Chapter 14
It's Time to Revisit the Use of FPGAs for Genetic Programming

Christopher Crary, Greg Stitt, Bogdan Burlacu, and Wolfgang Banzhaf

Abstract In the past, field-programmable gate arrays (FPGAs) have had some notable successes when employed for Boolean and fixed-point genetic programming (GP) systems, but the more common floating-point representations were largely off limits, due to a general lack of efficient device support. However, recent work suggests that for both the training and inference phases of floating-point-based GP, contemporary FPGA technologies may enable significant performance and energy improvements—potentially multiple orders of magnitude—when compared to general-purpose CPU/GPU devices. In this chapter, we highlight the potential advantages and challenges of using FPGAs for GP systems, and we showcase how novel algorithmic considerations likely need to be made in order to extract the most benefits from specialized hardware. Primarily, we consider tree-based GP, although we include suggestions for other program representations. Overall, we conclude that the GP community should earnestly revisit the use of FPGA devices, especially the tailoring of state-of-the-art algorithms to FPGAs, since valuable enhancements may be realized. Most notably, 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 platform.

C. Crary (✉) · G. Stitt
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
e-mail: ccrary@ufl.edu

G. Stitt
e-mail: gstitt@ufl.edu

B. Burlacu
Heuristic and Evolutionary Algorithms Laboratory, University of Applied Sciences Upper Austria, Hagenberg, Upper Austria, Austria
e-mail: bogdan.burlacu@fh-ooe.at

W. Banzhaf
Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA
e-mail: banzhafw@msu.edu

S. M. Winkler et al. (eds.), *Genetic Programming Theory and Practice XXI*, Genetic and Evolutionary Computation, https://doi.org/10.1007/978-981-96-0077-9_14

14.1 Introduction

Recent trends in machine learning highlight the need for energy-efficient computation, during both training and inference [25, 69]. For example, despite widespread success, neural networks often consume prohibitive amounts of power and energy for many use cases [8, 69], in addition to posing considerable scaling challenges for well-established use cases, such as data centers [1, 8, 69]. Such limitations can motivate other learning systems, such as genetic programming (GP) [7, 41, 55], where 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 [39, 44, 55]. However, training often remains complex with current GP techniques, which motivates improvements to training efficiency [11, 17, 55].

There are various ways to improve the training efficiency of GP, and they generally 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 as before.

In regard to its computational model, GP is in a remarkable yet challenging position. Although the algorithms of GP are often embarrassingly parallel [55], which can open the door to extremely high-performance and energy-efficient computation, the most widely accessible computing platforms—CPUs and GPUs—are not quite built for the multiple-program, multiple-data model of GP. 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 [30]. 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 [13, 17, 58].

Ideally, we would have a computing platform that more perfectly aligns with the computational model of GP, in order to improve performance and energy efficiency. In general, achieving such an alignment is the motivation for *domain-specific architectures*, for which either an application-specific integrated circuit (ASIC) or a reconfigurable computing platform such as a field-programmable gate array (FPGA) can be leveraged [30]. Although designing specialized hardware via an ASIC allows for the most flexibility in achieving desired performance and power/energy characteristics [30], FPGAs allow for reconfigurability and lower development costs, which we believe lowers the barrier to entry for any initial research efforts within the GP community. Ultimately, successful FPGA designs can be repurposed for an ASIC, with the goal of obtaining additional performance and energy benefits.

In this chapter, we establish various potential benefits and challenges of implementing modern GP with modern FPGAs. We loosely base our work on the state-of-the-art Operon tool [10, 11, 44], which is floating-point-based and tree-based, but we also include considerations for other GP domains. In years past, the application of FPGAs to floating-point-based GP was largely off limits, due to older technologies that did not efficiently support floating-point [31]. Now, with the advent of floating-point multiply-adders, along with the potential for numerous other resources and higher clock frequencies, improvements of multiple orders of magnitude may be accessible with FPGAs [16, 17]. Nevertheless, efficient mappings of GP to specialized hardware remain challenging, and we believe that various algorithmic considerations should be made in order to maximize amenability, by which we generally mean that unnecessary complexity should be stripped out.

For example, it is suggested in [17] that a key optimization needed for achieving significant computational efficiency with an FPGA relies on the idea of executing GP function primitives with a minimal amount of shared hardware resources. Unfortunately, designing such a resource-sharing mechanism for standard implementations of floating-point operators is challenging, due to high-performance requirements of the relevant algorithms often leading to overly complex or obfuscated realizations. However, this begs an important question: *does GP need standard operators?* For floating-point domains, could more efficient implementations that more roughly approximate the corresponding continuous operators suffice for evolution? In [16], we establish that the answer is *yes*, such operators can suffice.

As another example, we plan to integrate local search techniques [55], but we foresee that more complex algorithms such as Levenberg-Marquardt may not be appropriate for a hardware accelerator, due to a memory complexity that could likely create a performance bottleneck [40]. A more appropriate choice for local search may be a form of batched gradient descent, but additional studies are likely needed to determine effects on solution quality, especially in conjunction with other algorithmic simplifications. Also, similar considerations should be made for multi-objective evolutionary algorithms, such as NSGA-II [55].

Beyond efforts to accelerate the training procedures of GP with FPGAs, which is the main focus of this chapter, we note that the use of FPGAs for deploying final models is also of considerable interest. For instance, although a physical tree-based architecture may not be the best solution for accelerating the evaluation of arbitrary tree-based programs (Sect. 14.4), implementing a final tree-based model with such an architecture could minimize the latency and runtime of individual predictions, which is likely desirable. Ultimately, the reconfigurable nature of FPGAs readily allows for separate specialized architectures during inference, which could enable GP to provide even more competitive machine learning solutions [18].

The remainder of this chapter is as follows. In Sect. 14.2, we list some related work. In Sect. 14.3, we provide some relevant background on modern computing and FPGA devices. In Sect. 14.4, we detail some instances of how FPGA-based hardware architectures could be devised for GP. In Sect. 14.5, we establish various challenges and motivate future work. In Sect. 14.6, we conclude our study.

14.2 Related Work

Since prior work on FPGA-based GP is not frequently referenced, we provide a brief listing of all such works of which we are aware. Similar to how comparing different GP representations (e.g., tree-based GP and linear GP) can be tricky, comparing specialized hardware architectures—especially across GP domains—is problematic. In addition, with most of the following works employing significantly outdated FPGA devices, the advantages/disadvantages of each system are not clear within the context of modern technologies. Ultimately, due to these issues as well as space constraints, we largely avoid establishing either comparisons or results.

For traditional tree-based GP [41, 55], we refer the reader to [17, 23, 42, 62]. For Linear GP [9], see [12, 21, 32, 50]. For Cartesian GP [52], see [19, 61, 72, 73]. For geometric semantic GP [53], see [27, 49]. For grammar-guided GP [51], stack-based GP [65], tangled program graphs [39], and other well-known GP representations, we are unaware of implementations leveraging FPGA devices. Separately, we note that FPGAs have also been used for the application area of *evolvable hardware* [75, 76], although this has been primarily for evolving circuit-based solutions, rather than for improving the performance/energy characteristics of the GP procedure itself.

Of the aforementioned FPGA-based systems, it appears that the systems given in [12, 17, 23, 24] utilized floating-point, where [17] was our work on an initial tree-based accelerator, and where [23, 24] independently proposed an architecture conceptually similar to ours, with ours providing several important contributions, as described in [17]. The architecture presented in [12] was for linear GP, and it is a significant departure from the other floating-point-based architectures listed.

14.3 Background

In this section, we provide some relevant background on modern (digital) computing, which should motivate the exploration of FPGA-based architectures for GP.

14.3.1 A Brief Overview of Modern Computing

For about twenty years now, the practical relevance of *Moore's Law* has been waning, and *Dennard scaling* no longer applies [30]. 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.

[1] Recent work allows us to conclude that Moore's Law is still alive [59].

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 [30]. 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* [6, 30]. With consistent power budgets due to physical constraints, the number of general-purpose cores in a single chip approached a practical upper limit [22, 30]. 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 [30, 54, 56, 67, 70].

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) [30, 54, 56, 67, 70]. Although DSAs can sometimes serve as complete solutions, the latest trend is to integrate both general-purpose and domain-specific *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 [30]. Such a composite system is often 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 high-bandwidth memory. In terms of number of transistors, this chip represents roughly a *50 million times increase* from the Intel 4004 chip—the first commercially produced microprocessor—which consisted of 2,300 transistors.

14.3.2 Domain-Specific Architectures

Many application domains can benefit from the use of a domain-specific architecture (DSA) [30, 43]. 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 [30, 66].

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 desired performance and power/energy characteristics, as well as low unit costs, but this route usually requires years of NRE time and millions of dollars in NRE costs [30]. Unfortunately, according to *Rock's Law* [20], NRE costs often increase significantly with newer device technologies, which continually makes ASIC engineering 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 [30, 71]. 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] Oftentimes, RC systems gain appeal by trading off higher unit costs for both (1) lower NRE times/costs and (2) comparable performance/energy benefits. 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, energy-efficient solutions, and the ability to support different hardware designs over multiple reconfigurations, RC platforms are used for various research [54, 56, 60, 67, 70]. The most popular type of RC system is a *field-programmable gate array* (*FPGA*), which we detail in the next subsection.

14.3.3 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 [29]. 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 14.1. By way of LUTs, FPGAs support combinational logic. To additionally support sequential logic, FPGAs also increasingly leverage *flip-flop* memory components [29]. Ultimately, combinations of LUT and flip-flop components can allow for highly flexible circuit configurations, but implementing a 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 resources, 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 engines" [2]), so that common computing tasks can more readily be implemented [2, 34, 54, 66, 70]. In this respect, modern FPGAs are, themselves, SoC devices. Overall, by integrating many thousands or millions of components via a reconfigurable interconnect, FPGAs can

[2] However, the manufacturing of older technologies generally becomes cheaper over time [30].

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

[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" [4, 14, 26, 35], although double precision is still relatively complex.

Inputs			Outputs	
Cin	A	B	Sum	Cout
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

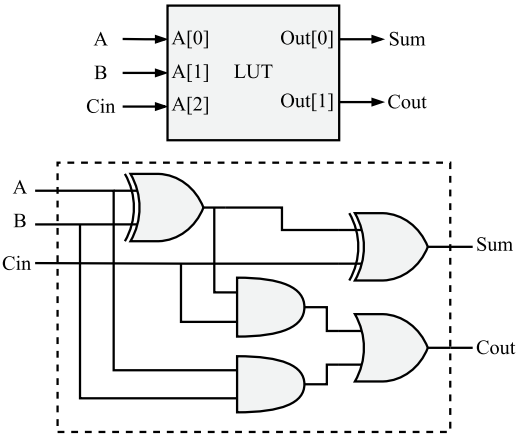

Fig. 14.1 A LUT-based implementation of a full adder, with carry-in and carry-out. Importantly, the depicted LUT memory could implement not only this circuit, but *all* 3-input, 2-output digital circuits. Also, note that the schematic in the bottom-right is just for illustration; with a LUT-based implementation, no logic gates are used. For a more thorough example, see [29, Example 5.5]

implement massively parallel designs, many of which provide significant performance/energy benefits when compared to CPU/GPU systems [30, 60].

Importantly, the aforementioned characteristics suggest that modern FPGAs may be an attractive implementation option for genetic programming (GP), where algorithms are often embarrassingly parallel and often exhibit both data-level and function-level parallelism. In the next section, we illustrate this concept further.

14.4 Applying FPGAs to Genetic Programming

In this section, we identify how modern FPGAs may provide valuable practical enhancements to genetic programming (GP). Most notably, we show how the performance (i.e., throughput) of *floating-point-based* GP may be improved by multiple orders of magnitude when compared to state-of-the-art CPU/GPU systems. We make our case for a form of tree-based GP loosely inspired by Operon [11]—a state-of-the-art tool in terms of both technology and solution quality [10, 44]—but we also provide considerations for other GP representations. We establish performance benefits within the context of program evaluation, since this is generally the performance bottleneck of GP systems [11, 55], and we then consider how evolution could align with such enhancements. Ultimately, designing high-performance hardware for evolution is a major next step for future work. We mainly consider performance rather than power/energy, since the former is more well studied [16, 17], but we note that it has been widely demonstrated that FPGAs provide significant power/energy improvements for various applications [2, 30, 54, 56, 67, 70].

Therefore, if we allow for the possibility of any of such benefits, we can estimate that improvements in *performance-per-watt* measures may be even more significant. Overall, our preliminary results motivate additional studies into FPGA-based GP.

14.4.1 Evaluation

First, we examine the architecture that we previously presented in [17]. As depicted in Fig. 14.2, the architecture 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. 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. 14.2. Although pipelining the tree precludes the existence of arbitrary control structures, such structures are usually unnecessary [10, 44], and pipelining enables data-level and/or function-level parallelism. Lastly, to further increase throughput, 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

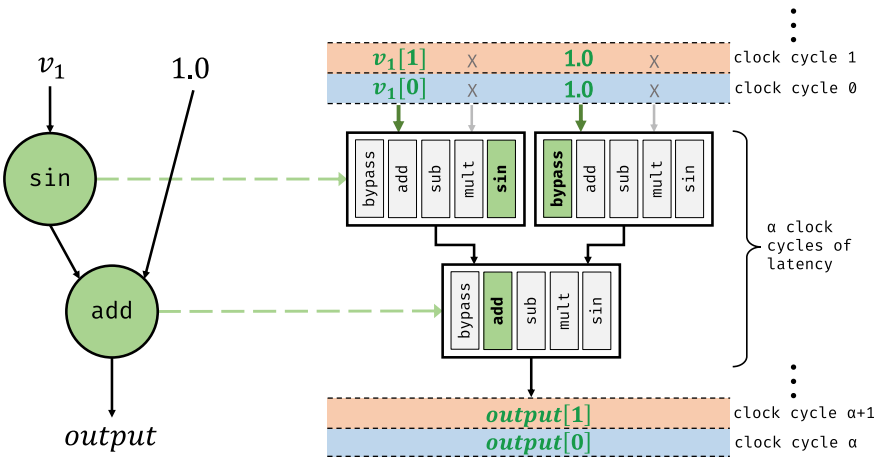

Fig. 14.2 A portrayal of our initial tree-based GP accelerator presented in [17], which can parallelize the evaluation of different data points and different solutions *every clock cycle* via a reconfigurable tree pipeline. Each node of the pipeline can perform any function within the GP primitive set, as well as a bypass, which allows for arbitrary program shapes

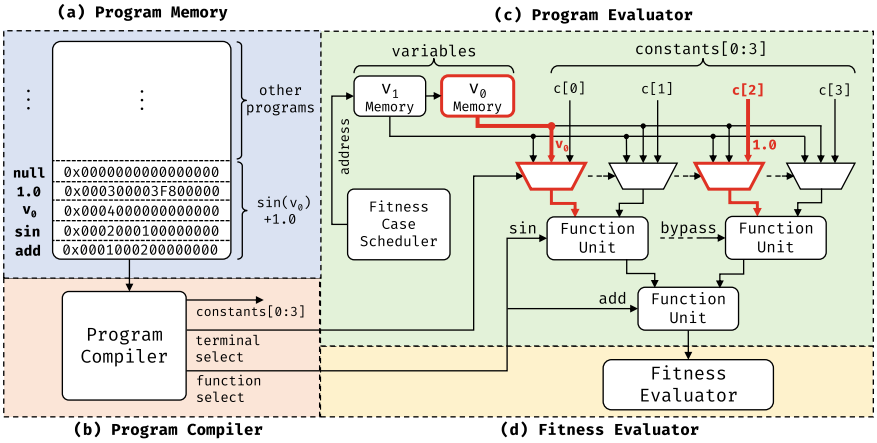

Fig. 14.3 High-level overview of the architecture presented in [17]. Programs (e.g., $\sin(v_0) + 1.0$) are stored in (**a**) *program memory* and dynamically compiled by the (**b**) *program compiler* into machine codes for the (**c**) *program evaluator*. The program evaluator uses a reconfigurable function tree pipeline to execute a compiled expression for a set of fitness cases, resulting in a set of estimated outputs to which the (**d**) *fitness evaluator* compares a set of desired outputs

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 an overall flow of this initial architecture, see Fig. 14.3.

In [17], we compared the aforementioned architecture to several recent CPU/GPU systems using the performance measure *node evaluations per second*, which is effectively the conventional *GP operations per second (GPops/s)*, but without a notion of evolution [13]. Using three different floating-point primitive sets of varying complexity, we showcased that our architecture implemented on a mid-range 14nm FPGA achieved an average speedup of 43× when compared to a recent GPU solution, TensorGP, implemented on 8nm process-node technology, and an average speedup of 4,902× when compared to a popular 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 state-of-the-art Operon tool executing on the same *2-processor* CPU system, our system was the fastest in several instances, and we detailed five potential extensions that could provide a 32–144× speedup over the initial design. Below, we summarize these extensions, and we include two additional extensions that augment the possible speedup range to be 128–576×. But first, in order to effectively explain the extensions, we highlight the three main challenges of our initial architecture:

1. *Exponential Growth.* For an m-ary tree with $m > 1$, where m is the maximum function arity of the primitive set, the amount of function units needed to implement the tree grows exponentially with increasing tree depth.

2. *Function Unit Complexity.* For the tree architecture to be able to support arbitrary programs, every function unit must support *all* function primitives.
3. *Low Resource Utilization.* 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. In addition, the utilization of low-level device primitives (e.g., DSPs) can be significantly less.

Now, we summarize the potential extensions which, if all were achieved simultaneously, could allow for a 128–576× speedup over the initial design:

1. *Use Compacted Trees.* To be able to more effectively leverage device resources as well as support larger program depths/sizes, we could employ "compacted tree" architectures that allow for the use of all resources that are currently unused due to exponential growth. See below for details. (Up to 2× speedup.)
2. *Multiplex Function Unit Resources.* Function unit primitives experience poor utilization due to the fact that they are implemented with independent IP blocks. This issue could be improved upon by implementing a function unit via a single IP block that multiplexes a minimal amount of some device resource(s), e.g., DSPs. Such an "overlay" could free up a significant amount of resources, allowing for further parallelization of program evaluation. (Between 2–6× speedup.)
3. *Design for Higher Clock Frequencies.* For our accelerator, performance (i.e., throughput) is directly proportional to clock frequency. With modern FPGAs, it is not uncommon for designs to achieve clock frequencies in the range 400–850 MHz after optimizing for timing [54, 67, 70]. We estimate that we can achieve up to a 2–3× higher average clock frequency once we optimize for timing and potentially move to a newer device [17]. (Between 2–3× speedup.)
4. *Use a Higher-End FPGA Device.* With a more modern, higher-end FPGA implemented on a newer process-node technology (e.g., [34]), we should be able to support at least 1.4× more ALM resources, 2× more DSP resources, and 1.5× more embedded memory resources, all in addition to higher clock frequencies [14]. Separately, such newer devices allow for an additional 2× more DSP resources when using half-precision floating-point or the recent bfloat16 precision [14], which may be useful for GP [11]. (Up to 4× speedup.)
5. *Use Multiple FPGAs.* The CPU results presented in [17] 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. (Up to 2× speedup.)
6. *Double-buffer GP Runs.* When our accelerator enters the context of a full GP system, including evolution, we aim to execute two GP runs simultaneously, by evolving one population while evaluating another. (Up to 2× speedup.)

In regard to "compacted trees," we note that we previously suggested two alternative architectures in [17]. Of these two architectures, we are currently most interested in the specialized "linear" processor that would directly execute prefix/postfix representations. Essentially, when compared to our initial architecture, we could potentially achieve better throughput and support larger program sizes/depths by

instantiating many specialized single-unit processing cores in parallel. The execution model of this architecture would be loosely similar to a standard CPU system, but we expect that the number of cores within our architecture could scale better, and we expect that our cores could have higher throughput within the context of GP, since we could execute entire function primitives every clock cycle [17].

In addition to higher throughput and larger program sizes/depths, some other potential benefits of a "specialized CPU" architecture are (1) input bandwidth requirements could be reduced, since multiple cores could share the same inputs and since the evaluation of a single program would take longer—whereas the evaluation of an entire population may take less time when accounting for multiple cores—which may allow datasets to more easily be stored off-chip; (2) the need for program compilation could be removed, since native prefix/postfix representations could be directly interpreted, which would eliminate a possible performance/area bottleneck; and (3) arbitrary control flow could potentially be supported, since every core would effectively only execute one program node at a time. On the other hand, some potential challenges introduced by the specialized CPU architecture are (1) pipelining a sequential execution model could require state memory (e.g., a temporary stack) to be duplicated multiple times, which could limit throughput and program sizes/depths if device resources are exhausted too quickly; and (2) any additional logic that is to be included for evaluation (e.g., fitness calculations, local search, etc.) may have to be included within each core in order to maximize performance, which may create a hardware area bottleneck depending on the complexity of the relevant logic. In any event, further investigations of this architecture are warranted.

For GP representations other than trees, it seems that specialized CPUs may also be attractive. For example, when compared to tree-based GP, it seems plausible that we could similarly pipeline the execution model of linear GP by duplicating register files. Separately, strategies for stack-based GP would likely be similar to tree-based GP, but the existence of stacks for different data types may be challenging for memory consumption. Ultimately, other architectures may be more appropriate for a given GP domain, e.g., spatially parallel architectures may be more appropriate for Cartesian GP [72], but it is still worth considering the use of sequential execution models. Notably, the major semiconductor company AMD has recently embraced similar execution models for neural networks and other domains by embedding specialized CPU cores—"AI engines"—within some of their Versal FPGAs [2].

Besides the architectural extensions already mentioned, we note that other performance enhancements can potentially be extracted from FPGAs, depending on the remaining aspects of the relevant GP system. For example, suppose that local search is to be supported and that a weight and bias term are to be allocated to each program node, similar to Operon [11]. In this context, there effectively can be up to $5\times$ as many nodes in each program once accounting for the extra multiplication and addition operations, and we could potentially compute these extra multiply-adds in parallel to standard node computations by allocating just one additional DSP resource to each computation core, which would allow for up to another $5\times$ speedup. In addition, when considering more complex evaluation routines, e.g., automatic differentiation and batched gradient descent, other "downstream" computations such

as weight/bias updates could potentially be parallelized as well, which may provide even more flexibility in achieving considerable performance enhancements.

Overall, within the context of tree-based GP, we note that modern FPGAs provide the potential for performance improvements of multiple orders of magnitude when compared to state-of-the-art tree-based CPU/GPU systems, like Operon [11]. In addition, based on various previous studies involving FPGA devices, such performance benefits may enable similarly significant energy improvements [2, 30, 54, 56, 67, 70]. Although it is not yet clear what modern FPGAs may achieve in other GP domains, we note that similar enhancements seem plausible, given that the program interpreter of Operon could be used as a baseline for a generic multiple-program, multiple-data system [11], and given that the hardware resources of an FPGA can be configured in most any manner. At the very least, our current results motivate additional studies into how FPGAs could be applied to various forms of GP.

14.4.2 Evolution

In general, we expect the performance of an overall GP system to largely be governed by how efficient evaluation is implemented, since the runtime of evaluation generally grows with the size of the training dataset, whereas the runtime of evolution often does not [11, 55]. Although Amdahl's Law suggests that evolution could eventually become a bottleneck [30]—since continually optimizing a fixed workload would have diminishing returns—we estimate that the hardware complexity of evolution will often be considerably less than that of evaluation, especially when considering the potential for more complex evaluation routines, like those for gradient-based local search [40]. In addition, if we intend to double-buffer two independent GP runs in order to maximize throughput across GP runs (Sect. 14.4.1), then we could reasonably allow the runtime of evolution to be equal to that of evaluation—since for each generation we could prevent dead cycles for evaluation just by having the evolutionary routines of one GP run be completed by the end of the evaluation routines for the other GP run—which could enable reduced hardware complexity for evolution. Lastly, following Gustafson's Law [28], we may be able to meaningfully augment the GP procedure such that the workload of evaluation is significantly increased without increasing the workload of evolution by the same amount, which may further ease hardware requirements for evolution.

For tree-based GP, we briefly consider possible strategies for tournament selection, subtree crossover, and one-point mutation [55], which are normally employed by the state-of-the-art Operon tool [10, 11, 44].[5] For tournament selection, hardware should be relatively simple, since we can likely just infer a tree of pipelined comparators in order to find a program that has the minimum/maximum fitness value within some tournament. With a pipelined implementation, tournaments can potentially be decided every clock cycle, which may ultimately mean that only one

[5] Other operators are often included as well, which we leave for future work (Sect. 14.5.2).

"selection engine" is needed to align with the performance of evaluation. For subtree crossover and one-point mutation, pipelined implementations are likely more challenging, since tree-based programs often need to be parsed in order to determine any viable crossover/mutation point(s) and in order to copy program data. Thus, multiple "variation engines" may be needed in parallel in order to align with potential evaluation speedups, but each engine should be relatively simple, containing mostly just a state machine and some shift registers. For all of these routines, we can likely allocate just a few small buffers for storing random numbers, and then have some independent circuit(s) continually keep the buffers full. For pseudo-random number generation, the *taus88* algorithm seems like a viable candidate, as it is robust and efficient for hardware [48]. Future work can consider other options, if necessary.

For GP representations other than trees, hardware implementations of evolution may be even less complex. For example, some representations readily allow for uniformly random crossover and mutation points, which can reduce or eliminate the need to parse programs during variation, and which in turn may enable the ability to use significantly less hardware when aligning with potential evaluation speedups. This fact motivates additional considerations into how tree-based algorithms could potentially be altered in order to maximize amenability to specialized hardware, and it also motivates continued explorations into which GP representation may be the most appropriate for a given application. Of course, many factors must be considered for the latter. For instance, if gradient-based local search is desired, techniques like automatic differentiation may be fairly straightforward to implement for tree-based expressions, since trees directly encode a "forward" and "reverse" pass, whereas implementations for other representations might be considerably more complex, since a graph structure may first need to be constructed.

Ultimately, it seems plausible that hardware for evolution could align well with the potential performance enhancements laid out for evaluation (Sect. 14.4.1). Therefore, we estimate that modern FPGAs may provide considerable performance/energy improvements to entire GP systems. However, there are still various challenges and unknowns, which we discuss further in the next section.

14.5 Challenges and Future Work

In this section, we discuss some challenges confronting the GP community in regard to the use of FPGA devices, and we then propose various avenues for future work.

14.5.1 Challenges

Although there is considerable potential for performance and power/energy enhancements when employing FPGA devices (Sects. 14.3 and 14.4), current technologies present at least three general challenges [66]:

1. *Productivity.* When compared to standard software development, it is generally regarded that FPGAs have low design productivity. There are multiple reasons for this, but the two main issues are that (1) designing meaningful circuits generally demands considerable *digital design* expertise and (2) device compilation can take hours or even days [66]. To reduce required efforts, many developments have taken place throughout the past few decades, such as the creation of numerous specialized programming languages [37], the extension of pre-existing programming languages (e.g., C++ and Python) [45, 57], the simplification of design tools and their compilation algorithms [63, 74], the creation of overlay circuit technologies [15, 64], and the integration of software-programmable computing cores within FPGA chips [2]. Overall, significant improvements have been made, especially in regard to *high-level synthesis (HLS)* [46], but classical techniques are still often necessary in order to implement a circuit with the desired performance/energy characteristics.[6] Ultimately, of the various challenges facing the GP community in regard to a wider adoption of FPGA devices, we believe that reduced productivity will likely be the most significant. However, for most of the community, understanding digital design is not required in order to contribute to efforts involving FPGA devices. For example, any work to reduce algorithmic complexities would likely be useful (Sect. 14.5.2).
2. *Amenability.* In general, not all algorithms are directly amenable to FPGA devices [66]. 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 [66]. For instance, pointers and recursions do not immediately make sense for most LUT-based circuits (Sect. 14.3.3), 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 [2, 34], but the mapping of certain functionality can still be challenging, and algorithmic tweaks are often necessary. Besides such 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 [66]. 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 [54, 56, 60, 66, 67, 70]. 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

[6] For some useful introductions to digital design, see [29, 68].

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 acceleration [66].[7] Whereas the development of GPUs was clearly motivated by graphics applications and, now, scientific computing, FPGAs evolved from a considerably smaller market targeting the development of "glue logic" and prototyping for ASIC devices [30, 71]. 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 listed for at least $10,000 [66]. Fortunately, this price gap has narrowed and even sometimes inverted, with recent high-end GPU systems costing anywhere from $1,000–$40,000 [47], 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 [3, 5].

14.5.2 Future Work

In the following, we establish several important avenues for future work, and we estimate difficulty with the high-level classifications *easy*, *medium*, and *hard*:

1. *Further explore "specialized CPU" architectures. (Medium to hard.)* As discussed in Sect. 14.4.1, we estimate that simpler architectures specialized to sequential execution models may be generally useful for GP, and especially useful for tree-based GP, as long as both data-level and function-level parallelism can be exploited. In work not yet publicized, we have implemented such an architecture for standard tree-based GP, as well as for a form of tree-based gradient descent, but we have not yet proven scalability. Thus, future work should explore scaling up the number of cores with such an architecture, especially in conjunction with additional complexity, such as high clock frequencies, high-bandwidth memory, and more advanced local search techniques.
2. *Improve amenability of evolutionary algorithms. (Medium to hard.)* In many instances, research ideas that are better aligned with available hardware and software resources have had a greater chance of success [33]. As such, when considering specialized hardware, there is motivation to reconsider whether state-of-the-art evolutionary algorithms are suitable for hardware, and whether any changes should be made. Of course, various trade-offs will inevitably surface if we attempt to redesign algorithms, but there may exist variants of algorithms that allow for better performance/energy characteristics while allowing for comparable (or better) solution quality. Although such work would likely best be accomplished by considering the design of hardware and algorithms in tandem, this need not be

[7] And if there is a killer app, an ASIC can likely extract additional benefits [36]. Regardless, there are still various notable applications for FPGAs, e.g., SmartNIC designs for data centers [56].

necessary. Primarily, we should strip out unnecessary algorithmic complexity and enable as much parallelism as possible. Most commonly, this may manifest as either removing control flow or reducing runtime/memory complexity. One example of this pertains to how real-valued function primitives are implemented, as we mentioned at the end of Sect. 14.1. Future work should explore how forms of evaluation, selection, variation, local search, and multi-objective optimization may be simplified, and how any combination of such changes may affect solution quality.

3. *Explore the use of FPGAs for GP inference. (Easy to medium.)* As mentioned in Sect. 14.1, FPGA devices may be separately useful for implementing final GP models, where a hardware architecture employed during this stage could be completely different from any architecture used for training. Besides the example given for tree-based GP in Sect. 14.1, we highlight that employing FPGAs for *tangled program graphs (TPGs)* may be especially interesting, given that such a GP representation can already allow for models that are thousands of times more efficient than state-of-the-art neural networks [18, 38, 39].
4. *Make FPGAs more accessible to GP researchers. (Medium.)* As mentioned in Sect. 14.5.1, although many high-level tools for utilizing FPGA technologies now exist, effectively implementing hardware still often requires considerable digital design expertise. As such, most GP researchers would likely incur a significant learning curve if attempting to design specialized hardware. Future work should explore additional technologies, techniques, and tutorials for aiding the GP community in this regard. For an initial starting point, see [29, 68].
5. *Establish new performance/energy measures. (Easy.)* The traditional measure for GP system performance, *GP operations per second (GPops/s)*, is often insufficient in that it does not reflect time taken for more complex operations (e.g., local search) nor differences in solution quality. To alleviate such issues, future work should explore additional comparison measures, such as those given in [16]. Note that a single measure need not encompass all of performance, energy, and fitness, although it is possible [16].
6. *Reconsider Boolean and fixed-point domains. (Medium.)* Performance/energy enhancements with modern FPGAs for Boolean and fixed-point domains will likely be even more pronounced. Future work should reconsider these domains.

14.6 Conclusion

Rather than just play the "hardware lottery" [33], the GP community has the potential to forge its own destiny by designing specialized hardware through FPGA devices. The computational model of GP is exceptional in that it is often embarrassingly parallel, although additional algorithmic considerations are likely needed in order to fully exploit the possible benefits of specialized architectures. This chapter explored such concepts and laid the groundwork for future efforts involving FPGA devices. Overall, 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.

References

1. Acun, B., Lee, B., Kazhamiaka, F., Maeng, K., Gupta, U., Chakkaravarthy, M., Brooks, D., Wu, C.J.: Carbon explorer: a holistic framework for designing carbon aware datacenters. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, pp. 118–132 (2023)
2. Alok, G.: Architecture apocalypse dream architecture for deep learning inference and compute - Versal AI core. Embedded World (2020)
3. Amazon: EC2 F1. https://aws.amazon.com/ec2/instance-types/f1/ (2024)
4. AMD: Versal ACAP DSP Engine Architecture Manual (AM004). https://docs.amd.com/r/en-US/am004-versal-dsp-engine/DSP58-Architecture (2018)
5. AMD: Heterogeneous Accelerated Compute Cluster (HACC) Program. https://www.amd.com/en/corporate/university-program/aup-hacc.html (2024)
6. Amdahl, G.M.: 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), pp. 483–485. Association for Computing Machinery, New York (1967). https://doi.org/10.1145/1465482.1465560
7. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction. Morgan Kaufmann, San Francisco (1998)
8. Bashir, N., Guo, T., Hajiesmaili, M., Irwin, D., Shenoy, P., Sitaraman, R., Souza, A., Wierman, A.: Enabling sustainable clouds: the case for virtualizing the energy system. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, pp. 350–358. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3472883.3487009
9. Brameier, M., Banzhaf, W., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007)
10. Burlacu, B.: GECCO'2022 symbolic regression competition: post-analysis of the Operon framework. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, GECCO '23 Companion, pp. 2412–2419. Association for Computing Machinery, New York (2023). https://doi.org/10.1145/3583133.3596390
11. Burlacu, B., Kronberger, G., Kommenda, M.: Operon C++: an efficient genetic programming framework for symbolic regression. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO '20, pp. 1562–1570. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3377929.3398099
12. Cheang, S.M., Leung, K.S., Lee, K.H.: Genetic parallel programming: design and implementation. Evol. Comput. **14**(2), 129–156 (2006). https://doi.org/10.1162/evco.2006.14.2.129
13. Chitty, D.M.: 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
14. Chromczak, J., Wheeler, M., Chiasson, C., How, D., Langhammer, M., Vanderhoek, T., Zgheib, G., Ganusov, I.: Architectural enhancements in Intel Agilex FPGAs. In: Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '20, pp. 140–149. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3373087.3375308
15. Coole, J., Stitt, G.: Adjustable-cost overlays for runtime compilation. In: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 21–24 (2015). https://doi.org/10.1109/FCCM.2015.49
16. Crary, C., Burlacu, B., Banzhaf, W.: Enhancing the computational efficiency of genetic programming through alternative floating-point primitives. In: Parallel Problem Solving from Nature. Springer (2024). https://doi.org/10.1007/978-3-031-70055-2_20

17. Crary, C., Piard, W., Stitt, G., Bean, C., Hicks, B.: Using FPGA devices to accelerate tree-based genetic programming: a preliminary exploration with recent technologies. In: European Conference on Genetic Programming (Part of EvoStar), pp. 182–197. Springer (2023). https://doi.org/10.1007/978-3-031-29573-7_12
18. Desnos, K., Bourgoin, T., Dardaillon, M., Sourbier, N., Gesny, O., Pelcat, M.: Ultra-fast machine learning inference through C code generation for tangled program graphs. In: 2022 IEEE Workshop on Signal Processing Systems (SiPS), pp. 1–6 (2022). https://doi.org/10.1109/SiPS55645.2022.9919237
19. Dobai, R., Sekanina, L.: Low-level flexible architecture with hybrid reconfiguration for evolvable hardware. ACM Trans. Reconfigurable Technol. Syst. (TRETS) **8**(3), 1–24 (2015)
20. Dréan, G.: The chips industry: Moore and Rock's laws. In: The Digital Era 2: Political Economy Revisited, pp. 125–135. Wiley Online Library (2019)
21. Eklund, S.: Time series forecasting using massively parallel genetic programming. In: Proceedings International Parallel and Distributed Processing Symposium (2003). https://doi.org/10.1109/IPDPS.2003.1213272
22. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA '11, pp. 365–376. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2000064.2000108
23. Funie, A.I., Grigoras, P., Burovskiy, P., Luk, W., Salmon, M.: Run-time reconfigurable acceleration for genetic programming fitness evaluation in trading strategies. J. Signal Process. Syst. **90**(1), 39–52 (2018). https://doi.org/10.1007/s11265-017-1244-8
24. Funie, A.I., Salmon, M., Luk, W.: A hybrid genetic-programming swarm-optimisation approach for examining the nature and stability of high frequency trading strategies. In: 2014 13th International Conference on Machine Learning and Applications, pp. 29–34 (2014). https://doi.org/10.1109/ICMLA.2014.11
25. García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H.: Estimation of energy consumption in machine learning. J. Parallel Distrib. Comput. **134**, 75–88 (2019)
26. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (CSUR) **23**(1), 5–48 (1991)
27. Goribar-Jimenez, C., Maldonado, Y., Trujillo, L., Castelli, M., Gonçalves, I., Vanneschi, L.: Towards the development of a complete GP system on an FPGA using geometric semantic operators. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp. 1932–1939 (2017). https://doi.org/10.1109/CEC.2017.7969537
28. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM **31**(5), 532–533 (1988). https://doi.org/10.1145/42411.42415
29. Harris, S.L., Harris, D.: Digital Design and Computer Architecture. Morgan Kaufmann (2015)
30. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Sixth Edition: A Quantitative Approach, 6th edn. Morgan Kaufmann Publishers Inc., San Francisco (2017)
31. Hettiarachchi, D.L.N., Davuluru, V.S.P., Balster, E.J.: Integer vs. floating-point processing on modern FPGA technology. In: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0606–0612 (2020). https://doi.org/10.1109/CCWC47524.2020.9031118
32. Heywood, M.I., Zincir-Heywood, A.N.: Register based genetic programming on FPGA computing platforms. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) Genetic Programming, pp. 44–59. Springer, Berlin (2000)
33. Hooker, S.: The hardware lottery. Commun. ACM **64**(12), 58–65 (2021). https://doi.org/10.1145/3467017
34. Intel: Intel Agilex™ M-Series FPGA and SoC FPGA Product Table (2015). https://cdrdv2.intel.com/v1/dl/getContent/721636
35. Intel: BFLOAT16 – Hardware Numerics Definition White Paper (2018). https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf

36. Jouppi, N., Young, C., Patil, N., Patterson, D.: 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
37. Kapre, N., Bayliss, S.: Survey of domain-specific languages for FPGA computing. In: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–12 (2016). https://doi.org/10.1109/FPL.2016.7577380
38. Kelly, S., Heywood, M.I.: Emergent solutions to high-dimensional multitask reinforcement learning. Evol. Comput. **26**(3), 347–380 (2018)
39. Kelly, S., Smith, R.J., Heywood, M.I.: Emergent Policy Discovery for Visual Reinforcement Learning Through Tangled Program Graphs: A Tutorial, pp. 37–57. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-04735-1_3
40. Kommenda, M., Burlacu, B., Kronberger, G., Affenzeller, M.: Parameter identification for symbolic regression using nonlinear least squares. Genet. Program Evolvable Mach. **21**(3), 471–501 (2020)
41. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
42. Koza, J.R., Bennett, F.H., Hutchings, J.L., Bade, S.L., Keane, M.A., Andre, D.: Evolving computer programs using rapidly reconfigurable field-programmable gate arrays and genetic programming. In: Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, FPGA '98, pp. 209–219. Association for Computing Machinery, New York (1998). https://doi.org/10.1145/275107.275141
43. Krishnakumar, A., Ogras, U., Marculescu, R., Kishinevsky, M., Mudge, T.: Domain-specific architectures: research problems and promising approaches. ACM Trans. Embed. Comput. Syst. **22**(2) (2023). https://doi.org/10.1145/3563946
44. La Cava, W., et al.: Contemporary symbolic regression methods and their relative performance. In: Vanschoren, J., Yeung, S., (eds.) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, vol. 1 (2021)
45. Lahti, S., Rintala, M., Hämäläinen, T.D.: 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
46. Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T.D.: 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
47. Leswing, Kif: Nvidia's latest AI chip will cost more than $30,000, CEO says (2024). https://www.cnbc.com/2024/03/19/nvidias-blackwell-ai-chip-will-cost-more-than-30000-ceo-says.html
48. L'Ecuyer, P.: Maximally equidistributed combined tausworthe generators. Math. Comput. **65**(213), 203–213 (1996)
49. Maldonado, Y., Salas, R., Quevedo, J.A., Valdez, R., Trujillo, L.: GSGP-hardware: Instantaneous symbolic regression with an FPGA implementation of geometric semantic genetic programming. Genet. Program. Evolvable Mach. **25**(2), 18 (2024). https://doi.org/10.1007/s10710-024-09491-5
50. Martin, P.: A hardware implementation of a genetic programming system using FPGAs and Handel-C. Genet. Program Evolvable Mach. **2**(4), 317–343 (2001). https://doi.org/10.1023/A:1012942304464
51. McKay, R.I., Hoai, N.X., Whigham, P.A., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genet. Program Evolvable Mach. **11**(3), 365–396 (2010). https://doi.org/10.1007/s10710-010-9109-y
52. Miller, J.F.: Cartesian genetic programming: its status and future. Genet. Program Evolvable Mach. **21**(1), 129–168 (2020). https://doi.org/10.1007/s10710-019-09360-6
53. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric semantic genetic programming. In: Coello, C.A.C., Cutello, V., Deb, K., Forrest, S., Nicosia, G., Pavone, M. (eds.) Parallel Problem Solving from Nature - PPSN XII, pp. 21–31. Springer, Berlin (2012)

54. Nurvitadhi, E., et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pp. 5–14. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3020078.3021740
55. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK Ltd (2008)
56. Putnam, A., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro **35**(3), 10–22 (2015). https://doi.org/10.1109/MM.2015.42
57. Quenon, A., Ramos Gomes Da Silva, V.: Towards higher-level synthesis and co-design with Python. In: Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21). ACM New York (2021)
58. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genet. Program Evolvable Mach. **10**(4), 447–471 (2009). https://doi.org/10.1007/s10710-009-9092-3
59. Roser, M., Ritchie, H., Mathieu, E.: What is Moore's Law? Our World in Data (2023). https://ourworldindata.org/moores-law
60. Ruiz-Rosero, J., Ramirez-Gonzalez, G., Khanna, R.: Field programmable gate array applications–a scientometric review. Computation **7**(4) (2019). https://doi.org/10.3390/computation7040063
61. Salvador, R., Otero, A., Mora, J., de la Torre, E., Riesgo, T., Sekanina, L.: Self-reconfigurable evolvable hardware system for adaptive image processing. IEEE Trans. Comput. **62**(8), 1481–1493 (2013)
62. Sidhu, R.P.S., Mei, A., Prasanna, V.K.: Genetic programming using self-reconfigurable FPGAs. In: Lysaght, P., Irvine, J., Hartenstein, R., (eds.) Field Programmable Logic and Applications, pp. 301–312. Springer, Berlin (1999). https://doi.org/10.1007/978-3-540-48302-1_31
63. Skalicky, S., Monson, J., Schmidt, A., French, M.: Hot & Spicy: improving productivity with Python and HLS for FPGAs. In: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 85–92 (2018). https://doi.org/10.1109/FCCM.2018.00022
64. So, H.K.H., Liu, C.: FPGA Overlays, pp. 285–305. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-26408-0_16
65. Spector, L., Robinson, A.: 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
66. Stitt, G.: Are field-programmable gate arrays ready for the mainstream? IEEE Micro **31**(6), 58–63 (2011). https://doi.org/10.1109/MM.2011.99
67. Stitt, G., Gupta, A., Emas, M.N., Wilson, D., Baylis, A.: Scalable window generation for the Intel Broadwell+Arria 10 and high-bandwidth FPGA systems. In: Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '18, pp. 173–182. Association for Computing Machinery (2018). https://doi.org/10.1145/3174243.3174262
68. Stitt, G.: VHDL and SystemVerilog Tutorials. https://stitt-hub.com/vhdl-and-systemverilog-tutorials/ (2024)
69. Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for modern deep learning research. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 13,693–13,696 (2020)
70. Tan, T., Nurvitadhi, E., Shih, D., Chiou, D.: Evaluating the highly-pipelined Intel Stratix 10 FPGA architecture using open-source benchmarks. In: 2018 International Conference on Field-Programmable Technology (FPT), pp. 206–213 (2018). https://doi.org/10.1109/FPT.2018.00038
71. Tessier, R., Pocek, K., DeHon, A.: Reconfigurable computing architectures. Proc. IEEE **103**(3), 332–354 (2015). https://doi.org/10.1109/JPROC.2014.2386883
72. Vašíček, Z., Sekanina, L.: Hardware accelerators for cartesian genetic programming. In: Genetic Programming: 11th European Conference, EuroGP 2008, Naples, Italy, March 26–28, 2008. Proceedings 11, pp. 230–241. Springer (2008)

73. Vašíček, Z., Sekanina, L.: Hardware accelerator of cartesian genetic programming with multiple fitness units. Comput. Inf. **29**(6+), 1359–1371 (2010)
74. Vipin, K., Fahmy, S.A.: FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. ACM Comput. Surv. **51**(4) (2018). https://doi.org/10.1145/3193827
75. Yao, X.: Following the path of evolvable hardware. Commun. ACM **42**(4), 46–49 (1999). https://doi.org/10.1145/299157.299169
76. Yao, X., Higuchi, T.: Promises and challenges of evolvable hardware. IEEE Trans. Syst., Man, Cybern., Part C (Appl. Rev.) **29**(1), 87–97 (1999). https://doi.org/10.1109/5326.740672