FPGA provides a 1.4× average improvement over Operon in terms of performance-per-watt. However, even more importantly, we also identify six future extensions that could allow for at least a 64–192× speedup over our current system. Altogether, our initial results provide considerable motivation for the continued exploration of FPGA-based GP systems. Especially when accounting for our proposed extensions, FPGAs may allow for faster and/or less costly GP runs, in which case it may also be possible for better solutions to be found when allowing an FPGA to consume the same amount of runtime/energy as another computing platform.

The remainder of the paper is organized as follows. Section 2 provides a high-level overview of modern computing, which should ultimately motivate

domain-specific architectures for GP systems. Section 3 describes related work, with emphasis placed primarily on recent achievements. Section 4 details our hardware accelerator architecture. Section 5 describes our design of experiments. Section 6 presents results. Section 7 discusses challenges regarding our current architecture, followed by planned future extensions that should address the challenges and allow for state-of-the-art performance, followed by some final considerations. Section 8 concludes our study.

2 Background

In this section, we provide some relevant background on modern (digital) computing. Importantly, we note that the following discussion provides appropriate context for the comparison of GP systems laid out in Sect. 3, and it also motivates domain-specific architectures, like the one we introduce in Sect. 4.

2.1 A high-level view of modern computing

For about twenty years now, the practical relevance of *Moore's Law* has been waning, and *Dennard scaling* no longer applies [33, 44]. In brief, Moore's Law describes an empirical regularity that the maximum number of transistors in an integrated circuit chip doubles roughly every two years,[1] and Dennard scaling refers to the idea that as transistor circuit area scales down, power density roughly stays the same.

Notably, from the mid-1980s to the early-2000s, a combination of Moore's Law and Dennard scaling allowed average CPU performance to roughly double (i.e., execution times to roughly halve) every two years [44]. Then, from the end of Dennard scaling in the early-2000s until the late-2010s, the use of multiple *general-purpose* cores per chip kept Moore's Law alive, but various theoretical and practical barriers led to a significant slowing of performance enhancements. Primarily, performance enhancements were constrained by consistent power budgets–which, in general, are constrained by electromigration, mechanical, and thermal limits–as well as the limits on parallelism as prescribed by *Amdahl's Law* [8]. With consistent power budgets due to physical constraints, the number of general-purpose cores in a single chip approached a practical upper limit [33, 44]. Thus entered the next big trend, which is still relevant today: *domain-specific architectures (DSAs)*. Importantly, with DSAs, hardware specialized to a particular application domain can often accomplish more with a similar, sometimes smaller, power budget [44, 84, 90, 114, 117].

In general, domain-specific architectures can offer equivalent, and sometimes better, performance and energy benefits when compared to modern general-purpose architectures, such as central processing units (CPUs) and graphics processing units (GPUs) [44, 84, 90, 114, 117]. Although DSAs can sometimes serve as complete solutions, the latest trend is to integrate both general-purpose and domain-specific

[1] Recent work allows us to conclude that Moore's Law is still alive [95].

chiplets into a single circuit, so that the system can efficiently support a wide range of applications while additionally being optimized for a particular subset [44, 99, 127]. Such a composite system is commonly referred to as a *system-on-chip (SoC)*. One recent, notable example of an SoC is the Apple M1 Ultra chip, which consists of 114 billion transistors primarily allocated to a 20-core CPU, 64-core GPU, 32-core "Neural Engine," and memory [10]. In terms of number of transistors, this chip represents roughly a *50 million times increase* from the Intel 4004 chip released in 1971—the first commercially produced microprocessor—which consisted of 2,300 transistors [53].

2.2 Domain-specific architectures

Many application domains can benefit from the use of specialized computer architectures [44, 61]. In general, a DSA can be leveraged when either (1) large amounts of algorithmic parallelism can be exploited or (2) some low-power, low-area, or specialized implementation is desired. However, practically speaking, additional factors must often be considered, such as those involving nonrecurring engineering (NRE) time, NRE cost, unit cost, sale volume, and sale price [44, 112, 119].

Naturally, there are different mechanisms for designing a DSA, and each comes with its own set of trade-offs. Generally speaking, an *application-specific integrated circuit (ASIC)* provides the most flexibility in achieving some desired performance and power characteristics, as well as low unit costs, but this route usually requires years of NRE time and millions of dollars in NRE costs [44]. Unfortunately, according to *Rock's Law* [31, 96], NRE costs often increase significantly with newer device technologies, which continually makes ASIC engineering more of a technical and economic challenge.[2] Thus, ASICs are usually most applicable for applications in which large NRE times/costs are tolerable and high sale volumes/prices are expected, so that NRE cost may be offset by a large amount of low-cost, high-profit sales.

Besides the fabrication of an ASIC, another alternative for designing a DSA is to utilize a *reconfigurable computing (RC)* system [25, 36, 44, 118]. In essence, RC systems are programmable computing systems in which specialized digital circuitry can be synthesized from different levels of abstraction, without recourse to integrated circuit development.[3] When compared to an ASIC, RC systems often gain appeal by trading off higher unit costs for lower NRE costs while also retaining significant performance/power benefits [44, 62]. In general, RC systems are useful for prototyping designs before ASIC development or for developing standalone solutions in which either (1) NRE cost must be low or (2) high unit cost can be amortized by high sale volume/price or the ability to reconfigure the device over time. Additionally, with the ability to design high-performance, low-power solutions, and the ability to support different hardware designs over multiple reconfigurations, RC

[2] However, the manufacturing of older technologies generally becomes cheaper over time [44].

[3] An RC system is itself an ASIC, yet an ASIC designed to expose some aspect(s) of reconfigurability.

Inputs			Outputs	
A	B	C_{in}	Sum	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

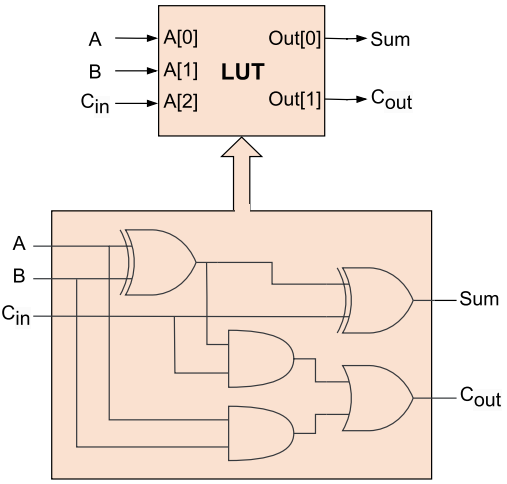


Fig. 1 An example LUT-based implementation. A LUT-based implementation of a full adder, with carry-in and carry-out. Importantly, the depicted LUT memory could implement not only this circuit, but *all* 3-input, 2-output digital circuits. Also, note that the schematic in the bottom-right is just for illustration; with a LUT-based implementation, no logic gates are used. For a more thorough example, see [43, Example 5.5]

platforms are used for various research [84, 90, 97, 114, 117, 123]. The most popular type of RC system is a *field-programmable gate array* (*FPGA*), which we detail in the next subsection.

2.3 Field-programmable gate arrays

Unfortunately, the name *field-programmable gate array* fails to capture the main mechanism by which FPGAs implement designs—there, in fact, does not exist any array of (logic) gates [43]. Primarily, there exist many small memories, known as *lookup tables (LUTs)*, which can implement the truth table(s) corresponding to some desired circuitry. For a simple example of such an implementation, see Fig 1. By way of LUTs, FPGAs support combinational logic. To additionally support sequential logic, FPGAs also increasingly leverage *flip-flop* memory components [43]. Ultimately, combinations of LUT and flip-flop components can allow for highly flexible circuit configurations, but implementing an entire system only with such components may not be feasible, depending on design complexity. To address this fact, modern FPGA systems also contain more coarse-grained components such as integer and floating-point multiply-adders,[4] high-bandwidth memories, general-purpose CPU cores, and sometimes other specialized computing cores (e.g., "AI

[4] These multiply-adders are often referred to as *digital signal processing* (*DSP*) *blocks/engines*, or just *DSPs* for simplicity. For floating-point, newer devices can efficiently implement single/half precision and "bfloat16" [6, 24, 37, 45, 50], although double precision is still relatively complex.

engines" [4]), so that common computing tasks can more readily be implemented [4, 45, 52, 84, 112, 117]. In this respect, modern FPGAs are, themselves, SoC devices. Overall, by integrating many thousands (or millions) of components via a reconfigurable interconnect, FPGA devices can implement massively parallel designs, many of which often require much less power to operate when compared to CPU/GPU systems [44, 97].

Although FPGA devices can provide significant benefits for many applications, various challenges still exist, which we can classify into three major categories:

1. *Productivity.* When compared to standard software development, it is generally regarded that FPGAs have low design productivity. Overall, there are numerous reasons for this, but the primary issue is that designing meaningful circuits generally demands considerable digital-design expertise, as well as knowledge of low-level device details [112]. To reduce required effort, many developments have taken place throughout the past few decades, including the creation of numerous specialized programming languages [55], the extension of pre-existing high-level programming languages (e.g., C++, Python, etc.) [66, 91], and the simplification of design tools [106]. Overall, significant improvements have been made, especially in regard to *high-level synthesis (HLS)* [65, 82, 93], but classical techniques are still often necessary in order to implement a circuit with the desired performance/energy characteristics.[5] In addition to the above, another important limitation regarding productivity is that the process of synthesizing a circuit into hardware, otherwise known as *compilation*, can take hours or even days [112]. Although functionality can be verified with software simulations, a common challenge is that design bugs often only appear once the relevant specification is interpreted into hardware [75]. Linting tools exist to help reduce the prevalence of this issue, but they do not catch everything. Therefore, with such lengthy compilation times, debugging can be an extremely inefficient and frustrating process [75]. Ultimately, the main cause for long compilation times is that the placement and routing (PR) of a circuit is an NP-hard problem [39, 128]. Unfortunately, PR often becomes increasingly difficult as FPGA architectures get larger and more complex. To alleviate the challenges of PR, numerous techniques have been considered, with three primary solutions being (1) heuristic or AI methods [16, 39, 128], (2) partial reconfiguration [78, 121], and (3) overlay technologies [26, 108, 113, 125]. In brief, *partial reconfiguration* is an ability provided by some FPGA architectures through which a subset of the FPGA can be programmed instead of the whole chip, so that the workload for PR is reduced. Separately, an *overlay* is a specialized circuit implemented on the FPGA device that exposes simpler, higher-level aspects of reconfigurability. Essentially, with an appropriate overlay, compilation can be near-instantaneous for the class of designs supported by the overlay [26, 108, 113, 125]. In addition, unlike standard FPGA compilations, overlays can allow for high-level designs to be specified by portable machine code. Thus, when moving to another FPGA technology, the overlay itself may

[5] For some useful introductions to digital design, see [43, 115].

need to be recompiled, but machine codes relevant to the overlay can be reused. Generally speaking, design iterations may necessitate the compilation of a new overlay, but several techniques have been designed to mitigate this challenge, e.g., the identification of a "supernet" in order to support the superset of some netlists [26], incremental compilation [125], and the use of device libraries in order to retain various compilation information. Similar to overlays, another strategy to combat large compilation times has been to embed within a FPGA system higher-level computing cores (e.g., "AI engines" [4]), in order to enable a more direct mapping of high-level software. Although such cores can sometimes allow for significant performance and energy benefits when compared to CPU/GPU devices [4], the capabilities of such cores may be too limited for certain applications.

2. *Amenability.* In general, not all algorithms are directly amenable to FPGA devices [112]. One key reason for this is that various high-level constructs do not have standard mappings to FPGAs, often because such mappings would not be widely useful and/or intuitive [112]. For instance, pointers and recursions do not immediately make sense for most LUT-based circuits (Sect. 2.3), and even though support may be possible through additional hardware complexity, alternative constructs are likely more suitable if performance/energy characteristics are important. Separately, even if an algorithm can be directly mapped, it is not guaranteed that an inferred circuit will effectively utilize low-level device resources, and poor mappings often lead to lower computational efficiency. Fortunately, such issues have become less pronounced over time with heterogeneous computing platforms and newer FPGA technologies that support additional and more versatile components [4, 52], but the mapping of certain functionality can still be challenging, and algorithmic tweaks are often necessary. In addition to these challenges, another amenability issue is that the complex interconnect within FPGA devices usually necessitates a clock frequency that is significantly lower than modern CPU/GPU devices [112]. For instance, even with high-end technologies, clock frequencies for programmable logic must almost always be set to less than 1 GHz, and generally much lower than that, often being at least a factor of five less than similarly recent CPU/GPU technologies [84, 90, 97, 112, 114, 117]. Thus, when the goal of using an FPGA is to achieve higher performance than CPU/GPU technologies, a massive amount of parallelism is often needed, which is frequently achieved through pipelining and/or pipeline duplication. However, when the goal of using an FPGA is to leverage enhanced energy efficiency, the lower clock frequencies can often be a plus.
3. *Cost.* Due in part to the aforementioned challenges regarding amenability, there has yet to be a "killer app" for FPGA devices [112].[6] Whereas the development of GPUs was clearly motivated by graphics applications and, now, general-purpose scientific computing [87], FPGAs evolved from a considerably smaller market targeting the development of "glue logic" and the prototyping of ASIC

[6] And if there is a killer app, an ASIC can likely be designed to extract additional benefits [54]. Regardless, there are various notable applications for FPGAs, e.g., SmartNIC designs for data centers [90].

devices [25, 36, 44, 118]. In general, the larger demand for GPUs has continually led to a significant discrepancy in subsidies for research/development, supply, and device costs. Until recently, it was not uncommon for the latest GPUs to cost a few hundred dollars while the latest FPGAs were at least $10,000 [112]. Nowadays, the price gap has narrowed or inverted, with some high-end GPU devices costing around $30,000 [71], but FPGA devices are still typically pricey or unavailable due to low supply. However, some free or cheap cloud-based platforms with FPGAs now exist; for example, see [5, 7].

Despite these notable challenges, the performance and energy benefits of reconfigurable hardware are frequently worth the effort [44, 84, 90, 97, 112, 114, 117]. Importantly, following from this section, it seems that genetic programming (GP) could be a prime candidate for FPGA devices, since the algorithms of GP are often embarrassingly parallel and often exhibit both data and function parallelism. In the remainder of this paper, we explore this possibility further.

3 Related work

One general tendency for improving GP has been to increase the computational efficiency of training procedures [12, 14, 19, 22, 23, 89, 94]. To improve training efficiency, focus is often placed on the program evaluation phase, since this phase generally involves evaluating hundreds or thousands of computer programs on hundreds or thousands of fitness cases (i.e., data points), for each of hundreds or thousands of generations [12, 14, 19, 22, 23, 89, 94]. Notably, the evolutionary phases of GP also typically operate on the same number of programs across the same number of generations, but these procedures are often not affected by the number of fitness cases,[7] which generally leads to a significant workload imbalance between evaluation and evolution. Even with efficient forms of evaluation, it has been shown that such procedures can often account for over 90% of the total runtime taken for a GP run [19].

In the remainder of this section, we review a few recent GP systems that have pushed the boundaries of computational efficiency. We continue from the review provided by Chitty in [23], including his novel GPU contributions presented in the same work, as well as a few systems developed before and after this study. Primarily, we focus on tree-based GP systems (Sect. 3.1), since these are the most common [86, 89, 122], although we also mention a few notable systems involving other program representations (Sect. 3.2). In addition, we only focus on notions of performance rather than energy, since most studies have not considered the latter.

Lastly, we note that this current paper provides an extension to our work in [27], where each of Sects. 2, 3, 4, 6, 7, and 8 include a significant amount of new content. More specifically, Sects. 2 and 3 are new and provide extensive background regarding how and why the acceleration of tree-based GP has been a challenging pursuit, as well as general motivation for the exploration of FPGA-based GP systems. Then,

[7] Of course, there exist exceptions, e.g., lexicase selection [63].

in Sect. 4, additional information about our preliminary evaluation architecture is detailed. Following this, in Sects. 5 and 6, additional experiments and results are documented. Lastly, in Sects. 7 and 8, expanded discussions of the relevant challenges, future work, and conclusions are presented.

3.1 Tree-based GP systems

We organize the systems listed in this section based on the relevant implementation technology, where we consider GPUs, CPUs, and FPGAs. For a short summary of the relevant GP systems, see Table 1.

3.1.1 GPU solutions

In [23], Chitty extends to a GPU system the ideas he presented in [22] involving a multi-core CPU system. Most importantly, Chitty reaffirms the idea that appropriately leveraging cache memory within CPU/GPU systems can greatly improve the evaluation performance of many GP systems. In particular, Chitty notes that when evaluating programs on multiple fitness cases—a common occurrence—typical cache memory layouts can often be effectively leveraged by performing node operations on batches of fitness cases. In this manner, memory locality is exploited more than with the traditional strategy that evaluates a program entirely before moving on to a following fitness case, since the proposed strategy can immediately reuse instructions and immediately use contiguous data. To effectively exploit cache memory, Chitty uses a two-dimensional stack representation as well as specialized GP operators, and extra care is taken to ensure that this format maps well to the chosen NVIDIA Kepler GK104 GPU architecture [23]. Chitty distinguishes

Table 1 Chronological listing of the tree-based GP systems discussed in Sect. 3.1

Year	Author(s)	Hardware	Performance (in billions)	Reference
1998	Koza et al.	FPGA	N/A	[60]
1999	Sidhu et al.	FPGA	N/A	[104]
2012	Augusto et al.	CPU/GPU	11.85–13.02 (GPops/s)	[11]
2017	Chitty	GPU	56–1,411 (GPops/s)	[23]
2017	Staats et al.	CPU/GPU	N/A	[111]
2018	Funie et al.	CPU+FPGA	N/A	[35]
2019	Langdon	2 CPUs	139 (GPops/s)	[67]
2020	Burlacu et al.	CPU	94 (NPS)	[19]
2021	Sathia et al.	CPU/GPU	N/A	[101]
2022	Langdon et al.	2 CPUs	1,103 (*effective* GPops/s)	[70]
2022	Baeta et al.	GPU	0.264 (GPops/s)	[12]
2022	Zhang et al.	CPU/GPU	N/A	[131]
2023	Crary et al.	FPGA	Up to 198.5 (NPS)	Current paper

For a distinction between GPops/s, effective GPops/s, and NPS, refer to the text. Also, note that performance numbers are given in billions

that even though GPU devices exhibit more spatial parallelism than CPU devices, there is often less high-performance cache memory per computing core within GPU devices, which makes it challenging to exploit such additional cores for GP. In order to reduce pressure on the limited GPU caches and better support a two-dimensional stack, Chitty also explores a linear representation through which a smaller stack depth and fewer stack operations can be used, where a compilation process is included to convert traditional postfix expressions into the prescribed linear notation. Lastly, to further reduce pressure on cache memory, local device registers are employed to implement a portion of stack memory.

Overall, when compared to the GPU-based implementation of GP from Robilliard et al. [94], which used a one-dimensional stack for evaluation, Chitty's optimized GPU implementation achieved a 1.88× average speedup in the context of four problem instances, which included one regression problem, two classification problems, and one Boolean logic problem [23]. In terms of the common performance measure known as *genetic programming operations per second (GPops/s)* [69], which is defined as the total number of program node evaluations divided by the total runtime of the GP procedure (including time taken for evolution), Chitty's system was notably able to attain a peak rate of 55.7 billion GPops/s for a classification problem as well as 1.411 trillion GPops/s for the chosen Boolean logic problem,[8] which at the time of publication in 2016 were seemingly the highest performance results ever reported [23].

In between the aforementioned GPU studies by Robilliard and Chitty [23, 94], Augusto et al. explored the use of the portable *OpenCL* platform for accelerating GP with both CPU and GPU devices [11]. Notably, a peak GPops/s value of 13.08 billion was achieved with a Nvidia GTX-285 GPU across a set of regression and classification datasets, which constituted a 126× speedup when compared to a sequential implementation using an AMD Phenom II X6 1090T CPU. Within this study, several forms of parallelization were explored for GPU program evaluation, and the one that performed best was similar to the best-performing strategies given by the studies of Robilliard and Chitty—specifically, compute multiple programs across multiple GPU "compute units," and compute multiple fitness cases across multiple threads within each compute unit [11, 23, 94]. Due to the fact that the results of Chitty were significantly better than those by Augusto et al. [11, 23], it seems that the simpler implementations offered by the OpenCL framework—which have less influence on the resulting mappings from software to hardware—may be causing a significant performance reduction, although other factors are likely also at play, e.g., differences in GPU technology, the parameters of the GP experiments, etc.

Following Chitty's work in [23], Staats et al. took a different approach for exploiting general-purpose GPU/CPU execution by using Python and the open-source *TensorFlow* library developed by Google [1]. Their system is known as *KarooGP* [111]. In general, TensorFlow provides generic mechanisms for optimizing/parallelizing computation specified by arbitrary data-flow graphs (DFGs), which is directly

[8] Note that for Boolean logic problems, a form of *bit-level parallelism* can also be exploited [89], which in this case allowed for an additional 32× speedup when using 32-bit words with the relevant GPU.

relevant to tree-based GP since tree-based expressions are a form of DFGs. It is from these capabilities that KarooGP was able to attain notable results for both an Apple MacBook Pro with 4 Intel i7 CPU cores and an NVidia Tesla P100-SXM2 GPU [111]. Specifically, in the context of one regression problem and three classification problems, two of which were large real-world datasets, it was shown that KarooGP was able to significantly outperform a baseline CPU software implementation, with peak speedup values of 877.2× and 851.2× for the CPU and GPU systems, respectively. However, no measurements of GPops/s were provided.

Although implemented with TensorFlow, Baeta et al. note that KarooGP does not leverage some useful updates that have since been made to the TensorFlow library [12]. Most importantly, in [111], KarooGP only leverages the original execution model of TensorFlow. In the past, TensorFlow evaluated DFGs only after first converting them into directed acyclic graphs (DAGs), but newer versions of TensorFlow also allow for the immediate execution of DFGs, which is known as "Eager" mode [3]. In addition, the mechanisms for constructing DAGs have become more efficient with the latest versions of TensorFlow [3]. Notably, even though TensorFlow's DAG conversion process allows for redundant sub-expressions to be cached and represented only once—which can significantly reduce the amount of computation performed—this process can infer significant overhead [12]. Such overhead can be amortized when evaluating a DAG on enough data points, but this overhead can still be pronounced in the context of GP, especially since program graphs are continually evolving [12].

To improve upon the TensorFlow implementation provided by KarooGP, Baeta et al. released their own tool known as *TensorGP* [12], which by default uses the Eager execution model for TensorFlow. Although geared toward evolutionary art contexts, TensorGP can be applied to many GP applications [12]. In the context of the challenging Pagie-1 symbolic regression problem, with a number of fitness cases equal to 16,384 and program depths varying from 4 to 26, average GPops/s values of 264.4 million and 255.4 million were achieved for a GTX TITAN X (12 GB) GPU and Intel Core i7-5930K (@3.7 GHz) CPU, respectively [12, Table 4]. When considering individual program depths, peak GPops/s values of around 300 million were achieved on average for both systems, although the CPU notably performed best for smaller programs and smaller fitness cases whereas the GPU performed similarly for both the smaller and larger extremes. When instead fixing the maximum program depth to be 12 and varying the number of fitness cases from 4,096 to 4,194,304, peak speedup values of 597.7× and 105.2× were provided for the GPU and CPU, respectively, when compared to a baseline software implementation [12, Table 6]. However, for this alternative experiment, GPops/s values were not provided. Finally, a separate experiment was performed in order to showcase how CUDA-optimized and AVX-optimized operators could be generated with TensorFlow in order to achieve further speedups. For their custom *warp* operator [12], peak GPops/s values of around 8.5 billion and 1.5 billion were achieved for the GPU and CPU systems, respectively.

Following from the above, we establish that the abstractions provided by TensorFlow might significantly limit the performance of TensorGP, given that Chitty was able to report 55.7 billion GPops/s for a regression problem with a CUDA-optimized

GPU implementation whereas Baeta et al. report around 300 million GPops/s for a separate regression problem with the standard implementation of TensorGP [12, 23]. This observation seems corroborated by two additional works in which CUDA-optimized systems are introduced and compared against TensorGP using the Pagie-1 symbolic regression problem [101, 131]. In [101], a stack-based GP system known as *cuml* is introduced,[9] and speedups of up to 48.9× are achieved when compared to TensorGP, using an Nvidia DGX-A100 GPU [101, Table 4]. Separately, in [131], an alternative CUDA-based system achieves speedups upwards of 68.6× when using an Nvidia RTX 3050 [131, Table 5]. For both studies, no measurements of GPops/s were reported.

Overall, we establish several important points that follow from the aforementioned GPU studies [11, 12, 23, 94, 111, 131]. Importantly, GP is often a memory-intensive application, and it is challenging to fully leverage the limited memory caches within GPU systems [23]. In terms of performance, executing tree-based programs directly on the GPU by way of a generic interpreter seems more effective than performing intermediate compilation [23, 94]. When using such an interpreter, subdividing the evaluation of multiple programs across multiple high-level "compute units" (e.g., streaming multiprocessors [40]) and then subdividing the evaluation of multiple fitness cases across multiple low-level cores (e.g., CUDA cores [40]) appears to be the best strategy for parallelism [11, 23, 94]. To further lower the overhead associated with program interpretation (e.g., conditional logic used to choose which primitive to execute), low-level L1 memory caches can be better utilized when using a two-dimensional stack that associates an independent stack to each fitness case within a batch [23]. However, when allocating a 2D stack to each thread of execution, we see that only a small amount of memory is available for each stack (e.g., 64 32-bit values for the GPU in [23] if the number of threads is equal to the number of CUDA cores), which can significantly limit the benefits of this approach when compared to CPU devices [22, 23]. Although modern high-end GPU devices can now contain roughly 4× as much L1 cache memory [9], 2D stacks would still be fairly small, implying that similar conclusions can likely be drawn about GPU-based GP in the present day.[10] Ultimately, due to limited strategies for parallelism as well as limited capacities of local memory (especially once subdividing such memory across all low-level cores/threads), the presence of many additional cores within GPU devices when compared to CPU devices is not necessarily useful for GP. Notably, in Sect. 3.1.2, we showcase that recent CPU solutions have achieved better performance when compared to the aforementioned GPU studies.

3.1.2 CPU solutions

In a similar vein to Chitty's work in [22, 23], Langdon extended the *GPQuick* tool to a two-processor, 12-core, 48-thread Intel Xeon Gold 6126 (2.60 GHz) server [67,

[9] We include this stack-based system here since it is almost identical to standard tree-based GP [101].

[10] Higher-level memories (e.g., L2/L3 caches) can be considerably larger, but this stack approach heavily relies on lower-level memory (e.g., L1 caches and registers) to maximize performance.

105].[11] To extend GPQuick, Langdon makes use of the special AVX-512 instructions within the chosen server CPU in order to execute 16 fitness cases at once for each of 48 threads. Notably, the primary goal of this work was not to speed up the GP procedure, but rather to evolve gigantic programs containing around 400 million nodes. As such, the extensions made to GPQuick were specialized to a particular experiment involving the Sextic polynomial regression problem, with 48 fitness cases and a population size of 48, in order to map well to the relevant CPU system. Nevertheless, not only were such humongous programs able to be evolved and evaluated, 139 billion GPops/s were achieved in the process. At the time of publication in 2019, this seemingly constituted the best single-computer (albeit, with two processors) GPops/s results ever reported, with exception to a few results for Boolean logic problems, which naturally exhibit more parallelism (Sect. 3.1.1, footnote 8). However, we additionally note that these results were for simple arithmetic primitives, and that more complex primitives (e.g., sin, cos, etc.) would likely reduce throughput considerably.

Following the work given in [67], Langdon separately introduced a novel tactic for evaluating very large programs that have no side effects [89], which allowed for a significant boost in evaluation performance [68]. In particular, due to the fact that the evolution of very large programs generally causes comparatively small semantic (i.e., behavioral) changes in programs [68, 89], Langdon notes that evaluation can start at modified subtrees and traverse outward toward the root node of the overall tree, stopping whenever it is determined that the new program is semantically equivalent to its predecessor. With this so-called "incremental evaluation," Langdon shows that the evaluation of individuals can take much less time than when evaluating every node. In this context, if the performance measure of GPops/s is allowed to be defined based on the total number of program nodes in the population, rather than the total number of nodes evaluated, some significant speedups occur over previously listed results. To distinguish this atypical definition of GPops/s, we submit that such a measure should be called *effective GPops/s*. Overall, when applying incremental evaluation to the Sextic polynomial problem with an unlisted 16-core Intel CPU, 571 billion effective GPops/s were achieved [68]. Finally, Langdon and Banzhaf extended the ideas presented in [67] to include those presented in [68], which, when using a 3.00GHz Intel Xeon Gold 6136 server, allowed for a significant 1.103 trillion effective GPops/s on the Sextic polynomial problem [70]. However, when distinguishing between effective GPops/s and the original definition of GPops/s, we submit that the value of 139 billion listed in [67] is still the largest GPops/s value reported for non-Boolean domains.

Besides the aforementioned studies, another notable work involves the symbolic regression engine known as *Operon* [19]. In their original study of Operon, Burlacu et al. provide further evidence for leveraging flattened tree representations as well as the most up-to-date SIMD instructions for modern CPU architectures. But perhaps most importantly, in addition to the typical tree-based GP procedures, Operon

[11] GPQuick uses a flattened tree representation for programs as well as an 8-bit machine code for representing program nodes, both of which contribute to computational efficiency [105].

incorporates nonlinear least squares (NLS) optimization in order to speed up the discovery of appropriate constant terms while evolving expressions [58]. Although the computation associated with NLS can significantly slow down the tool [19, 58], a peak GPops/s value of 2.8 billion is achieved in the context of nine symbolic regression problems for an AMD Ryzen 3900X, 12 core, 24 thread (3.8 GHz) CPU system. To showcase the discrepancy in GPops/s values due to the presence of NLS optimization, Burlacu et al. also perform an experiment for measuring the raw evaluation speed of their system excluding evolution and NLS optimization [19]. In particular, when using double-precision arithmetic to evaluate 1,000 randomly generated programs with an average size of 50 nodes on 1,000 random fitness cases, 94.3 billion GPops/s is achieved when using all 24 CPU threads. In addition, Burlacu et al. later make the case that using single-precision arithmetic instead of double-precision arithmetic can allow for roughly a 2× speedup in terms of performance, from which we can extrapolate a potential peak performance value of around 188.6 billion GPops/s. However, we submit here that since no evolution was performed for the aforementioned experiment, the given performance measure is not exactly GPops/s, and we choose to distinguish this type of measure with the term *nodes per second (NPS)*, to align with our experiments described in Sect. 5. Nevertheless, Burlacu et al. also showcase how program evaluation for their system can, on average, take roughly 87% of total runtime when using only 50 fitness cases, and roughly 99% of total runtime when using 1,000 fitness cases. Therefore, we estimate that GPops/s values for full GP runs of Operon including only arithmetic primitives and excluding NLS optimization could largely be similar to the NPS estimates we just listed, which could potentially constitute the highest GPops/s results ever reported.

From the above, we emphasize that if taken at face value, the peak GPops/s value of 2.8 billion for Operon including NLS optimization is considerably lower than the GPops/s values presented by Langdon in [67]. However, it is important to note that this performance measure is not indicative of the resulting quality of program solutions. To showcase this concept, we consider a recent large-scale symbolic regression benchmarking study involving Operon, 13 other symbolic regression tools, and 7 traditional machine learning techniques, including neural networks [64]. Ultimately, in the context of a diverse set of 252 regression problems, it was shown that Operon was a clear winner above all other tools in terms of average solution quality, yet it was not nearly the fastest. (The typical runtime performance of Operon was still relatively fast—close to the median across all tools [64, Fig. 1].) Although the system presented by Langdon [67] was not included in this study, it seems plausible that the NLS optimization could allow Operon to remain superior in terms of solution quality, as further showcased by a recent paper from Burlacu regarding the GECCO 2022 symbolic regression competition [18].

Overall, we emphasize that remarkable performance results were given by the aforementioned CPU studies. Nevertheless, we make note of some potential limitations. First, in regard to Langdon's work involving the *GPQuick* tool [67], we observe that the results given are only in the context of extremely large programs consisting of around 400 million nodes, a small population size of 48, and a single specialized regression problem with 48 fitness cases. Therefore, it is not clear how the extended *GPQuick* tool would perform in the context of smaller programs, larger

population sizes, and a wider range of applications. Separately, in regard to Landgon's work on "incremental evaluation" [68], we establish that it is not clear how the given results would apply to contexts employing significantly smaller programs, although we estimate that the results would be less pronounced, since smaller sizes can often prevent the code bloat that was expected to allow for near semantic equivalence [68]. Lastly, in regard to the original Operon study by Burlacu et al. [19], although the listed performance results may not generalize to all GP configurations, most of the chosen configurations are common [19, Table 4], and larger studies were additionally performed within [18, 64].

3.1.3 FPGA solutions

As already showcased, there exist numerous works that discuss mechanisms for accelerating tree-based GP within the context of CPU and GPU devices. However, for FPGA devices, there are comparatively few works [34, 35, 60, 104].

In the early years of GP (circa 1998), Koza et al. began to explore the potential of FPGA devices [60], although the available FPGA technology greatly limited the feasibility of certain GP applications—for example, applications requiring support for floating-point arithmetic were generally impractical [45]. As a proof-of-concept, Koza et al. employed a Xilinx XC6216 FPGA to evolve sorting networks [60].

Soon after the aforementioned work from Koza et al., in the year 1999, Sidhu et al. utilized a novel *multi-context* FPGA that supported a set of possible circuit configurations [104]. With such a device, various "tree templates" were evolved directly on the FPGA, in a manner that would now be similar to partial reconfiguration (Sect. 2.3). For practical reasons, the tree templates implemented on the chosen FPGA had to represent complete binary trees, which limited the possible program shapes and sizes. Notably, both a Boolean and regression problem were evolved, but the chosen Xilinx XC6264 FPGA technology necessitated a maximum possible program depth of 7, along with other significant constraints.

After the aforementioned publication from Sidhu et al. [104], it appears that work on FPGA systems for tree-based GP stopped until the mid-2010s, when Funie et al. created various FPGA-based GP systems for performing financial trading strategies [34, 35]. Here, we focus on the implementation given within [35], since it is the only one that was purely tree-based. In this work, a fully-pipelined, mixed-precision design was created for accelerating program evaluation. Notably, when compared to a baseline multi-threaded high-frequency trading system executing on two six-core Intel Xeon E5-2640 CPUs, a Maxeler MPCX node containing the aforementioned CPUs and a Stratix V 5SGSMD8N1F45C2 FPGA was able to achieve speedups of up to 22× [35]. We note that Funie et al. restricted programs to be complete binary trees with depths up to 4, although this was said to be sufficient for their chosen application.

It seems that until our recent work [27], which this current paper is extending, there were no other FPGA-based systems for standard tree-based GP. Importantly, we submit that with modern FPGA devices now natively supporting floating-point multiply-adders, along with numerous other resources and the potential for higher clock frequencies [117], improvements of multiple orders of magnitude may be accessible with FPGAs (Sects. 6 and 7). Compared to the architectures given in [35,

60, 104], ours 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. In addition, with the ability to dynamically compile arbitrary expressions directly on the target device, future extensions of our design can also accelerate evolutionary stages directly on chip. Besides dynamic compilation, we 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.

3.2 Other GP systems

Regarding program representations other than the traditional tree, comparatively few works have focused on the performance of program evaluation. In brief, besides CPU studies, there have been GPU and FPGA solutions for Linear GP [21, 32, 46, 77, 126], Cartesian GP [30, 42, 80, 100, 102], and Geometric Semantic GP [38, 76, 120]. In addition, GPU solutions have been created for grammar-guided GP [20] and Tangled Program Graphs [132]. For stack-based GP, at least one GPU study has been conducted [101], although this was for simple stack-based GP [88], and not for a more complex system like PushGP [109]. Finally, we note that the application area of evolvable hardware [129, 130] has leveraged FPGA devices, although this has been with the primary intention of evolving circuits, rather than accelerating the GP procedure via a single circuit, which is the purpose of our architecture presented in Sect. 4.

4 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 performance bottleneck (Sect. 3). In future work, we plan to investigate acceleration of the entire GP process.

Overall, as depicted in Fig. 2, our accelerator leverages a specialized full tree of generic computing resources in order 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, since a new input can continually be pushed in every cycle [43, 44].[12] To further increase throughput, the accelerator also dynamically compiles programs for the tree during evaluation, so that

[12] The latency of a "function unit" ultimately depends on the primitive set, but for our experiments it was anywhere from tens to hundreds of cycles (Sect. 5). Importantly, latency is amortized when operating on enough data/programs, and what matters more is throughput, as we showcase in our results (Sect. 6).

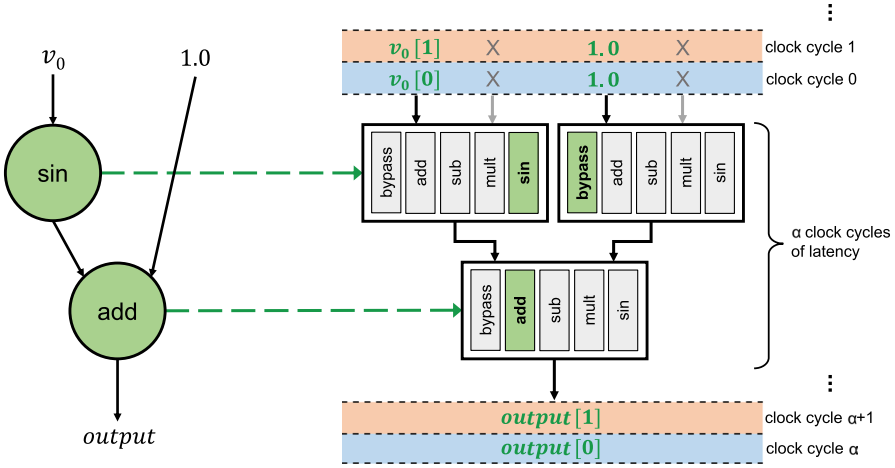


Fig. 2 A portrayal of how our GP accelerator can parallelize the evaluation of different data points and different solutions *every clock cycle* via a reconfigurable tree pipeline. Each node of the pipeline can perform any function within the GP primitive set, as well as a bypass, which allows for arbitrary program shapes

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.

The accelerator architecture currently consists of four major components, as shown in Fig. 3. The *program memory* (Sect. 4.1) stores candidate program solutions, where each candidate is encoded in a language defined by the specification of

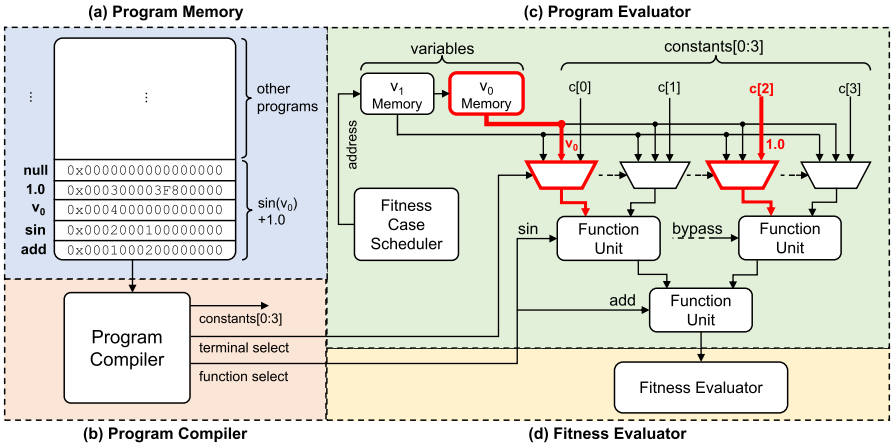


Fig. 3 High-level overview of the accelerator architecture. The accelerator stores programs (e.g., $\sin(v_0) + 1.0$) in **a** *program memory*, which are dynamically compiled by the **b** *program compiler* into configuration data for the **c** *program evaluator*. The program evaluator uses a reconfigurable function tree pipeline to execute a compiled expression for a set of fitness cases, resulting in a set of outputs to which the **d** *fitness evaluator* compares a set of desired outputs

a particular primitive set. The *program compiler* (Sect. 4.2) 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* (Sect. 4.3). 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, as shown in Fig 2. Finally, the *fitness evaluator* (Fig. 3d) compares the output of a current program to the relevant target data by way of some metric, which is *root-mean-square error* for this work.

4.1 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:

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 (Sect. 4.2).

We encode program expressions via a prefix (i.e., Polish) notation. In essence, such a representation flattens tree-based programs into a linear structure [89]. For example, Fig. 3a shows how our architecture could support the program $\sin(v_0) + 1.0$ 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. Due to the presence of the null word, any additional program can immediately follow a previous program within memory. Note that, as with standard prefix notation, children nodes are not necessarily placed next to one another in contiguous memory.

4.2 Program compiler

We use the term "compile" to refer to the generic process of translating a source language to a target language. Within our system, the *program compiler* is specialized circuitry that dynamically compiles prefix-based tree expressions—encoded with the language defined in Sect. 4.1—into low-level configuration data relevant

to the program evaluator.[13] Compilation executes in parallel to program evaluation, so that it may be possible to switch between programs within a single clock cycle. Currently, the program compiler is implemented as a finite-state machine (FSM) that continually writes configuration information into a configuration buffer. The configuration data contains three major components (Fig. 3b and c): (1) *function select* values that configure individual function units within the program evaluator (Sect. 4.3), (2) *terminal select* values that dictate whether a variable or constant terminal is connected to a corresponding tree input, and (3) *constant* values that specify constant terminals.

To compile a program, the program compiler conducts a pre-order traversal on a model of the relevant function tree (Sect. 4.3), so that compilation can execute in parallel to program evaluation. Ultimately, the objective is to populate three memories with the three types of configuration information specified in the previous paragraph. Each memory allocates enough locations to store information for the full physical tree, but any arbitrary program shape is supported, and only the relevant memory locations are configured during compilation. The memory for function select values is organized in a prefix fashion, i.e., index i corresponds to the function select value for the function unit whose index is i when listed in prefix notation (Sect. 4.3). Separately, the memories for terminal select and constant values are organized such that index i corresponds to the function tree input (i.e., leaf node) whose index is i when the inputs are listed in a linear fashion, from leftmost to rightmost leaf.

When compiling a program, if the current node under consideration has a depth smaller than the current node in the physical tree—where the depth of the program node is specified by the relevant portion of the machine code (Sect. 4.1)—the compiler first traverses to the top-leftmost portion of the physical tree that has this specified depth, where left children that are encountered along the way are configured to perform the bypass function, so that the relevant node output will be bypassed through the lower levels of the physical tree. In addition, throughout compilation, we avoid backtracking through previously visited nodes by utilizing several predefined lookup tables that identify the indices of the next function node and next terminal input within the physical tree given a current function node index [28]. Once compilation has finished, the three pertinent sets of outputs—function selects, terminal selects, and constant values—are available in the first of two sets of buffers, where the first set can shift into the second set in order to actively drive the signals of the program evaluator. If the program evaluator is ready to accept a program (Sect. 4.3), then the configuration information from the first set of buffers is shifted into the second set of buffers. Otherwise, the configuration is reserved until the program evaluator is ready. For more details, refer to the Python implementation given in our codebase [28].

Notably, depending on the shapes/sizes of programs being compiled and the number of fitness cases that are to be streamed into a function tree (Sect. 4.3), the

[13] Since we are ultimately translating from one machine code to another, we could also potentially refer to our compilation as a form of "microcode translation" [44], although we choose not to do so.

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.

4.3 Program evaluator

The program evaluator (Fig. 3c) 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 main motivation behind this architecture is that, with tree-based GP, every program expression is a tree. Therefore, if the architecture provides a physical tree of generic resources such that each resource is capable of computing all function primitives, then the tree can compute *entire programs* in parallel. However, even more notably, if we additionally design the generic resources (i.e., function units) to be *pipelined*, then the architecture can generate an output for an entire program *every clock cycle* after some initial latency, as shown in Fig. 2. Although pipelining the tree precludes the existence of arbitrary control structures, such structures are usually unnecessary [18, 64], and pipelining enables data-level and/or function-level parallelism [43, 44]. Lastly, as described in Sect. 4.2, our architecture includes a compiler in hardware that translates compact prefix-based expressions into machine codes for the tree while it is evaluating, so that the tree may switch between programs *within a single clock cycle*. Importantly, generating outputs for entire programs every clock cycle and changing programs within a single cycle are forms of parallelism that have not been achieved with general-purpose CPU/GPU architectures.

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 and 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 (or identity) function is used to feed the leftmost input of a function unit to its output through registers 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 by the target device. Notably, with such primitives, we can always achieve a new output every cycle after some initial latency, and latency will be amortized when operating on enough data/programs. In the future, we plan to implement some support for variable-latency functions, so that we may pursue some more complex GP applications, such as those regarding general program synthesis [59]. For now, to ensure that all levels of the tree output results at the same time, we give each function primitive the same latency, utilizing some delay register(s) when required.

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. At runtime, an external entity (e.g., a processor) loads fitness case data into the variable memories using memory-mapped I/O. (Constants are specified within program machine codes, as described in Sect. 4.1.) Next, the program evaluator waits until the program compiler provides configuration information for a program expression. Upon receiving such data, the program evaluator triggers the *fitness case scheduler*, which begins streaming all relevant fitness case data (i.e., all relevant variables/constants) through the function tree, as dictated by the program compiler. Upon inputting the last fitness case into the function tree, the fitness case scheduler signals the program compiler to provide a following program, if any are available. Since the program compiler operates independently of the program evaluator, it is possible for the function tree to start evaluating a new program *one cycle* after input data for a final fitness case is passed into the function tree.

As of now, each variable memory fans out to every input of the tree to maximize throughput. Since multiple tree inputs will only ever use the same fitness case of a particular variable at any given time [89], we simply fan out the same variable memory to every function tree input. Although it is possible for this strategy to cause routing bottlenecks when compiling the design, we have been able to support multiple variables in combination with tree structures having well over a hundred inputs (Sect. 6). For constants, we allocate a 32-bit register for each tree input, which can be configured by the program compiler whenever appropriate (Sect. 4.2). For variable memory, we currently target on-chip embedded memory—rather than larger, off-chip RAM—so that we may maintain the highest possible memory bandwidth. Although we already are able to support relatively large amounts of fitness cases and constants via embedded memory (Sect. 6), we expect to eventually target newer FPGA systems with on-board *high-bandwidth memory* [52], so that we may readily consider applications demanding even larger amounts of data. More common DDR memory may also be viable, although we save this for future work.

5 Experiments

In this section, we detail our design of experiments, where the overall goal of these experiments was to compare our architecture (Sect. 4) with the core evaluation engines given by three tree-based GP software tools: *DEAP* [29], *TensorGP* [12], and *Operon* [19]. The computing technologies we use are listed in Sect. 1.

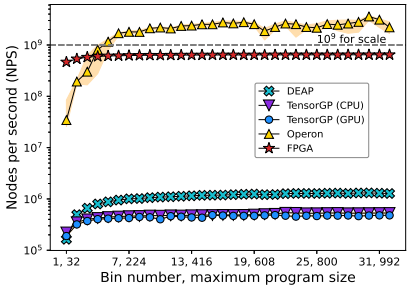
Note that we do not compare against other FPGA-based GP systems due to the fact that no comparable tree-based system is actively maintained and/or open-source. In addition, even if such a system was available, the relevant code would likely not be readily portable to our chosen FPGA, in which case considerable effort would need to made to re-implement the other FPGA solution, which is not the aim of this study. Rather, our goal is to assess the general viability of FPGA acceleration for tree-based GP when compared to modern CPU and GPU solutions. Nevertheless, to provide the reader with some initial comparisons between FPGA-based GP systems, we detailed several related works within Sect. 3.1.

5.1 Comparison measures

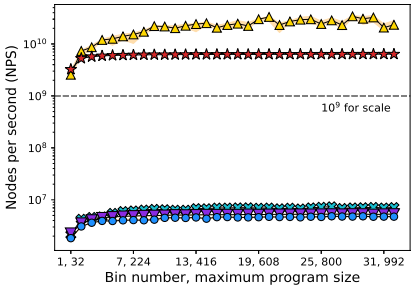
We use three measures to compare each device/tool: (1) *nodes per second (NPS)*, (2) *(average) electric power*, and (3) *nodes per watt (NPW)*. The NPS measure is defined as a total number of GP nodes divided by a total runtime.[14] Electric power is defined as a total amount of electrical energy divided by a total runtime. Finally, the NPW measure is defined as NPS divided by power. Overall, NPS provides an estimate of performance (i.e., throughput) in terms of GP nodes, power provides an estimate of device energy consumption per unit time, and NPW relates performance to power. Notably, the totals used in the NPS and power calculations can be for multiple experiments as long as only ratio is computed, similar to how it has been documented for the *floating-point operations per second (FLOPS)* performance measure [107]. More specifically, for a single-number performance/power measure, we would like an estimate to be (1) proportional to a total amount of work and (2) inversely proportional to a total amount of runtime [107], since this aligns with Amdahl's Law [8], whereas averaging across multiple performance/power ratios can be arbitrarily outweighed by a single ratio, no matter the total runtime [107]. With meaningful aggregate performance (e.g., NPS) and power estimates, meaningful aggregate performance-per-watt (e.g., NPW) estimates can also be computed.

For each tool and for each combination of various primitive sets and numbers of fitness cases (detailed in the following subsections), we conduct a performance experiment in which we measure evaluation runtimes for each of 32 *program bins*, with each bin consisting of 512 distinct random programs. From runtimes, we calculate nodes-per-second (NPS) estimates by dividing a relevant total number of nodes by a corresponding runtime. Due to significant performance differences between each GP tool, we use a different total number of executions for certain tools. For Operon and TensorGP, we run the set of experiments 11 times. For DEAP, which executed the set of bin experiments in about 44 h (due to poor scaling at larger numbers of fitness cases), we run each experiment just once. Ultimately, running each DEAP experiment once is justified by the fact that any fluctuations in runtime due to other system processes are insignificant when compared to the processes used by the experiments, as indicated by the narrow 75th

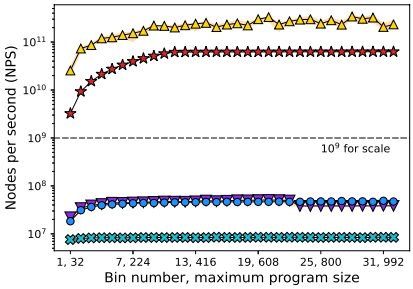
[14] Frequently, the statistic *GP operations per second (GPops/s)* is used to compare the runtime performance of GP tools [23], but we use NPS to emphasize that our tests do not leverage evolution.



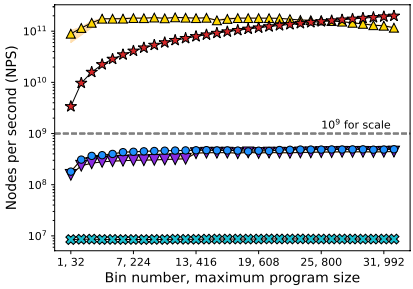
(a) Primitive set nicolau_a, 10 fitness cases. **Max value: 3.59 billion NPS**.



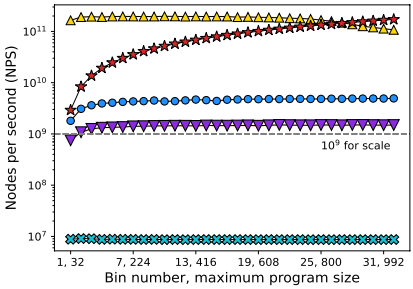
(b) Primitive set nicolau_a, 100 fitness cases. **Max value: 34.0 billion NPS**.



(c) Primitive set nicolau_a, 1K fitness cases. **Max value: 340 billion NPS**.



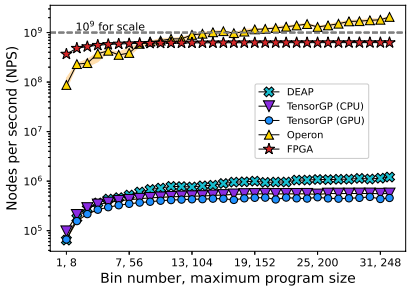
(d) Primitive set nicolau_a, 10K fitness cases. **Max value: 199 billion NPS**.



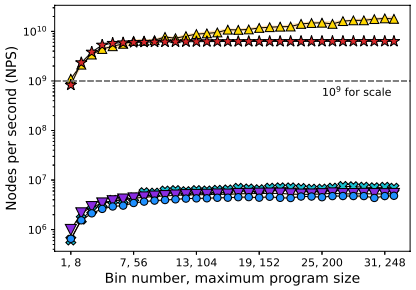
(e) Primitive set nicolau_a, 100K fitness cases. **Max value: 197 billion NPS**.

Fig. 4 Median sample nodes per second (NPS) vs. program bin number and maximum program size, for the nicolau_a primitive set. For Operon and TensorGP, NPS values corresponding to the 75^{th}/25^{th} percentiles for runtime are also plotted. Note that the legend from **a** applies to all sub-figures, and note the use of a log scale

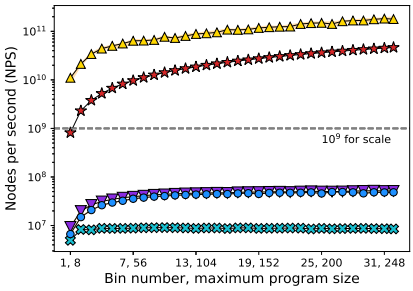
/25th percentile regions for the runtimes of TensorGP given in Figs. 4, 5 and 6. Lastly, for the FPGA, we also run the set of performance experiments only once, since the relevant evaluation circuitry has a deterministic runtime, which is tracked by special clock cycle counters allocated within the device.



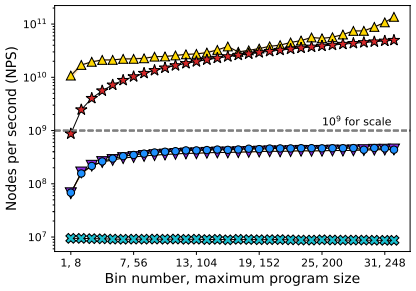
(a) Primitive set nicolau_b, 10 fitness cases. **Max value: 2.07 billion NPS**.



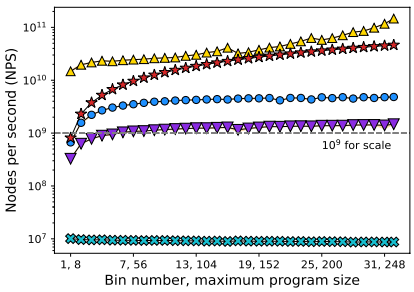
(b) Primitive set nicolau_b, 100 fitness cases. **Max value: 18.3 billion NPS**.



(c) Primitive set nicolau_b, 1K fitness cases. **Max value: 183 billion NPS**.



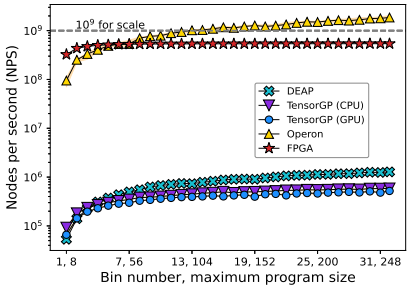
(d) Primitive set nicolau_b, 10K fitness cases. **Max value: 136 billion NPS**.



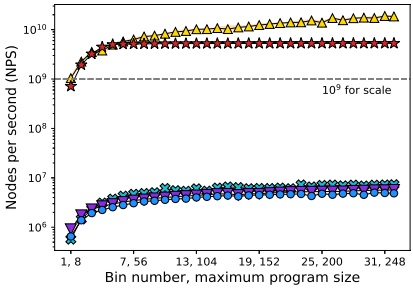
(e) Primitive set nicolau_b, 100K fitness cases. **Max value: 147 billion NPS**.

Fig. 5 Sample median nodes per second (NPS) vs. program bin number and maximum program size, for the nicolau_b primitive set. For Operon and TensorGP, NPS values corresponding to the 75th/25th percentiles for runtime are also plotted. Note that the legend from **a** applies to all sub-figures, and note the use of a log scale

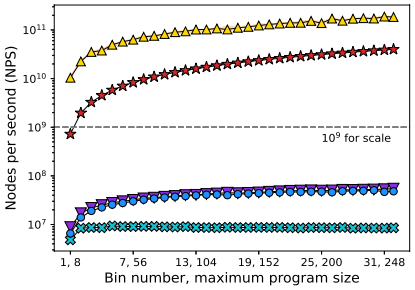
In addition to the performance experiments, we perform a separate set of power experiments using the same tools and combinations of parameters. For these experiments, we perform the same numbers of runs as previously prescribed, except we perform 11 runs for the FPGA, since power draw is not deterministic. Importantly, we separate power experiments from performance experiments due to the fact that



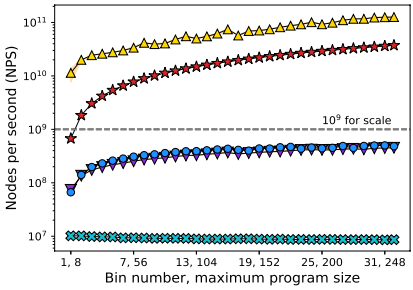
(a) Primitive set nicolau_c, 10 fitness cases. **Max value: 1.86 billion NPS**.



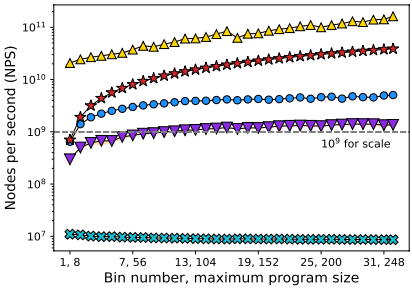
(b) Primitive set nicolau_c, 100 fitness cases. **Max value: 18.8 billion NPS**.



(c) Primitive set nicolau_c, 1K fitness cases. **Max value: 188 billion NPS**.



(d) Primitive set nicolau_c, 10K fitness cases. **Max value: 126 billion NPS**.



(e) Primitive set nicolau_c, 100K fitness cases. **Max value: 162 billion NPS**.

Fig. 6 Sample median nodes per second (NPS) vs. program bin number and maximum program size, for the nicolau_c primitive set. For Operon and TensorGP, NPS values corresponding to the 75^{th}/25^{th} percentiles for runtime are also plotted. Note that the legend from **a** applies to all sub-figures, and note the use of a log scale

CPU-based measurements for power/energy can have a significant effect on the results of individual runs for the CPU-based Operon, since the runtime/energy of an individual Operon experiment is often a fraction of the runtime/energy incurred by an internal measurement, which can cause such measurements to become largely skewed. These issues are also relevant to DEAP and the CPU version of TensorGP,

but they are less pronounced than with Operon, since the runtime of these other tools is ultimately shown to be much higher. Separately, such measurement issues are not as relevant to the GPU and FPGA, since the bulk of the computation for power/energy measurements for these devices is handled by the CPU. However, we aim to establish as consistent of a solution as possible for all devices, which we describe in the following.

For the CPU-based tools, we measure power by independently measuring runtime and energy consumption of each program bin experiment, where we reference special hardware counters for energy, as detailed in our code [28, 73]. In order to ensure that measuring runtime and energy has a practically-negligible effect on power results, we choose to run each program bin experiment continually such that we can bound the runtime and energy of an internal power measurement to be at most 1% of the total runtime/energy for an experiment, so that any effect on resulting power values is also bounded by 1%.[15] We do not use any smaller percentage for an upper-bound due to practical implications on total runtime, as we showcase below. From a sample size of 1000 CPU power measurements, we establish that 17 s is a reasonable runtime for each bin experiment in order to have this upper-bound criterion be met by each tool [28]. However, to also minimize possible power discrepancies at the beginning of an experiment, we ramp up the CPU by running the relevant experiment for at least one second before beginning a power measurement, for all tools except DEAP.[16] Therefore, each bin experiment is generally to take at least 18 s (at least 17 s for DEAP), which ultimately means that the total set of power experiments for each CPU-based tool—across all bins and across all runs—is often at least 26.4 h.[17] Following from the established 1% bound on power measurements, we ultimately find that there is a potential swing of about $\pm$ 2.8 Watts for the CPU-based measurements [28].

For the GPU and FPGA, there do not exist special hardware counters for tracking energy consumption, and we instead take *instantaneous* power measurements using firmware provided by Nvidia and Intel [28, 49, 85]. To estimate average power for each bin experiment, we simply average a set of instantaneous power measurements,[18] which is reasonable since average power is equivalent to the integral of instantaneous power over time [48]. To remain consistent with the CPU-based tools while also remaining practical in terms of runtime, we utilize the same 1-second ramp-up and 17-second loop previously established, and we aim to most accurately compute average power by taking as many instantaneous power measurements as

[15] For DEAP, in which some bin experiments would take more an hour and the relevant hardware energy counters would overflow multiple times, we had to create a more involved process, which made it more difficult to estimate the effect that the measurements had on runtime/energy. Instead, we estimated the worst-case power difference by running two separate experiments for the least complex bin experiment (for which measurements should have the largest effect) and ensured that this was within 1% [28].

[16] We choose not to perform this ramp-up for DEAP, since many bin experiments can take over an hour, in which case the total runtime of power experiments for DEAP would be more than 80 h.

[17] In reality, the runtime can be significantly longer for TensorGP and DEAP, since running some bin experiments once or twice can take longer than either 17 s or 1 s.

[18] Note that this is *not* the same thing as averaging *average power* ratios (Sect. 5.1).

possible within the 17-second loop. Ultimately, we find that this means *at least* 489 and 597 instantaneous power measurements are taken for each FPGA and GPU bin experiment, respectively [28], where we allow the particular number of measurements to differ between bins and tools, which is reasonable given stochastic effects and given that we are averaging over all instantaneous power measurements. Overall, as with the CPU-based tools, the runtime of all GPU/FPGA power experiments is at least 26.4 h.

Lastly, following from the aforementioned performance and power experiments, we estimate nodes-per-watt (NPW) values. Notably, although runtimes are different between the two sets of experiments—which is appropriate due to the challenges in measuring power for Operon, as described above—both the performance and power measurements correspond to the same program bins, which allows us to meaningfully consider a division between the relevant NPS and power ratios.

5.2 Primitive sets

Three distinct primitive sets are chosen. These primitive sets are inspired by work from Nicolau et al. [83], and, as such, are respectively named nicolau_a, nicolau_b, and nicolau_c. The first primitive set contains 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 [83]. The second primitive set contains the same functions as the first, but also includes sin and tanh. Lastly, the third primitive set contains the same functions as the second, but also includes exp, log, and sqrt, where log and sqrt were "protected" in the typical GP sense [59, 89]. We choose these specific primitive sets since they are relevant to symbolic regression [59, 64, 89], our primary target domain. To implement a primitive set, we utilize standard Intel floating-point IP for our chosen FPGA [51]. For functions that are not readily available, e.g., the aq function described above, we simply chain together the relevant IP blocks.

For a primitive set containing function set F, $|F| - 1$ terminal variables and one ephemeral random constant are employed so that the program generator (Sect. 5.3) can consistently construct programs in which the proportion of functions/terminals is approximately 0.5, so that the average runtime of a particular primitive set is not dictated by having more of one primitive type. Thus, there are 3, 5, and 8 variables for the defined nicolau_a, nicolau_b, and nicolau_c primitive sets, respectively.

5.3 Program generation

For each primitive set, we construct a set of 32 program bins, each containing 512 random programs with sizes in some fixed range, where the particular range is dependent on the bin and primitive set, as we describe further below. The maximum possible program depth/size is chosen to be the largest that the target FPGA can support while also supporting up to 100,000 fitness cases for each of

the relevant variable terminal memories (Sect. 4.3). We manually determine the maximum possible function tree depth for each primitive set through multiple hardware compilations; ultimately, depth values 8, 6, and 6 are respectively chosen for nicolau_a, nicolau_b, and nicolau_c. For a maximum possible function tree depth d, it is possible to support a program depth of up to $d + 1$ (Sect. 4.3), which corresponds to a maximum possible program size of $2^{d+2} - 1$, since every primitive set contains functions with arity of at most two. For a maximum size s, we evenly divide the range of program sizes [1, s] into 32 bins.

To randomly generate program expressions for each set of bins—which are kept the same for each GP tool—we utilize DEAP [29]. We choose DEAP for this task because it is simple to extend. DEAP offers, by default, several classic GP program initialization algorithms: *full*, *grow*, and *ramped half-and-half* [59, 89]. Unfortunately, via the original version of these algorithms, the size of a generated program is completely random beyond a specified depth constraint, which makes it too cumbersome to generate 512 distinct random programs for the bin structures established above. To circumvent this issue, we employ a custom version of the *grow* method that allows for the specification of a minimum/maximum program size, from which a random value is chosen in a uniform manner [28]. Overall, choosing 512 distinct random programs for each bin structure means that 16,384 programs are used to evaluate each of the three primitive sets, which corresponds to a total of 49,152 random programs.

5.4 Fitness cases

We use the term "fitness case" in the conventional sense [89], which is to refer to a data point corresponding to all relevant variables, which is multi-dimensional in general. As described in Sect. 5.2, we utilize 3, 5, and 8 variables for the nicolau_a, nicolau_b, and nicolau_c primitive sets, respectively. For each primitive set, we use five amounts of fitness cases: 10, 100, 1,000, 10,000, and 100,000. For each number of fitness cases, we randomly generate input/target data in the range [0, 1), and we use 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 chosen (1) primitive sets, (2) maximum number of variables, and (3) maximum number of fitness cases, which, as shown in [64], allows for many.

6 Results

Figures 4, 5, and 6 compare the performance of each evaluation engine in terms of nodes-per-second (NPS) values, for all fifteen combinations of the three primitive sets and five numbers of fitness cases. For each combination, we plot results for the five GP tools: (1) DEAP, (2) TensorGP with CPU, (3) TensorGP with GPU, (4) Operon, and (5) our FPGA-based accelerator. In particular, for each plot

Table 2 Aggregate NPS values across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon	FPGA
10	9.836e+04	4.629e+04	4.084e+04	**1.319e+08**	5.599e+07
100	6.032e+05	4.633e+05	4.074e+05	**1.473e+09**	5.492e+08
1000	7.736e+05	3.963e+06	4.107e+06	**1.473e+10**	3.366e+09
10000	7.926e+05	3.624e+07	4.146e+07	**8.744e+09**	4.138e+09
100000	7.958e+05	1.249e+08	4.126e+08	**9.317e+09**	3.929e+09
All Tests	7.946e+05	6.311e+07	9.127e+07	**9.191e+09**	3.895e+09

Additionally, the last row provides an average across all experiments. The best values are given in bold. See Sect. 5.1 for more details on such aggregate ratios

Table 3 Aggregate power values across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon	FPGA
10	117.9	91.8	91.5	210.6	**61.0**
100	195.5	91.4	90.6	212.9	**65.9**
1000	247.5	91.5	90.9	247.3	**78.8**
10000	251.4	91.9	91.2	268.8	**81.7**
100000	251.8	106.4	93.6	274.3	**80.8**
All Tests	249.7	95.0	91.5	242.8	**73.7**

Additionally, the last row provides an average across all experiments. The best values are given in bold. See Sect. 5.1 for more details on such aggregate ratios

Table 4 Aggregate NPW values across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon	FPGA
10	8.340e+02	5.041e+02	4.466e+02	6.265e+05	**9.180e+05**
100	3.085e+03	5.068e+03	4.495e+03	6.921e+06	**8.337e+06**
1000	3.126e+03	4.332e+04	4.519e+04	**5.958e+07**	4.272e+07
10000	3.152e+03	3.945e+05	4.548e+05	3.253e+07	**5.063e+07**
100000	3.161e+03	1.174e+06	4.408e+06	3.397e+07	**4.864e+07**
All Tests	3.181e+03	6.639e+05	9.970e+05	3.786e+07	**5.288e+07**

Additionally, the last row provides an average across all experiments. The best values are given in bold. See Sect. 5.1 for more details on such aggregate ratios

representing a GP tool and for each program bin containing 512 programs, we mark the NPS value corresponding to the median runtime of the relevant set of runs, where the number of fitness cases changes between subfigures. In addition, for the tools in which performance experiments were run more than once (i.e., TensorGP and Operon), regions for NPS values associated with the 75th and 25th percentiles of runtime are plotted around each sample point; only a few of such percentile regions are noticeable, meaning that most runtimes vary little between multiple runs.

Table 5 NPS improvement for the FPGA versus all other tools, across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon
10	569.3×	1210×	1371×	0.4245×
100	910.3×	1185×	1348×	0.3727×
1000	4351×	849.5×	819.7×	0.2285×
10000	5220×	114.2×	99.80×	0.4732×
100000	4937×	31.45×	9.522×	0.4217×
All Tests	4902×	61.72×	42.67×	0.4238×

Additionally, the last row provides an average across all experiments. See Sect. 5.1 for more details on such aggregate ratios

Table 6 Power improvement for the FPGA versus all other tools, across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon
10	1.93×	1.51×	1.50×	3.45×
100	2.97×	1.39×	1.38×	3.23×
1000	3.14×	1.16×	1.15×	3.14×
10000	3.08×	1.12×	1.12×	3.29×
100000	3.12×	1.32×	1.16×	3.40×
All Tests	3.39×	1.29×	1.24×	3.30×

Additionally, the last row provides an average across all experiments. See Sect. 5.1 for more details on such aggregate ratios

Table 7 NPW improvement for the FPGA versus all other tools, across all primitive sets, for each number of fitness cases

No. Cases	DEAP	TGP (CPU)	TGP (GPU)	Operon
10	1101×	1821×	2056×	1.465×
100	2702×	1645×	1855×	1.205×
1000	13670×	986.1×	945.2×	0.7170×
10000	16060×	128.3×	111.3×	1.556×
100000	15390×	41.44×	11.04×	1.432×
All Tests	16620×	79.65×	53.04×	1.397×

Additionally, the last row provides an average across all experiments. See Sect. 5.1 for more details on such aggregate ratios

In addition to the aforementioned figures, Tables 2, 3 and 4 provide aggregate estimates for NPS, power, and nodes-per-watt (NPW), where the provided estimates are relevant to *all* performance/power experiments associated with a given number of fitness cases. More specifically, for a given number of fitness cases, estimates are computed by dividing two totals, where each total is relevant to *all* performance/power experiments associated with that number of fitness cases, i.e., all runs of each program bin, across all primitive sets. For example, aggregate NPS estimates are computed by forming the total number of GP nodes relevant to *all* performance experiments corresponding to a given number of fitness cases—i.e., all runs of each program bin, across all primitive sets—and then by dividing this total by the total runtime associated with all such node computation. As we described in

Sect. 5.1, computing single-number estimates of our chosen measures for multiple experiments is meaningful as long as we compute only ratio. Ultimately, we provide these aggregate estimates in order to identify general trends for performance, power, and performance-per-watt, all within a reasonable amount of space. In addition, Tables 5, 6 and 7 provide ratios of improvement for NPS, power, and NPW when comparing the FPGA to all other tools. For more fine-grained trends involving program bins and primitive sets, we refer the reader to our code [28], as well as to Figs. 4, 5, and 6, which we consider further in the following.

Overall, in terms of raw performance (i.e., NPS), our accelerator mostly performed second-best behind Operon, but due to significant power improvements (Table 6), the FPGA was generally superior to all other tools in terms of performance-per-watt (i.e., NPW), as shown in Table 7. On average, across all NPS experiments, the FPGA was 4,902× faster than DEAP, 61.72× faster than TensorGP executing with the CPU, 43.14× faster than TensorGP executing with the GPU, and 0.4238× faster (i.e., 2.36× slower) than Operon. In terms of power, the FPGA was 3.39× better than DEAP, 1.29× better than the CPU-based TensorGP, 1.24× better than the GPU-based TensorGP, and 3.30× better than Operon. When relating performance to power through the performance-per-watt (i.e., NPW) measure, the FPGA was on average 16,620× better than DEAP, 79.65× better than the CPU-based TensorGP, 53.04× better than the GPU-based TensorGP, and 1.397× better than Operon.

In regard to Operon, our accelerator also sometimes obtained the highest NPS values, e.g., for the larger programs and larger number of fitness cases with the nicolau_a primitive set (Fig. 4), and for the smaller programs and smaller numbers of fitness cases across all primitive sets. In some other instances, our accelerator performed very similarly to Operon in terms of NPS, e.g., for the medium-sized programs and medium-sized numbers of fitness cases with nicolau_b (Fig. 5). In general, the speedups we achieved stemmed from the fact that our accelerator could fully parallelize program evaluation once a program was compiled for the physical tree. For larger numbers of fitness cases (e.g., 10K and 100K), compilation was completely amortized after the first program (Sect. 4.2), 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 use a lower clock frequency for nicolau_c in order to support more complex primitives, which allowed nicolau_b to have better performance. Such discrepancies in clock frequency when performing multiple design compilations also explain the small drop in power for the FPGA between 10K and 100K fitness cases (Table 3).

In regard to TensorGP, our accelerator was able to consistently outperform a modern GPU device, where our results align with prior work authored by the developers of TensorGP [12]. Following from the power results in Table 3, we note that TensorGP appears to struggle with fully utilizing the relevant CPU and GPU, which are rated for a maximum power draw of 280 and 350 Watts, respectively. In a separate set of experiments for TensorGP not reported here, we noticed that for extremely large numbers of fitness cases (e.g., 16 million), CPU/GPU utilization and power draw finally tended toward maximum values. The fact that the GPU is not utilized more for our particular set of experiments is not necessarily

Table 8 FPGA design statistics for the fifteen compilations performed

	No. Cases	Clock	Logic	DSP	Memory
Nicolau A	10	194	522,520 (56.0%)	3,328 (57.8%)	3,964 (33.8%)
	100	191	522,821 (56.0%)	3,328 (57.8%)	3,965 (33.8%)
	1000	189	522,599 (56.0%)	3,328 (57.8%)	3,968 (33.9%)
	10000	197	525,539 (56.3%)	3,328 (57.8%)	4,041 (34.5%)
	100000	170	524,841 (56.2%)	3,328 (57.8%)	4,744 (40.5%)
Nicolau B	10	189	591,687 (63.4%)	3,118 (54.1%)	2,659 (22.7%)
	100	190	591,150 (63.4%)	3,118 (54.1%)	2,660 (22.7%)
	1000	185	594,730 (63.7%)	3,118 (54.1%)	2,665 (22.7%)
	10000	197	595,602 (63.8%)	3,118 (54.1%)	2,774 (23.7%)
	100000	184	599,027 (64.2%)	3,118 (54.1%)	3,829 (32.7%)
Nicolau C	10	163	717,705 (76.9%)	5,086 (88.3%)	3,811 (32.5%)
	100	159	729,097 (78.1%)	5,086 (88.3%)	3,812 (32.5%)
	1000	159	717,885 (76.9%)	5,086 (88.3%)	3,820 (32.6%)
	10000	148	732,517 (78.5%)	5,086 (88.3%)	3,983 (34.0%)
	100000	154	720,037 (77.2%)	5,086 (88.3%)	5,566 (47.5%)
Average		178	613,850 (65.8%)	3,844 (66.7%)	3,751 (32.0%)

Clock represents the resulting clock frequency, which is given in Megahertz. *Logic* represents the number of consumed logic resource blocks, which are known as "adaptive logic modules (ALMs)" in the context of the relevant Intel FPGA. *DSP* represents the number of consumed "digital signal processing" blocks, which effectively are floating-point multiply-adders in this application. *Memory* represents the number of consumed "M20K" memory resources, which are the largest embedded memory blocks given by the relevant Intel FPGA. Lastly, note that the size of the designs could not be increased due to challenges regarding exponential growth, as described in Section 7.1

limited to the implementation of TensorGP, due to general challenges associated with GPU-based GP (Sect. 3.1.1). However, given our results for the other CPU-based tools, the fact that the CPU is not more utilized by TensorGP may indicate some inefficiencies with employing TensorFlow for smaller datasets (Sect. 3.1.1). Again, we emphasize that our performance results for TensorGP align with prior work authored by the developers of TensorGP [12], which should remove any concerns of our experiments somehow causing a significant issue.

In regard to DEAP, our accelerator was able to consistently outperform the relevant CPU system by multiple orders of magnitude, especially for larger numbers of fitness cases, where three orders of magnitude was achieved for performance, and where four orders of magnitude was achieved for performance-per-watt. Interestingly, DEAP sometimes performed better than TensorGP for the smaller numbers of fitness cases, although TensorGP scaled much better with larger numbers of fitness cases.

All in all, in terms of raw performance, we emphasize 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 our system consistently outperformed a modern GPU running the recent

high-performance TensorGP tool, as well as the same CPU system running both TensorGP and the popular DEAP tool. When additionally considering power and performance-per-watt measures, the FPGA was shown to be generally superior to all other systems. Importantly, in the following section, we detail six future extensions that could allow for at least a 64–192× speedup over our initial design. Therefore, taken altogether, our initial results showcase considerable potential for FPGA-based GP, and there is clear motivation to continue this line of work.

To conclude this section, we include some FPGA design statistics in Table 8, which resulted from the fifteen compilations performed for the fifteen combinations of primitive set and number of fitness cases (Sect. 5). The table provides details for clock frequency and the utilization of ALMs (i.e., logic blocks), DSPs (i.e., floating-point multiply-add blocks), and M20Ks (i.e., embedded memory). Notably, the resulting clock frequencies are fairly low, due to the current accelerator architecture not being fully optimized for timing, and due to the designs taking up a fairly large percentage of the total FPGA area. Importantly, the size of the designs could not be increased due to challenges regarding exponential growth, as we describe in the next section.

7 Current challenges and potential optimizations

Below, we list three challenges of our initial accelerator architecture. Then, we present six potential optimizations that could alleviate the three challenges and allow for an updated accelerator to achieve a speedup over our current design by at least 64–192×, in addition to allowing support for larger program depths/sizes. After this, we discuss some important considerations that follow from the present challenges.

7.1 Current challenges

Broadly speaking, there are three main challenges with the current architecture:

1. *Exponential Growth.* For an m-ary function tree with $m > 1$, where m is the maximum function arity of the primitive set, 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.)
2. *Function Unit Complexity.* For a function tree (Sect. 4.3) 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 ALM and DSP blocks available within the target FPGA device, as shown in Table 8. More spe-

cifically, due to the challenge of exponential growth (detailed in the previous list item), we could not scale up tree depth any further, since this would require roughly double the amount of ALM/DSP resources, which is impossible for the current FPGA device.
3. *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 at most $\frac{1}{|F|}$, and likely worse since programs are often not full trees. The utilization of low-level device primitives (e.g., DSPs) can be significantly lower, depending on the function implementation.

7.2 Potential optimizations

We note that the following optimizations are independent from one another,[19] and, thus, if all could be achieved simultaneously, a speedup between $2 \cdot 2 \cdot 4 \cdot 2 \cdot 2 = 64\times$ and $2 \cdot 6 \cdot 4 \cdot 2 \cdot 2 = 192\times$ could be achieved over our current accelerator, where additional speedups may be possible when considering timing optimization. We highlight that these potential optimizations, when paired with our results in Sect. 6, provide considerable motivation for the continued exploration of FPGA-based GP systems.

1. *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 [124]. 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 duplicated or 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 due to exponential growth, we could improve upon our performance results by upwards of 2×.
2. *Multiplex Function Unit Resources.* Function unit primitives experience poor utilization due to the fact that they are implemented with independent IP blocks.

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

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., ALMs and DSPs. Such an "overlay" (Sect. 2.3) 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. (The function units were also primarily responsible for ALM consumption.) 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 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.

3. *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 much higher clock frequencies [84, 114, 117]. 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. Notably, for our current FPGA, we refer to the extensive study given in [117] that showcases how frequencies in the range 400–850 MHz are attainable for various designs. To be conservative, we do not assume that we could achieve such clock frequencies, but we provide them as a potential goal. In any event, we reiterate that we have not fully optimized for timing.
4. *Use a Higher-End FPGA Device.* With a more modern, higher-end FPGA implemented on newer process-node technology (e.g., [52]), we should be able to support at least 1.4× ALM resources, 2× more DSP resources, and 1.5× more embedded memory resources, all in addition to higher clock frequencies [24]. Separately, such newer devices allow for an additional 2× more DSP resources when using half-precision floating-point or the recent bfloat16 precision [24]. Since positive results were reported for the state-of-the-art Operon tool when using single-precision floating point instead of double-precision floating point [19], it seems plausible that further reductions in precision could also provide meaningful results. Therefore, with up 4× more DSP resources, we expect that we can further parallelize our current floating-point computations by up to 4×, implying a speedup of up to 4×.
5. *Use Multiple FPGAs.* The CPU results presented in Sect. 6 rely on a dual-socket server populated with two CPU packages, whereas we currently only utilize a single FPGA for our accelerator. Therefore, out of fairness, we could parallelize our design across two FPGAs (e.g., by instantiating multiple evaluation engines), which would allow for up to a 2× speedup. However, there are two considerations

to note here: (1) we would ideally also utilize two GPU devices for the TensorGP results, and (2) when using newer CPU device technologies, a single CPU may be able to provide comparable results to the server utilized in this paper. Regarding the second point, we emphasize that with the current experimental setup, the CPU and FPGA both leverage 14nm device technologies, suggesting that utilizing two FPGA devices would still be fair. Future systems could leverage additional FPGAs, if appropriate.

6. *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 can be no more than the total time taken for evaluation, then this optimization should allow our accelerator to achieve a speedup of up to 2×.

7.3 Final considerations

Following from the previous sections, we establish some final considerations. Perhaps the most notable challenge of the initial tree architecture is the exponential growth in area when scaling up program depth. For example, regarding the experiments described in Sect. 5, the current architecture only supports program depth values up to 10 for the most basic primitive set chosen, and only depth values up to 8 for the most complex primitive sets chosen, where the utilization for the function units within the respective trees can be fairly low, depending on program size. Although these appear to be the largest depths ever recorded for an FPGA-based GP system [35, 104], it is common for larger depth values to be used [59, 89], but not necessarily. Nevertheless, to improve upon this limitation, two possible architectural advancements have already been suggested in Sect. 7.1: (1) a unified "parallel/sequential" tree, inspired by prior work on tree-based accumulator architectures [124], and (2) a linear architecture that natively handles flattened tree (e.g., prefix/postfix) representations. To us, the linear architecture currently seems to be the most interesting of the two suggestions, since it could potentially be applicable to other forms of GP, e.g., linear GP, stack-based GP, etc. [17, 89, 110]. However, as we have already noted, to have a linear architecture that is fully pipelined would seemingly require duplicates of state memory (e.g., multiple temporary stacks) in order to maximize throughput, which would likely infer significantly more memory resources than a tree representation.

Interestingly, the linear architecture could either be implemented so that it fully unrolls a program through a physical chain of function units, or so that it only utilizes a single function unit with feedback, which could be envisioned as a specialized CPU. With the former, more FPGA resources could be allocated to a single evaluation engine, so that the total number of engines could be less when scaling up, which may ultimately help if additional hardware is to be associated with each engine (e.g., hardware for local search). With the latter, we would ideally implement

as many engines as there are programs, but the actual amount would depend on the complexity of the function set as well as the population size. For both linear architecture variants, it seems that the same amount of state memory duplicates would be needed, but the former would likely also require that inputs be pipelined, which would add an amount of memory that grows proportionally to either (1) the product of program size and the number of variables, if all variables were passed through the pipeline, or (2) the square of program size, if each function unit within the pipeline selected from a single set of variable memories, with a linearly increasing delay for units further in the chain.

Thus, the linear architecture involving only a single function unit with feedback might be preferable to the fully-unrolled variant, but some mixture of both ideas could also potentially work. With the single-unit architecture, there is another possible benefit: support for recursion and arbitrary control flow may be significantly easier to implement. Unfortunately, there is also a challenge that is most relevant to the single-unit linear architecture: with multiple evaluation engines executing in parallel, the management of multiple simultaneous program outputs is necessitated, which can make other aspects of the design more complex, especially if additional computation directly follows from program evaluation (e.g., local search). In the case of the single-unit architecture, if hundreds of evaluation engines would have to be in parallel in order to extract significant performance benefits, the amount of memory (or bandwidth) that would be required to support program outputs might become impractical. Overall, future work should consider all of such trade-offs in more detail.

On a separate note, we refer back to Sect. 7.2, in which we mention the possibility of the average speedup of our accelerator improving by upwards of 192×. Importantly, following from Amdahl's Law [8], the speedup of an overall GP system depends on the workload proportions between program evaluation and the remainder of the GP procedure. When incorporating evolution, if there are fixed workload proportions, continually improving only evaluation would eventually cause evolution to become the bottleneck, which would subsequently bottleneck GPops/s improvements. However, by Gustafson's Law [41], perhaps we could alter the GP procedure such that program evaluation and/or evolution could have a dynamic workload. Notably, if we double-buffer GP runs (Sect. 7.2), and if we continually maintain that evolution takes no longer than evaluation (e.g., through increased parallelism), then evaluation would never be bottlenecked by evolution and all improvements to evaluation can be useful.

In addition to the previous points, we establish that if evolution is also to be implemented within the FPGA, then it is possible that some of the resources that would be needed for an additional evaluation speedup would ultimately have to be allocated elsewhere. Therefore, we note that the upper end of the 64–192× speedup range may be optimistic, but we conclude that some significant speedups should still be possible. These potential speedups would likely also affect performance-per-watt (i.e., NPW) values in a similar manner, but it is not yet clear how the corresponding optimizations would affect power consumption, so we do not provide exact estimates for possible performance-per-watt improvements. Future work should also consider whether or not some meaningful combination of a CPU and FPGA could

be designed such that the FPGA is not responsible for all the complexities of the GP system. For instance, if some complex local search procedure (e.g., nonlinear least squares fitting) is desired, perhaps this could be meaningfully done by the CPU at infrequent intervals of time, so that any runtime penalty would not be severe.

8 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 for 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 the recent state-of-the-art Operon tool executing on the same 2-processor CPU system, our tool was the fastest in several instances, and when considering energy consumption, our tool was 1.4× better than Operon in terms of performance-per-watt. Importantly, we also described six future extensions that could provide at least a 64–192× speedup over our current design, which motivates the continued exploration of FPGA-based GP systems.

Although the processes of evolution were not yet implemented, we note that it is plausible that workload proportions for program evaluation will be similar to what was presented in the original study of Operon (Sect. 3.1.2), since the accelerator exhibits evaluation performance close to Operon [19], and since we can likely double-buffer GP runs in order to ensure that evolution does not bottleneck evaluation (Sect. 7.2). Thus, just as we argued for Operon within Sect. 3.1.2, this could allow our accelerator to exhibit the largest GPops/s values ever reported. Separately, within Sect. 7.3, we established that useful accelerator architectures may be possible for program representations other than trees. For instance, we described some "linear" architectures that may be widely applicable. If similar performance enhancements could be expected for other GP domains, significant benefits may manifest when compared to the pre-existing systems developed for these domains. As an example, consider the recent GP framework known as Tangled Program Graphs (TPG), through which individual programs are typically represented as in linear GP [56]. So far, not many significant optimizations for the training procedure of TPG have been explored, yet remarkable results have still been reported; for instance, TPG has discovered multi-task, high-dimensional reinforcement learning policies with over 1000× fewer operations than Deep-Q Learning when using the same amount of training time [56]. Notably, we posit that the evaluation scheme provided by TPG is just as challenging (if not more challenging) to optimize for CPU/GPU systems, due to the reasons discussed in Sect. 1. Therefore, the benefits of domain-specific

architectures could end up being even more significant for complex systems such as TPG, which further motivates this line of work.

Overall, we affirm that extending the work presented in this paper is a worthwhile pursuit. When taking everything together, FPGAs may ultimately allow for faster and/or less costly GP runs, in which case it may also be possible for better solutions to be found when allowing an FPGA to consume the same amount of runtime/energy as another computing platform. From our initial results, it seems likely that some highly significant performance enhancements can be achieved with modern FPGA devices, and once considerations of power and energy are made, even more benefits may be present. Notably, designs established for FPGA platforms should also be applicable to ASICs (Sect. 2.2), which may be an important next step after the development of a full FPGA-based GP system, since this could lead to even higher performance and energy efficiency, similar to how tensor processing units (TPUs) have extracted additional benefits for neural network contexts [54]. There has been some preliminary work on the design of ASICs for GP (e.g., [74]), but there is still much left to explore.

Author Contributions All authors contributed to the conception, design, or analysis of the study. The manuscript was prepared by Christopher Crary and was reviewed by all authors.

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

Data availibility See reference [28].

Declarations

Conflict of interest The authors have no conflict of interest to declare.

References

1. M. Abadi , P. Barham, J. Chen, et al., TensorFlow: A system for large-scale machine learning, in ed. by K. Keeton, T. Roscoe, Proceedings of the 12th USENIX symposium on Operating Systems Design and Implementation (OSDI 2016) (ACM, New York, NY, USA, 2016), pp. 265–283, https://doi.org/10.5555/3026877.3026899
2. B. Acun, B. Lee , F. Kazhamiaka, et al., Carbon explorer: A holistic framework for designing carbon aware datacenters, in ed. by TM. Aamodt, NE. Jerger, M. Swift, Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2023), vol 2. (ACM, New York, NY, USA, 2023) pp. 118–132, https://doi.org/10.1145/3575693.3575754
3. A. Agrawal, A. Modi, A. Passos, et al., TensorFlow Eager: A multi-stage, Python-embedded DSL for machine learning, in ed. by A.Talwalkar, V. Smith, M. Zaharia, Proceedings of the Second Conference on Machine Learning and Systems (MLSys 2019), pp. 178–189 (2019). https://proceedings.mlsys.org/paper_files/paper/2019/file/b3cd73d353d39e5cf6f6e9ff8d14c87f-Paper.pdf
4. G. Alok, Architecture apocalypse dream architecture for deep learning inference and compute - Versal AI core, in Proceedings of the of the Embedded World 2020 Exhibition and Conference, (2020). https://download.amd.com/docnav/documents/aem/white_papers-EW2020-Deep-Learning-Inference-AICore.pdf

5. Amazon (2024) EC2 F1. https://aws.amazon.com/ec2/instance-types/f1/
6. AMD Versal ACAP DSP engine architecture manual (AM004), (2018). https://docs.amd.com/r/en-US/am004-versal-dsp-engine/DSP58-Architecture
7. AMD, Heterogeneous Accelerated Compute Cluster (HACC) program (2024). https://www.amd.com/en/corporate/university-program/aup-hacc.html
8. G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67, Spring). Thompson Book Company, (Washington, D.C., 1967) pp. 483–485. https://doi.org/10.1145/1465482.1465560
9. A. Michael, P. Greg, K. Ronny, S. Nick, M. Vishal, B. Gonzalo, R. Sridhar, NVIDIA Hopper architecture in-depth (2022). https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/
10. Apple (2022) M1 Ultra chip. https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/
11. D.A. Augusto, H.J. Barbosa, Accelerated parallel genetic programming tree evaluation with OpenCL. J. Parallel Distrib. Comput. **73**(1), 86–100 (2013). https://doi.org/10.1016/j.jpdc.2012.01.012
12. F. Baeta, J. Correia, T. Martins et al., Exploring genetic programming in TensorFlow with TensorGP. SN Comput. Sci. **3**(154), 1–16 (2022). https://doi.org/10.1007/s42979-021-01006-8
13. W. Banzhaf, P. Nordin, R.E. Keller et al., *Genetic programming—an introduction* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998)
14. W. Banzhaf , S. Harding, WB. Langdon, et al., Accelerating genetic programming through graphics processing units, in et. by R. Riolo, T. Soule, B. Worzel, Genetic Programming Theory and Practice VI. (Springer US, Boston, MA, 2009) pp. 1–19, https://doi.org/10.1007/978-0-387-87623-8_15
15. N. Bashir, T. Guo , M. Hajiesmaili, et al., Enabling sustainable clouds: The case for virtualizing the energy system, in ed. by C. Curino, G. Koutrika, R. Netravali, Proceedings of the ACM Symposium on Cloud Computing (SoCC 2021) (ACM, New York, NY, USA, 2021) pp. 350–358, https://doi.org/10.1145/3472883.3487009
16. V. Betz, J. Rose, VPR: a new packing, placement and routing tool for FPGA research, in: W. Luk, Y.K. CP, M. Glesner, Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL 1997) (Springer Berlin, Heidelberg, LNCS, 1997) pp. 213–222, https://doi.org/10.1007/3-540-63465-7_226
17. M. Brameier, W. Banzhaf, W. Banzhaf, Linear Genetic Programming. (Springer, New York, NY, USA, 2007). https://doi.org/10.1007/978-0-387-31030-5
18. B. Burlacu, GECCO'2022 symbolic regression competition: Post-analysis of the Operon framework, in ed. by S. Silva, L. Paquete, Proceedings of the 2023 Genetic and Evolutionary Computation Conference Companion (GECCO 2023) (ACM, New York, NY, USA, 2023) pp. 2412–2419, https://doi.org/10.1145/3583133.3596390
19. B. Burlacu, G. Kronberger, M. Kommenda, Operon C++: An efficient genetic programming framework for symbolic regression, ed. by C. Artemio Coello, Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion (GECCO 2020) (ACM, New York, NY, USA, 2020) pp. 1562–1570, https://doi.org/10.1145/3377929.3398099
20. A. Cano, B. Krawczyk, Evolving rule-based classifiers with genetic programming on GPUs for drifting data streams. Pattern Recogn. **87**, 248–268 (2019). https://doi.org/10.1016/j.patcog.2018.10.024
21. S.M. Cheang, K.S. Leung, K.H. Lee, Genetic parallel programming: design and implementation. Evol. Comput. **14**(2), 129–156 (2006). https://doi.org/10.1162/evco.2006.14.2.129
22. D.M. Chitty, Fast parallel genetic programming: multi-core CPU versus many-core GPU. Soft Comput. **16**(10), 1795–1814 (2012). https://doi.org/10.1007/s00500-012-0862-0
23. D.M. Chitty, Faster GPU-based genetic programming using a two-dimensional stack. Soft Comput. **21**(14), 3859–3878 (2017). https://doi.org/10.1007/s00500-016-2034-0
24. J. Chromczak, M. Wheeler, C. Chiasson, et al., Architectural enhancements in Intel Agilex FPGAs, in ed. by S. Neuendorffer, L. Shannon, Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2020) (ACM, New York, NY, USA, 2020) pp. 140–149, https://doi.org/10.1145/3373087.3375308
25. K. Compton, S. Hauck, Reconfigurable computing: a survey of systems and software. ACM Comput. Surveys (CSUR) **34**(2), 171–210 (2002). https://doi.org/10.1145/508352.508353

26. J. Coole, G. Stitt, Adjustable-cost overlays for runtime compilation, in L. Shannon, D. Andrews, Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2015). IEEE, pp 21–24, (2015). https://doi.org/10.1109/FCCM.2015.49
27. C. Crary, W. Piard, G. Stitt, et al., Using FPGA devices to accelerate tree-based genetic programming: A preliminary exploration with recent technologies, in ed. by G. Pappa, M. Giacobini, Z. Vasicek, Proceedings of the 26th European Conference on Genetic Programming (EuroGP 2023, Part of EvoStar), (Springer, Cham, LNCS, 2023) pp. 182–197, https://doi.org/10.1007/978-3-031-29573-7_12
28. C. Crary, W. Piard, G. Stitt, et al., Code repository (2024). https://github.com/christophercrary/journal-gpem-2023/
29. FM De Rainville, FA. Fortin, MA. Gardner, et al., DEAP: A Python framework for evolutionary algorithms, in Soule T, Moore JH (eds) Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Companion (GECCO 2012), (ACM, New York, NY, USA, 2012) pp. 85–92, https://doi.org/10.1145/2330784.2330799
30. R. Dobai, L. Sekanina, Low-level flexible architecture with hybrid reconfiguration for evolvable hardware. ACM Trans. Reconfig. Technol. Syst. (TRETS) **8**(3), 1–24 (2015). https://doi.org/10.1145/2700414
31. G. Dréan , The Chips Industry: Moore and Rock's Laws, (ISTE Ltd, London, UK, 2019), pp. 125–135. https://doi.org/10.1002/9781119468967.ch6
32. S. Eklund (2003) Time series forecasting using massively parallel genetic programming, in ed. by M. Cosnard, A. Gottlieb, J. Dongarra, Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003). IEEE, https://doi.org/10.1109/IPDPS.2003.1213272
33. Esmaeilzadeh H, Blem E, St. Amant R, et al., Dark silicon and the end of multicore scaling, in ed. by R. Iyer, Q. Yang, A. González, Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA 2011) (ACM, New York, NY, USA, 2011) pp. 365–376. https://doi.org/10.1145/2000064.2000108
34. AI. Funie, M. Salmon, W. Luk, A hybrid genetic-programming swarm-optimisation approach for examining the nature and stability of high frequency trading strategies, in ed. by XW. Chen, G. Qu, P. Angelov, et al., Proceedings of the 13th International Conference on Machine Learning and Applications (ICMLA 2014). IEEE, pp 29–34, (2014). https://doi.org/10.1109/ICMLA.2014.11
35. A.I. Funie, P. Grigoras, P. Burovskiy et al., 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
36. M.B. Gokhale, P.S. Graham, Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays, (Springer, New York, NY, USA, 2006). https://doi.org/10.1007/b136834
37. D. Goldberg, What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163
38. C. Goribar-Jimenez, Y. Maldonado, L. Trujillo, et al., Towards the development of a complete GP system on an FPGA using geometric semantic operators, in ed. by JA. Lozano, C. Coello, J. Ceberio, Proceedings of the 2017 IEEE Congress on Evolutionary Computation (CEC 2017). IEEE, pp 1932–1939, (2017). https://doi.org/10.1109/CEC.2017.7969537
39. V. G. Gudise, G. K. Venayagamoorthy, FPGA placement and routing using particle swarm optimization, in ed. by A. Smailagic, M. Bayoumi, Proceedings of the 2004 IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004). IEEE, pp 307–308 (2004). https://doi.org/10.1109/ISVLSI.2004.1339567
40. P. Gupta, CUDA refresher: The CUDA programming model (2020). https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/
41. J.L. Gustafson, Reevaluating Amdahl's Law. Commun. ACM **31**(5), 532–533 (1988). https://doi.org/10.1145/42411.42415
42. S. L. Harding, W. Banzhaf, Hardware acceleration for CGP: Graphics processing units, in ed. by J. F. Miller, Cartesian Genetic Programming, (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), pp. 231–253, https://doi.org/10.1007/978-3-642-17310-3_8
43. S.L. Harris, D. Harris, *Digital design and computer architecture: ARM edition*, 1st edn. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2015)
44. J.L. Hennessy, D.A. Patterson, *Computer architecture: a quantitative approach*, 6th edn. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017)

45. D. L. N. Hettiarachchi, V. S. P. Davuluru, E. J. Balster, Integer vs. floating-point processing on modern FPGA technology, in ed. by S. Chakrabarti, R. Paul, Proceedings of the 10th Annual Computing and Communication Workshop and Conference (CCWC 2020). IEEE, pp 606–612, (2020). https://doi.org/10.1109/CCWC47524.2020.9031118
46. M. I. Heywood, A. N. Zincir-Heywood, Register based genetic programming on FPGA computing platforms, in ed. by R. Poli, W. Banzhaf, W. B, Langdon, et al., Proceedings of the 3rd European Conference on Genetic Programming (EuroGP 2000), (Springer, Berlin, Heidelberg, Berlin, Heidelberg, LNCS, 2000), pp. 44–59, https://doi.org/10.1007/978-3-540-46239-2_4
47. S. Hooker, The hardware lottery. Commun. ACM **64**(12), 58–65 (2021). https://doi.org/10.1145/3467017
48. P. Horowitz, W. Hill, I. Robinson, *The art of electronics*, 3rd edn. (Cambridge University Press, Cambridge, UK, 2015)
49. Intel (2016) Intel FPGA Programmable Acceleration Card D5005 data sheet. https://cdrdv2-public.intel.com/691516/ds-pac-d5005-683568-691516.pdf
50. Intel (2018) BFLOAT16 - hardware numerics definition. https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf
51. Intel (2021) Floating-point IP cores user guide. https://cdrdv2-public.intel.com/666430/ug_altfp_mfug-683750-666430.pdf
52. Intel (2024) Intel Agilex M-Series FPGA and SoC FPGA product table. https://cdrdv2.intel.com/v1/dl/getContent/721636
53. Intel (2024) The story of the Intel 4004. https://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html
54. N. Jouppi, C. Young, N. Patil et al., Motivation for and evaluation of the first tensor processing unit. IEEE Micro **38**(3), 10–19 (2018). https://doi.org/10.1109/MM.2018.032271057
55. N. Kapre, S. Bayliss, Survey of domain-specific languages for FPGA computing, in ed. by P. Ieene, W. Najjar, J. Anderson, et al., Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL 2016) (EPFL, Lausanne, Switzerland, 2016) pp. 1–12, https://doi.org/10.1109/FPL.2016.7577380
56. S. Kelly, M.I. Heywood, Emergent solutions to high-dimensional multitask reinforcement learning. Evol. Comput. **26**(3), 347–380 (2018). https://doi.org/10.1162/evco_a_00232
57. S. Kelly , R. J. Smith, M. I. Heywood, Emergent policy discovery for visual reinforcement learning through tangled program graphs: a tutorial. in ed. by W. Banzhaf, L. Spector, L. Sheneman, Genetic Programming Theory and Practice XVI (Springer, Cham, 2019), pp. 37–57, https://doi.org/10.1007/978-3-030-04735-1_3
58. M. Kommenda, B. Burlacu, G. Kronberger et al., Parameter identification for symbolic regression using nonlinear least squares. Genet. Program Evolvable Mach. **21**(3), 471–501 (2020). https://doi.org/10.1007/s10710-019-09371-3
59. J.R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, 1st edn. (MIT Press, Cambridge, MA, USA, 1992)
60. J. R. Koza, F. H. Bennett, J. L. Hutchings, et al., Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming, in ed. by J. Cong, S. Kaptanoglu, Proceedings of the 6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 1998), (ACM, New York, NY, USA, 1998) pp. 209–219. https://doi.org/10.1145/275107.275141
61. A. Krishnakumar, U. Ogras, R. Marculescu et al., Domain-specific architectures: research problems and promising approaches. ACM Trans. Embed. Comput. Syst. **22**(2), 1–26 (2023). https://doi.org/10.1145/3563946
62. I. Kuon, J. Rose, Measuring the gap between FPGAs and ASICs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(2), 203–215 (2007). https://doi.org/10.1109/TCAD.2006.884574
63. W. La Cava, T. Helmuth, L. Spector et al., A probabilistic and multi-objective analysis of lexicase selection and ϵ-lexicase selection. Evol. Comput. **27**(3), 377–402 (2019). https://doi.org/10.1162/evco_a_00224
64. W. La Cava, P. Orzechowski, B. Burlacu, et al., Contemporary symbolic regression methods and their relative performance, in ed. by J. Vanschoren, S. Yeung, Proceedings of the 35th Conference in Neural Information Processing Systems (NeurIPS 2021), Track on Datasets and Benchmarks 1, (2021). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/c0c7c76d30bd3dcaefc96f40275bdc0a-Paper-round1.pdf

65. S. Lahti, P. Sjövall, J. Vanne et al., Are we there yet? a study on the state of high-level synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **38**(5), 898–911 (2019). https://doi.org/10.1109/TCAD.2018.2834439
66. S. Lahti, M. Rintala, T.D. Hämäläinen, Leveraging modern C++ in high-level synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **42**(4), 1123–1132 (2023). https://doi.org/10.1109/TCAD.2022.3193646
67. W. B. Langdon, Parallel GPQUICK, in ed. by M. López-Ibáñez, Proceedings of the 2019 Genetic and Evolutionary Computation Conference Companion (GECCO 2019) (ACM, New York, NY, USA, 2019), pp. 63–64, https://doi.org/10.1145/3319619.3326770
68. W. B. Langdon, Incremental evaluation in genetic programming, in ed. by T. Hu, N. Lourenço, E. Medvet, Proceedings of the 24th European Conference on Genetic Programming (EuroGP 2021, Part of EvoStar), (Springer, Cham, LNCS, 2021), pp. 229–246, https://doi.org/10.1007/978-3-030-72812-0_15
69. W. B. Langdon, W. Banzhaf, A SIMD interpreter for genetic programming on GPU graphics cards, in M. O'Neill, L. Vanneschi, S. Gustafson, et al., Proceedings of the 11th European Conference on Genetic Programming (EuroGP 2008, Part of EvoStar), (Springer. Springer, Berlin, Heidelberg, LNCS, 2008), pp. 73–85, https://doi.org/10.1007/978-3-540-78671-9_7
70. W.B. Langdon, W. Banzhaf, Long-term evolution experiment with genetic programming. Artif. Life **28**(2), 173–204 (2022). https://doi.org/10.1162/artl_a_00360
71. K. Leswing (2024) Nvidia's latest AI chip will cost more than $30,000, CEO says. https://www.cnbc.com/2024/03/19/nvidias-blackwell-ai-chip-will-cost-more-than-30000-ceo-says.html
72. P. Li, J. Yang, MA. Islam, et al., Making AI less "thirsty": Uncovering and addressing the secret water footprint of AI models. arXiv preprint arXiv:2304.03271 pp 1–16 (2023). https://doi.org/10.48550/arXiv.2304.03271
73. Linux Kernel development community (2024) Power capping framework. https://www.kernel.org/doc/html/next/power/powercap/powercap.html
74. J. Lu, H. Jia, N. Verma et al., Genetic programming for energy-efficient and energy-scalable approximate feature computation in embedded inference systems. IEEE **67**(2), 222–236 (2018). https://doi.org/10.1109/TC.2017.2738642
75. J. Ma, G. Zuo, K. Loughlin, et al., Debugging in the brave new world of reconfigurable hardware, in ed. by B. Falsafi, M. Ferdman, S. Lu, et al., Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022), (ACM, New York, NY, USA, 2022), pp. 946–962, https://doi.org/10.1145/3503222.3507701
76. Y. Maldonado, R. Salas, J.A. Quevedo et al., GSGP-hardware: instantaneous symbolic regression with an FPGA implementation of geometric semantic genetic programming. Genet. Program Evol. Mach. **25**(2), 18 (2024). https://doi.org/10.1007/s10710-024-09491-5
77. P. Martin, A hardware implementation of a genetic programming system using FPGAs and Handel-C. Genet. Program Evolv. Mach. **2**(4), 317–343 (2001). https://doi.org/10.1023/A:1012942304464
78. E. J. McDonald, Runtime FPGA partial reconfiguration. In: Proceedings of the 2008 IEEE Aerospace Conference (AERO 2008). IEEE, pp. 1357–1363 (2008). https://doi.org/10.1109/AERO.2008.4526368
79. T. Mickle, J. Rennison, Nvidia becomes most valuable public company, topping Microsoft (2024). https://www.nytimes.com/2024/06/18/technology/nvidia-most-valuable-company.html?smid=url-share
80. J.F. Miller, Cartesian genetic programming: its status and future. Genet. Program Evolv. Mach. **21**(1), 129–168 (2020). https://doi.org/10.1007/s10710-019-09360-6
81. D. Myers, R. Mohawesh, V.I. Chellaboina et al., Foundation and large language models: fundamentals, challenges, opportunities, and social impacts. Clust. Comput. **27**(1), 1–26 (2024). https://doi.org/10.1007/s10586-023-04203-7
82. R. Nane, V.M. Sima, C. Pilato et al., A survey and evaluation of FPGA high-level synthesis tools. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016). https://doi.org/10.1109/TCAD.2015.2513673
83. M. Nicolau, A. Agapitos, Choosing function sets with better generalisation performance for symbolic regression models. Genet. Program Evolvable Mach. **22**(1), 73–100 (2020). https://doi.org/10.1007/s10710-020-09391-4
84. E. Nurvitadhi, et al., Can FPGAs beat GPUs in accelerating next-generation deep neural networks?, in ed. by J. Greene, J. H. Anderson, Proceedings of the 2017 ACM/SIGDA International

Symposium on Field-Programmable Gate Arrays (FPGA 2017) (ACM, New York, NY, USA, 2017), pp. 5–14, https://doi.org/10.1145/3020078.3021740
85. Nvidia (2016) NVIDIA-SMI documentation. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf
86. M. O'Neill, L. Vanneschi, S. Gustafson et al., Open issues in genetic programming. Genet. Program Evolv. Mach. **11**(3), 339–363 (2010). https://doi.org/10.1007/s10710-010-9113-2
87. J.D. Owens, D. Luebke, N. Govindaraju et al., A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007). https://doi.org/10.1111/j.1467-8659.2007.01012.x
88. T. Perkis, Stack-based genetic programming, in Proceedings of the First IEEE Conference on Evolutionary Computation (ICEC 1994). IEEE, pp 148–153 (1994). https://doi.org/10.1109/ICEC.1994.350025
89. R. Poli, W. B. Langdon, N. F. McPhee, A Field Guide to Genetic Programming, 1st edn. Lulu Enterprises, UK Ltd, (2008). http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli08_fieldguide.pdf
90. A. Putnam, A.M. Caulfield, E.S. Chung 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
91. A. Quenon, V. Ramos Gomes Da Silva, Towards higher-level synthesis and co-design with Python, in ed. by R. Nigam, A. Sampson, S. Neuendorffer, et al., Proceedings of the 2021 Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE 2021), (ACM, New York, NY, USA, 2021), pp. 1–3, https://capra.cs.cornell.edu/latte21/paper/20.pdf
92. I. Rahkovsky, A. Toney, K.W. Boyack et al., AI research funding portfolios and extreme growth. J. Front. Res. Metrics Anal. **6**, 1–13 (2021). https://doi.org/10.3389/frma.2021.630124
93. R. Fernandez, D. de Bulnes, Y. Maldonado, L. Trujillo, Development of multiobjective high-level synthesis for FPGAs. J. Sci. Program. **2020**(7095048), 1–25 (2020). https://doi.org/10.1155/2020/7095048
94. D. Robilliard, V. Marion-Poty, C. Fonlupt, Genetic programming on graphics processing units. Genet. Program Evolv. Mach. **10**(4), 447–471 (2009). https://doi.org/10.1007/s10710-009-9092-3
95. M. Roser, H. Ritchie, E. Mathieu, What is Moore's Law? Our World in Data, (2023) . https://ourworldindata.org/moores-law
96. P.E. Ross, 5 Commandments [technology laws and rules of thumb]. IEEE Spectr. **40**(12), 30–35 (2003). https://doi.org/10.1109/MSPEC.2003.1249976
97. J. Ruiz-Rosero, G. Ramirez-Gonzalez, R. Khanna, Field programmable gate array applications-a scientometric review. Computation **7**(63), 1–111 (2019). https://doi.org/10.3390/computation7040063
98. R. Sabherwal, V. Grover, The societal impacts of generative artificial intelligence: A balanced perspective. J. Assoc. Inf. Syst. 25(1):13–22 (2024). https://doi.org/10.17705/1jais.00860
99. R. Saleh, S. Wilton, S. Mirabbasi et al., System-on-chip: Reuse and integration. Proc. IEEE **94**(6), 1050–1069 (2006). https://doi.org/10.1109/JPROC.2006.873611
100. R. Salvador, A. Otero, J. Mora et al., Self-reconfigurable evolvable hardware system for adaptive image processing. IEEE Trans. Comput. **62**(8), 1481–1493 (2013). https://doi.org/10.1109/TC.2013.78
101. V. Sathia, V. Ganesh, S. R.T. Nanditale, Accelerating genetic programming using GPUs (2021). arXiv preprint arXiv:2110.11226 pp. 1–10. https://doi.org/10.48550/arXiv.2110.11226
102. L. Sekanina, Z. Vasicek, CGP acceleration using field-programmable gate arrays, in ed. by J. F. Miller, Cartesian Genetic Programming, (Springer, 2011), pp. 217–230, https://doi.org/10.1007/978-3-642-17310-3_7
103. Semiconductor Research Corporation (2021) The decadal plan for semiconductors. https://www.src.org/about/decadal-plan/decadal-plan-full-report.pdf
104. R. P. S. Sidhu, A. Mei, V. K. Prasanna, Genetic programming using self-reconfigurable FPGAs, in ed. by P. Lysaght, J. Irvine, R. Hartenstein, Proceedings of the 9th International Workshop on Field Programmable Logic and Applications (FPL 1999), (Springer, Berlin, Heidelberg, LNCS, 1999), pp. 301–312, https://doi.org/10.1007/978-3-540-48302-1_31
105. A. Singleton (1994) Genetic programming with C++. Byte Magazine (February 1994):171–176
106. S. Skalicky, J. Monson, A. Schmidt, et al., Hot & spicy: Improving productivity with Python and HLS for FPGAs, in ed. by G. Schelle, S. Wilton, Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2018). IEEE, pp 85–92 (2018). https://doi.org/10.1109/FCCM.2018.00022

107. J.E. Smith, Characterizing computer performance with a single number. Commun. ACM **31**(10), 1202–1206 (1988). https://doi.org/10.1145/63039.63043
108. H.K.H. So, C. Liu, FPGA overlays, in *FPGAs for software programmers*. ed. by D. Koch, D. Ziener (Springer International Publishing, Cham, 2016), pp.285–305
109. L. Spector, A. Robinson, Genetic programming and autoconstructive evolution with the Push programming language. Genet. Program Evolvable Mach. **3**(1), 7–40 (2002). https://doi.org/10.1023/A:1014538503543
110. L. Spector, J. Klein, M. Keijzer, The Push3 execution stack and the evolution of control, in ed. by H. G. Beyer, U. M. O'Reilly, Proceedings of the 7th Annual Genetic and Evolutionary Computation Conference Companion (GECCO 2005), (ACM, New York, NY, USA, 2005), pp. 1689–1696, https://doi.org/10.1145/1068009.1068292
111. K. Staats, E. Pantridge, M. Cavaglia, et al., TensorFlow enabled genetic programming, in G. Ochoa, Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion (GECCO 2017), (ACM, New York, NY, USA, 2017) pp. 1872–1879. https://doi.org/10.1145/3067695.3084216
112. G. Stitt, Are field-programmable gate arrays ready for the mainstream? IEEE Micro **31**(6), 58–63 (2011). https://doi.org/10.1109/MM.2011.99
113. G. Stitt, J. Coole, Intermediate fabrics: virtual architectures for near-instant FPGA compilation. IEEE Embed. Syst. Lett. **3**(3), 81–84 (2011). https://doi.org/10.1109/LES.2011.2167713
114. G. Stitt, A. Gupta, M. N. Emas, et al., Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems, in ed. by J. H. Anderson, K. Bazargan, Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2018), (ACM, New York, NY, USA, 2018), pp. 173–182, https://doi.org/10.1145/3174243.3174262
115. G. Stitt, (2024) VHDL and SystemVerilog tutorials. https://stitt-hub.com/vhdl-and-systemverilog-tutorials/
116. E. Strubell, A. Ganesh, A. McCallum, Energy and policy considerations for modern deep learning research, in ed. by F. Rossi, V. Conitzer, F. Sha, Proceedings of the 34th AAAI Conference on Artificial Intelligence, vol 34, (AAAI Press, Palo Alto, CA, USA, 2020), pp. 13693–13696, https://doi.org/10.1609/aaai.v34i09.7123
117. T. Tan, E. Nurvitadhi, D. Shih, et al., Evaluating the highly-pipelined Intel Stratix 10 FPGA architecture using open-source benchmarks, in ed. by K. Sano, Y. Yamaguchi, Y. Osana, 2018 International Conference on Field-Programmable Technology (FPT 2018). IEEE, pp 206–213, (2018). https://doi.org/10.1109/FPT.2018.00038
118. R. Tessier, K. Pocek, A. DeHon, Reconfigurable computing architectures. Proc. IEEE **103**(3), 332–354 (2015). https://doi.org/10.1109/JPROC.2014.2386883
119. S. M. Trimberger, Field-programmable Gate Array Technology, 1st edn. (Springer, New York, NY, USA, 2012) https://doi.org/10.1007/978-1-4615-2742-8
120. L. Trujillo, J.M. Muñoz Contreras, D.E. Hernandez et al., GSGP-CUDA: a CUDA framework for geometric semantic genetic programming. SoftwareX **18**(101085), 1–7 (2022). https://doi.org/10.1016/j.softx.2022.101085
121. K. Vipin, S.A. Fahmy, FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications. ACM Comput. Surv. **51**(4), 1–39 (2018). https://doi.org/10.1145/3193827
122. D.R. White, J. McDermott, M. Castelli et al., Better GP benchmarks: community survey results and proposals. Genet. Program Evolv. Mach. **14**, 3–29 (2012). https://doi.org/10.1007/s10710-012-9177-2
123. M. Wijtvliet, L. Waeijen, H. Corporaal, Coarse grained reconfigurable architectures in the past 25 years: Overview and classification, in ed. by W. Najjar, A. Gerstlauer, Proceedings of the 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS XVI). IEEE, pp 235–244, (2016). https://doi.org/10.1109/samos.2016.7818353
124. D. Wilson, G. Stitt, The unified accumulator architecture: a configurable, portable, and extensible floating-point accumulator. ACM Trans. Reconfig. Technol. Syst. **9**(3), 1–23 (2016). https://doi.org/10.1145/2809432
125. D. Wilson, G. Stitt, Seiba: an FPGA overlay-based approach to rapid application development, in ed. by R. Cumplido, M. Platzner, D. Andrews, Proceedings of the 2019 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2019). IEEE, pp 214–221, (2019). https://doi.org/10.1109/ReConFig48160.2019.8994693

126. G. Wilson, W. Banzhaf, Linear genetic programming GPGPU on Microsoft's Xbox 360. In: Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC 2008). IEEE, pp 378–385, (2008). https://doi.org/10.1109/CEC.2008.4630825, can't find editors
127. W. Wolf, The future of multiprocessor systems-on-chips, in, ed. by S. Malik, L. Fix, A.B Kahng, Proceedings of the 41st Annual Design Automation Conference (DAC 2004) (ACM, New York, NY, USA, 2004) pp. 681–685, https://doi.org/10.1145/996566.996753
128. Y. L. Wu, D. Chang, On the NP-completeness of regular 2D FPGA routing architectures and a novel solution, in ed. by P. Storms, Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design (CAD 1994). IEEE, pp 362–366, (1994) https://doi.org/10.1109/ICCAD.1994.629819
129. X. Yao, Following the path of evolvable hardware. Commun. ACM **42**(4), 46–49 (1999). https://doi.org/10.1145/299157.299169
130. X. Yao, T. Higuchi, Promises and challenges of evolvable hardware. IEEE Trans. Syst., Man, Cybernet., Part C (Appl. Rev.) **29**(1), 87–97 (1999). https://doi.org/10.1109/5326.740672
131. R. Zhang, A. Lensen, Y. Sun, Speeding up genetic programming based symbolic regression using GPUs, in ed. by S. Khanna, J. Cao, Q. Bai, et al., Proceedings of the 19th Pacific Rim International Conference on Artificial Intelligence (PRICAI 2022) (Springer, Cham, 2022), pp. 519–533. https://doi.org/10.1007/978-3-031-20862-1_38
132. Z. Zhou, Strategies to speedup Tangled Program Graphs (TPG) framework for genetic programming. Bachelor of Computer Science Thesis, Dalhousie University (2020). https://doi.org/10.13140/RG.2.2.17908.37768

Authors and Affiliations

Christopher Crary[1] · Wesley Piard[1] · Greg Stitt[1] · Benjamin Hicks[1] · Caleb Bean[1] · Bogdan Burlacu[2] · Wolfgang Banzhaf[3]

✉ Christopher Crary
ccrary@ufl.edu

Wesley Piard
wespiard@ufl.edu

Greg Stitt
gstitt@ufl.edu

Benjamin Hicks
benjamin.hicks@ufl.edu

Caleb Bean
caleb.bean@ufl.edu

Bogdan Burlacu
bogdan.burlacu@fh-ooe.at

Wolfgang Banzhaf
banzhafw@msu.edu

[1] Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA

[2] Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Upper Austria, Austria

[3] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA