Synthesized Differentiable Programs

Anonymous Author(s)
Affiliation
Address
email

Abstract

Program synthesis algorithms produce interpretable and generalizable code that captures input data but are not directly amenable to continuous optimization using gradient descent. In theory, any program can be represented in a Turing complete neural network model, which implies that it is possible to compile syntactic programs into the weights of a neural network by using a technique known as *neural compilation*. This paper presents a combined algorithm for synthesizing syntactic programs, compiling them into the weights of a neural network, and then tuning the resulting model. This paper's experiments establish that program synthesis, neural compilation, and differentiable optimization together form an efficient algorithm for inducing abstract algorithmic structure and a corresponding local set of desirable complex programs

1 Introduction

Program synthesis efficiently induces abstract computer programs from data. Alternatively, gradient optimization induces parameterized functions which can be seen as a relaxed form of program search [1, 2, 3]. However, programs recovered via gradient optimization will be represented as real-valued weights, in contrast to code in a higher-level language. Generally, program synthesis is more appropriate for finding abstract algorithmic structures and gradient optimization is a flexible but less specialized technique for relaxed program induction. This paper unifies these two paradigms by leveraging *neural compilation and decompilation*: techniques for transforming code into neural network weights and transforming weights back into code [4, 5, 6, 7]. This hybrid algorithm retains both the generalization of program synthesis and the flexibility of gradient optimization.

The closest ideas to this paper are forms of *neurosymbolic programming* [3], and AutoML, which each mix elements of program synthesis, symbolic search, and differentiable computing [8, 9, 10, 11, 12, 13, 14, 15, 16, 17]. However, program synthesis combined with neural compilation and optimization is a unique and direct form of hybrid discrete-continuous neurosymbolic search.

Neural Compilation The neural compilation algorithm in this paper is a replication of [6]. Historically, [4] established the Turing completeness of neural networks, which implies the existence of a neural compiler: a function that maps any Turing-complete program into the weights of a neural network. Shortly after, [5] created the first neural compiler, based on Pascal. However, this neural compiler could not tune compiled programs using gradient descent. Accordingly, [6] created the first neural compiler which was *adaptive* and could be locally tuned with gradient descent. This focused on a minimal assembly language that ran on a minimal differentiable computer, a type of recurrent neural network with explicit memory and addressing schemes. Afterward, [7] created a neural interpreter for a higher-level language called forth, which used a differentiable stack machine. However, both [6] and [7] utilized human-written programs as initializations for optimization. In contrast, this paper utilizes program synthesis as a method for efficiently finding abstract algorithmic structures.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

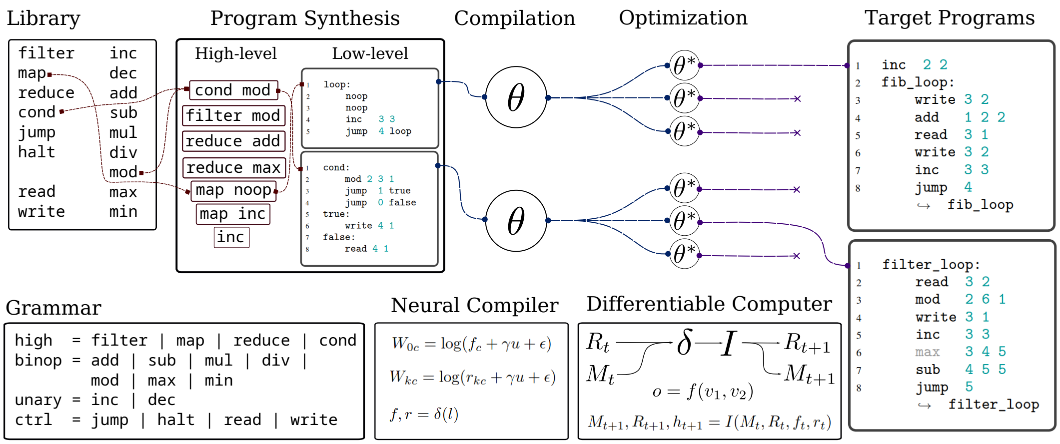

Figure 1: A neurosymbolic program induction algorithm which leverages program synthesis to find abstract algorithmic structure that is compiled into weights and optimized to find specific programs

Program Synthesis Program synthesis was anticipated as early as Turing, and underlied Solomonoff's theory of inductive inference [18, 19], though the first critical historical milestone for program synthesis was the FlashFill program [20]. Components such as version space algebras, equivalence-Graphs, and synthesis through unification were essential in efficiently searching the combinatorial space of computer programs, which is otherwise intractable [1, 21, 22, 23, 24]. However, an even more critical feature of effective program synthesis is *abstraction*: the ability to create a customized library of higher-level programs which capture common patterns, making program induction more efficient [25, 26, 27, 28]. Finding abstractions and composing them is a central feature of human intelligence, and therefore also central for machines [29, 30, 31, 32, 33].

Differentiable Computing The minimal differentiable computer in this paper builds on recurrent neural networks [34] and their many extensions [35, 36, 37, 38, 39]. These architectures aim to induce programs from data via gradient optimization. However, doing so is challenging due to an overabundance of local but suboptimal solutions, as well as technical issues such as unstable gradients [40]. Despite this, with the correct formulation and resources, it is possible to recover interesting algorithms, such as simple planning or sorting algorithms [37].

2 Algorithm

The algorithm in this paper (depicted in Figure 1) has three primary steps: synthesis, compilation, and optimization. First, a program synthesis algorithm searches a high-level language for abstract program templates. Then, these programs are mapped into the minimal assembly language specified in [6]. A neural compiler converts this lower-level assembly program into the weights of a neural network, which then act as the initialization for optimization. Finally, the optimization algorithm tunes this program by using gradient descent. Optimization is repeated multiple times, as algorithmic induction is highly sensitive to initialization.

Minimal assembly language acts as a common interface between synthesis and optimization, and neural compilation is the means for using this interface. While minimal assembly is easily compiled, it is unideal for direct program synthesis, even when using equivalence graphs or version space algebras. For example, in a 4 register machine with 13 instructions, there are roughly 100 million distinct instruction-argument pairs in only three lines of minimal assembly. Because of this, designing an appropriate high-level language plays a large part in the success of the overall algorithm, especially since the language design controls what abstract algorithm templates are included. By carefully manipulating this, it should be possible to recover desirable target programs reliably.

2.1 Neural Compilation

Fundamentally, the minimal differentiable computer is a recurrent neural network with a controller δ, interpreter I, language L, memory tensor M, register tensor R, and halt state h: (δ, I, L, M, R, h). The controller outputs an instruction f and register arguments r based on the instruction register l:

$$f, r = \delta(l) \tag{1}$$

Where f is a one-hot encoding corresponding to an assembly instruction, and r contains three one-hot encodings specifying register arguments. Typically the first two registers are inputs, and the third register is used to store output, such as `add(`r_1`, `r_2`, `r_3`)`, which adds the values in r_1 and r_2 and stores the result in r_3. The controller is parameterized with four weight matrices W_k, which determine the function f, and the arguments r as functions of l, the instruction register:

$$f = \texttt{softmax}(W_0 l) \qquad a_k = \texttt{softmax}(W_k l) \quad k > 0 \tag{2}$$

Neural compilation works by inverting `softmax` and setting W_k to produce a desired instruction (f, r) at instruction count c. Uniform noise u is added with a magnitude γ, which allows flexible optimization but preserves the desired instruction. A small constant ϵ is added for numerical stability:

$$W_{0c} = \log(f_c + \gamma u + \epsilon) \qquad W_{kc} = \log(r_{kc} + \gamma u + \epsilon) \quad k > 0 \tag{3}$$

The machine interpreter I is a function that uses the recurrent state (memory and registers) and the instruction specified by the controller:

$$M_{t+1}, R_{t+1}, h_{t+1} = I(M_t, R_t, f_t, r_t) \tag{4}$$

First, arguments r_{kt} are resolved to their values v_{kt} by a register lookup:

$$v_{kt} = r_{kt} R_t \tag{5}$$

Many functions, such as `add` only depend on input registers, and not on memory state. For a machine in base b, outputs are stored in a $|L| \times b \times b \times b$ lookup table T, where the first dimension corresponds to a function f, the second two dimensions represent values v_1 and v_2, and the final dimension encodes the output of $f(v_1, v_2)$. For the `read` instruction, the $b \times \mathbf{1} \times b$ sub-tensor of T corresponding to reading is set to the current memory, M_t, and for special instructions `write, jump, halt`, sub-tensors of T are zero. T is indexed differentiably using an Einstein summation, which is analogous to using an addition or multiplication table, but for all assembly instructions and arguments

$$o_t = \texttt{einsum}(klmn, k, l, m \rightarrow n, T, f_t, v_{1t}, v_{2t}) \tag{6}$$

Then, registers are updated with a soft write parameterized by r_3, the output argument:

$$R_{t+1} = R_t \odot (1 - r_{3t}) + o_t \otimes r_{3t} \tag{7}$$

Writing to memory uses w_t, the scalar component of f_t representing the write probability.

$$M_{t+1} = (1 - w_t)M_t + w(1 - v_1 t) \cdot \mathbf{1} \odot M_t + v_{1t} \odot v_{2t} \tag{8}$$

The jump instruction modifies the instruction register l probabilistically using j, a scalar component of f_t representing the jump probability, and z, the scalar component of v_{1t} representing the probability that v_{1t} is zero. $T_{\texttt{inc}}$ denotes the sub-tensor of T for the increment instruction, and l_n would be the next instruction if the jump is not taken.

$$l_n = l_t \cdot T_{\texttt{inc}} \qquad l_{t+1} = l_n(1 - j) + r_{2t} z + j l_n (1 - z) \tag{9}$$

Finally, the halting probability h_t is simply a scalar component of f_t.

2.2 Optimization

Once a program has been compiled into program weights, it is optimized using the adam optimizer [41], and a loss function with two components: correctness and efficiency. Correctness is a masked cross-entropy loss between a predicted tensor P and labels L across the final dimension. μ is a vector mask across the first dimension. Correctness is calculated for registers R and memory M:

$$\underset{\text{correctness}}{\mathcal{L}(P, L, \mu)} = \mu \odot \texttt{cross\_entropy}(P, L) \tag{10}$$

Efficiency is a differentiable penalty for the number of computation steps:

$$h_{t>k} = \texttt{max}(h_{t \leq k}) \qquad \underset{\text{efficiency}}{\mathcal{L}(h_t)} = \sum \mathbf{1} - h_t \tag{11}$$

And the composite loss is a weighted combination of the correctness and efficiency losses:

$$\underset{\text{composite}}{\mathcal{L}(\hat{M}, \hat{R}, M, R, h, \mu)} = \lambda(\underset{\text{correctness}}{\mathcal{L}(\hat{M}, M, \mu_M) + \mathcal{L}(\hat{R}, R, \mu_R)}) + \lambda \underset{\text{efficiency}}{\mathcal{L}(h)} \tag{12}$$

Neural networks and optimization components are implemented in `jax` and `equinox` [16, 17].

```
map_loop:
    read  1 2
    inc   2 2
    write 1 2
    inc   1 1
    jump  3
    ↪ map_loop
```

```
inc 4 4
inc 4 4
sum_loop:
    read  3 2
    add   1 2 1
    inc   3 3
    max   3 4 5
    sub   4 5 5
    jump  5
    ↪ sum_loop
write 3 1
```

```
inc  2 2
fib_loop:
    write 3 2
    add   1 2 2
    read  3 1
    write 3 2
    inc   3 3
    jump  4
    ↪ fib_loop
```

Listing 1: Minimal assembly code for map, sum-reduce, and fibonacci functions

3 Experiments

These experiments explore which algorithms can be recovered via program synthesis, optimization, or a combined algorithm. An ideal evaluation task involves high-level algorithmic structure that can be established via program synthesis but contains sub-components that are continuous or best optimized as neural networks. Program synthesis finds the overall structure of a program, and the local optimizer tunes this program locally. The primary experiment uses a budget of $k = 100$ optimization runs and compares structured initializations to random initializations. Since many algorithms share a common structure (recursion, looping, conditionals, etc), starting with an algorithm template acts as a positive inductive bias, similar to how the choice of network architecture affects program behavior. Recovery is based on observational equivalence over a dataset of sampled program outputs. This allows recovering syntactically different solutions to a problem and discourages overfitting to a particular input-output pair.

Generally, algorithmic skeletons are better initializations than random initialization, but it is common for differentiable tuning to discard large parts of algorithm structure in certain problems. Since program synthesis finds various algorithmic skeletons, it outperforms using multiple uniform random initializations. Even programs that aren't directly enumerated, such as the Fibonacci program (Listing 1, Table 1), can be recovered using the combination of synthesis and tuning. Introducing no-ops into program synthesis (and not penalizing them) can be advantageous, as gradient descent tuning does not naturally model concepts like insertion. Table 1 includes no-op-padded program initializations in the second half. Interestingly, a few results defy intuition, such as that `inc` is harder to find, and that `map dec` is not transitive with `map inc`, we hypothesize that this is because it is difficult for optimization to represent simpler programs, as it typically saturates the available instructions. A preliminary grid search found a noise parameter in the neighborhood of $\gamma = 0.3$, which is sufficient for gradient information to capture the local program space.

Table 1: Recovery rates for selected algorithms and initializations

Algorithm	`inc`	`map inc`	`map dec`	`reduce`	Parity	Fibonacci
Optimization	19%	86%	56%	41%	95%	4%
Synthesis	100%	100%	100%	100%	100%	**0**%
Both	100%	100%	100%	100%	100%	**75**%
Initializations						
`map inc`	-	100%	7%	-	7%	26%
`map dec`	-	45%	100%	-	6%	41%
`loop no-op`	100%	100%	100%	100%	49%	**75**%

Table 1 shows the percent of perfect algorithms recovered for each algorithm and different initial program structures. Program synthesis will recover many of the program structures listed in this table, some of which will be near-misses to a desired program. Then, differentiable tuning can find a local variant of the program that is close to a desired program. This shows that, for this neural architecture, the combined synthesis-compilation algorithm is more computationally efficient than optimization alone.

4 Limitations & Future Work

While the neural compilation method introduced by [6] is straightforward to compute and implement, it could be more adaptive and general. One major limitation is the lack of parameters in the network model: each instruction and its arguments are determined only from the instruction register, and the function used is linear with a softmax activation. For example, in a network model for a 32 instruction program, there are only $3,640$ parameters. While this is desirable for some applications, it is in contrast to implementations such as [34, 35, 36, 37] where network behavior is a function of memory and input, and modern network architectures that have millions or billions of parameters. Also, using a recurrent neural network inherently makes representing long programs and sequences difficult because of the unstable gradient problem. Future work will explore neural compilation techniques that are more adaptive and tunable but retain interpretability.

The minimal differentiable computer introduced in [6] is a relatively weak program induction baseline. Future work will include stronger end-to-end differentiable algorithm induction baselines, especially modern architectures [37, 39]. However, the minimal differentiable computer is highly compute and parameter efficient.

The program synthesis algorithm given in this paper is relatively simple compared to modern techniques. In particular, it does not generate abstractions or utilize neural search heuristics such as those in [25]. These elements are modular and would most likely boost performance, especially if used in tandem with differentiability-based tuning. Finally, given sufficient computing power and time, a more advanced version of this algorithm would likely be successful on more interesting tasks, such as sorting or planning algorithms that are embedded in larger neural programs.

References

[1] Armando Solar-Lezama. *Introduction to Program Synthesis*. URL: https://people.csail.mit.edu/asolar/SynthesisCourse/ (visited on 08/27/2021).

[2] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010. URL: http://dx.doi.org/10.1561/2500000010.

[3] Swarat Chaudhuri et al. "Neurosymbolic programming". In: *Foundations and Trends in Programming Languages* 7.3 (2021), pp. 158–243. ISSN: 23251131. DOI: 10.1561/2500000049.

[4] Hava T. Siegelmann. "Computation beyond the turing limit". In: *Science* 268.5210 (1995), pp. 545–548. ISSN: 00368075. DOI: 10.1126/science.268.5210.545.

[5] Frédéric Gruau, Jean Yves Ratajszczak, and Gilles Wiber. "A neural compiler". In: *Theoretical Computer Science* 141.1-2 (1995), pp. 1–52. ISSN: 03043975. DOI: 10.1016/0304-3975(94)00200-3.

[6] Rudy Bunel et al. "Adaptive neural compilation". In: *Advances in Neural Information Processing Systems* (2016), pp. 1452–1460. ISSN: 10495258. arXiv: 1605.07969.

[7] Matko Bošnjak et al. "Programming with a differentiable forth interpreter". In: *34th International Conference on Machine Learning, ICML 2017*. Vol. 2. 2017, pp. 842–859. ISBN: 9781510855144. arXiv: 1605.06640. URL: http://proceedings.mlr.press/v70/bosnjak17a.html?ref=https://githubhelp.com.

[8] Dweep Trivedi et al. "Learning to Synthesize Programs as Interpretable and Generalizable Policies". In: (2021). arXiv: 2108.13643. URL: http://arxiv.org/abs/2108.13643.

[9] Lazar Valkov et al. *HOUDINI: Lifelong Learning as Program Synthesis*.

[10] Thomas Pierrot Instadeep et al. "Learning Compositional Neural Programs with Recursive Tree Search and Planning". In: *Advances in Neural Information Processing Systems* 32 (2019). URL: https://github.com/instadeepai/AlphaNPI.

[11] Xinyun Chen et al. "Compositional Generalization via Neural-Symbolic Stack Machines". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1690–1701.

[12] Alexander L. Gaunt et al. "Differentiable Programs with Neural Libraries". In: *34th International Conference on Machine Learning, ICML 2017* 3 (2016), pp. 1965–1977. DOI: 10.48550/arxiv.1611.02109. arXiv: 1611.02109. URL: https://arxiv.org/abs/1611.02109v2.

[13] Ameesh Shah et al. *Learning Differentiable Programs with Admissible Neural Heuristics*. Tech. rep. arXiv: 2007.12101v5.

[14] Esteban Real et al. “AutoML-Zero: Evolving Machine Learning Algorithms From Scratch”. In: *37th International Conference on Machine Learning, ICML 2020* PartF16814 (2020), pp. 7963–7975. arXiv: 2003.03384. URL: https://arxiv.org/abs/2003.03384v2.

[15] Daiyi Peng et al. “PyGlove: Symbolic programming for automated machine learning”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 96–108.

[16] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[17] Patrick Kidger and Cristian Garcia. “Equinox: neural networks in JAX via callable PyTrees and filtered transformations”. In: *arXiv preprint arXiv:2111.00254* (2021).

[18] B.J. Copeland. *Alan Turing’s Electronic Brain: The Struggle to Build the ACE, the World’s Fastest Computer*. OUP Oxford, 2012. ISBN: 9780199609154. URL: https://books.google.com/books?id=YhQZnczOS7kC.

[19] J Solomonoff Raymond. “A formal theory of inductive inference i”. In: *Information and Control* 7 (1964), pp. 1–22.

[20] Sumit Gulwani. “Automating String Processing in Spreadsheets Using Input-output Examples”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: ACM, 2011, pp. 317–330. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926423. URL: http://doi.acm.org/10.1145/1926385.1926423.

[21] Program Synthesis. “Emina Torlak Computer-Aided Reasoning for Software”. In: ().

[22] Tessa Lau et al. “Programming by Demonstration Using Version Space Algebra”. In: *Mach. Learn.* 53.1-2 (2003), pp. 111–156. ISSN: 0885-6125. DOI: 10.1023/A:1025671410623. URL: https://doi.org/10.1023/A:1025671410623.

[23] Tom M. Mitchell. “Generalization as Search”. In: *Artif. Intell.* 18.2 (1982), pp. 203–226. DOI: 10.1016/0004-3702(82)90040-6. URL: https://doi.org/10.1016/0004-3702(82)90040-6.

[24] Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. “Synthesis through unification”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9207 (2015), pp. 163–179. ISSN: 16113349. DOI: 10.1007/978-3-319-21668-3_10/FIGURES/3. arXiv: 1505.05868. URL: https://link.springer.com/chapter/10.1007/978-3-319-21668-3_10.

[25] Kevin Ellis et al. “DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning”. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2021), pp. 835–850. DOI: 10.1145/3453483.3454080.

[26] Kevin Ellis et al. “Library Learning for Neurally-Guided Bayesian Program Induction”. In: ().

[27] Matthew Bowers et al. “Top-Down Synthesis For Library Learning”. In: *Proceedings of the ACM on Programming Languages* 1 (2022). URL: https://doi.org/.

[28] Kensen Shi et al. *CROSSBEAM: LEARNING TO SEARCH IN BOTTOM-UP PROGRAM SYNTHESIS*. URL: https://github.com/google-research/crossbeam.

[29] Melanie Mitchell. “Abstraction and Analogy-Making in Artificial Intelligence”. In: (). arXiv: 2102.10717v2.

[30] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. “Human-level concept learning through probabilistic program induction”. In: *Science* 350.6266 (2015), pp. 1332–1338.

[31] Brenden M Lake et al. “Building machines that learn and think like people”. In: *Behavioral and brain sciences* 40 (2017).

[32] Brenden M Lake. “Compositional generalization through meta sequence-to-sequence learning”. In: (). arXiv: 1906.05381v2.

[33] Miguel Lázaro-Gredilla et al. “Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs”. In: (2018). arXiv: 1812.02788v1.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[35] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[36] Alex Graves and Greg Wayne. "Neural Turing Machines". In: (). arXiv: 1410.5401v2.

[37] Alex Graves and Google Deepmind. "Adaptive Computation Time for Recurrent Neural Networks". In: (). arXiv: 1603.08983v6.

[38] Scott Reed and Nando De Freitas. "Neural Programmer-Interpreters". In: *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings* (2015). DOI: 10.48550/arxiv.1511.06279. arXiv: 1511.06279. URL: https://arxiv.org/abs/1511.06279v4.

[39] Mostafa Dehghani et al. "Universal transformers". In: *arXiv preprint arXiv:1807.03819* (2018).

[40] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. PMLR. 2013, pp. 1310–1318.

[41] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

A Appendix

A.1 Interpreting distributed programs

Differentiable programs are *distributed*, because the instruction register is a probability distribution. This means that multiple instructions can be carried out at once, which makes one-to-one decompilation difficult, and also prevents these programs from being easily human-interpretable. Distributed execution is affected by two factors in our model: the probability that a current instruction is a jump instruction, and the probability that the comparison register for the jump instruction is equal to zero. Also, every operation is distributed, so each register's values and all memory values are multinomial distributions created by softmax, which overlap with one another. Thus, in a longer non-trivial program, decompiling network weights into a one-to-one interpretation is more difficult. However, starting with a decompilable algorithm increases the probability that a tuned algorithm will be interpretable, as the initialized algorithm is less distributed than a naturally recovered algorithm.

```
read  0 1
inc   1 1
read  1 1
write 0 1
halt
```

```
map_loop:
    read  1 2
    inc   2 2
    write 1 2
    inc   1 1
    jump  3
      ↪ map_loop
```

```
cond:
    mod 2 3 1
    jump  1 true
    jump  0 false
true:
    write 4 1
false:
    read 4 1
```

```
loop:
    noop
    noop
    inc   3 3
    jump  4 loop
```

```
filter_loop:
    read  3 2
    mod   2 6 1
    write 3 1
    inc   3 3
    max   3 4 5
    sub   4 5 5
    jump  5
      ↪ filter_loop
```

```
inc  2 2
fib_loop:
    write 3 2
    add   1 2 2
    read  3 1
    write 3 2
    inc   3 3
    jump  4
      ↪ fib_loop
```