Ultra-Fast Machine Learning Inference through C Code Generation for Tangled Program Graphs

Karol Desnos, Thomas Bourgoin, Mickael Dardaillon, Nicolas Sourbier, Olivier Gesny, Maxime Pelcat

▶ **To cite this version:**

Karol Desnos, Thomas Bourgoin, Mickael Dardaillon, Nicolas Sourbier, Olivier Gesny, et al.. Ultra-Fast Machine Learning Inference through C Code Generation for Tangled Program Graphs. 2022 IEEE Workshop on Signal Processing Systems (SiPS), Nov 2022, Rennes, France. pp.1-6, 10.1109/SiPS55645.2022.9919237 . hal-03845227

HAL Id: hal-03845227
https://hal.science/hal-03845227v1

Submitted on 9 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Ultra-Fast Machine Learning Inference through C Code Generation for Tangled Program Graphs

Karol Desnos*, Thomas Bourgoin*, Mickaël Dardaillon*, Nicolas Sourbier*, Olivier Gesny† and Maxime Pelcat*
*Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, 35000 Rennes, France. first.last@insa-rennes.fr
†Silicom, 35000 Rennes, France. ogesny@silicom.fr

***Abstract*— Tangled Program Graph (TPG) is a Reinforcement Learning (RL) technique based on genetic programming concepts. On state-of-the-art learning environments, TPGs have been shown to offer comparable competence with Deep Neural Networks (DNNs), for a fraction of their computational and storage cost.**

The contribution of this paper focuses on accelerating the inference of pre-trained TPGs, through the generation of standalone C code. While the training process of TPGs, based on genetic evolution principles, requires the use of flexible data structures supporting random mutations, this flexibility is no longer needed when focusing on the inference process.

Evaluation of the proposed approach on four computing platforms, including embedded CPUs, produces an acceleration of the TPG inference by a factor 50 compared to state-of-the-art implementations. The inference performance obtained within a complex RL environment range between hundreds of nano-seconds to micro-seconds, making this approach highly competitive for edge Artificial Intelligence (AI).

***Index Terms*—machine learning, Tangled Program Graph, embedded systems**

I. Introduction

In a decade, Artificial Intelligences (AIs) powered by Deep Neural Networks (DNNs) have superseded conventional algorithms in many domains, from computer vision [4] to robotics [8]. While the capabilities of DNNs are impressive, their huge computational complexity is an obstacle to their integration in edge computing platforms, where computing, memory, and energy resources are scarce. To embed AIs in edge computing platform, an alternative approach is to develop new machine learning techniques that rely on light-by-construction models, such as the TPG Reinforcement Learning (RL) model studied in this paper.

TPG, which stands for Tangled Program Graph, is a RL model proposed by Kelly and Heywood in [10]. Building on state-of-the-art genetic programming techniques, Tangled Program Graphs (TPGs) are grown from scratch for each learning environment in which they are trained. Hence, the topology and the complexity of the TPG adapt themselves to the complexity of the learned task, without requiring an expert to select an appropriate network structure. In recent works [7], [9]–[11], TPGs have proven to be a very promising model for building AIs, being competitive with state-of-the-art DNNs for a fraction of their computation and memory cost, both for training and inference.

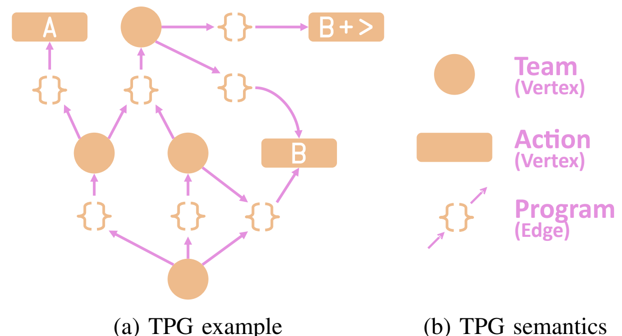
This paper introduces a new design flow to enable ultra-fast and lightweight inference RL agent based on the TPG model. The core of this contribution consists of translating pre-trained TPG graphs into standalone and standard C code, compilable without depending on any third-party library. Resulting inference times measured with the generated code are 24 to 85 times faster than inference within a state-of-the-art framework for TPGs, executed on identical hardware.

The TPG model and motivations behind this work are introduced in Section II. Related works on the acceleration of machine learning techniques are reviewed in Section III. Then, Section IV details the code generation process at the core of this paper. Section V evaluates and discusses the inference performance obtained on four different general purpose and embedded platforms with a state-of-the-art RL environment. Finally, Section VI concludes this paper.

II. Context & Motivations

A. Tangled Program Graph (TPG)

The semantics of the Tangled Program Graph (TPG) model, depicted in Figure 1, consists of three elements composing a direct graph: *programs*, *teams* and *actions*. The *teams* and the *actions* are the vertices of the graph, *teams* being internal vertices, and *actions* being the leaves of the graph. The *programs*, associated to the edges of the graph that each connects a source *team* to a destination *team* or *action* vertex. Self-loops, that is an edge connecting a *team* to itself, are not allowed in TPGs.



(a) TPG example (b) TPG semantics

Fig. 1: Semantics of the Tangled Program Graphs (TPGs)

From afar, a *program* can be seen as a black box that takes the current state of the learning environment as an input, processes it, and produces a real number, called a *bid*, as a result. In more detail, a *program* is a sequence of arithmetic *instructions*, like additions or exponents. As

depicted in Figure 2, each *instruction* takes as an operand either data coming from the observed learning environment, or the value stored in a register by a previous *instruction*. The last value stored in a specific register, generally called `R0`, is the result *bid* produced by the *program*.

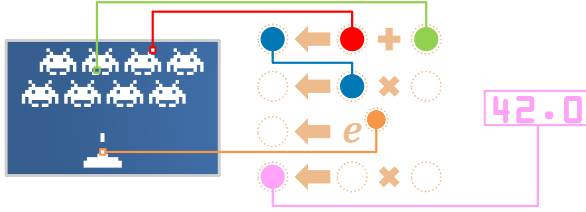


Fig. 2: *Program* from a TPG. On the left, the learning environment state fed to the *program*. In the middle, the sequence of instructions of the *program*. On the right, the result produced by the program.

The execution of a TPG starts from its unique root *team*, when a new state of the environment becomes available. All *programs* associated to outgoing edges of the root *team* are executed with the current state of the environment as their input. Once all *programs* have completed their execution, the edge associated to the largest *bid* is identified, and the execution of the TPG continues following this edge. If another *team* is pointed by this edge, its outgoing *programs* are executed, still with the same input state, and the execution continues along the edge with the largest *bid*[1]. Eventually, the edge with the largest *bid* leads to an *action* vertex. In this case, the *action* is executed by the learning agent, a new resulting state of the environment is received, and the TPG execution restarts from its root *team*.

The genetic evolution process of a TPG relies on a graph with several root *teams*. At a given generation of the learning process, each root *team* of the TPG represents a different *policy* whose fitness is evaluated. Worst-fitting root *teams*, which obtained the lowest rewards, are deleted from the TPG. To create new root *teams* for the next generation of the evolution process, randomly selected remaining *teams* from the TPG are duplicated with all their outgoing edges. Then, these new edges undergo a random mutation process. A detailed description of this evolution process can be found in [9].

B. Motivations and Opportunities for Inference Acceleration

The main motivation behind this work is to accelerate the inference of AIs based on the TPG model by decreasing the computational complexity of the inference process. A first benefit of accelerating the inference of TPGs is to obtain better response time for AI agent implemented with this model, which may be a strong asset for integration in tightly constrained real-time systems. A second benefit of faster inference time is energy savings. Indeed, reducing the computation time needed to obtain the same result will lighten the computational load of the processor used, which generally translates into power savings. A third benefit of decreasing the computational complexity of TPG inference is that it may allow developers to fulfill their time constraints while using a less powerful, less expensive, and less environmentally impacting hardware for their systems.

[1]If a team is visited several times, previously taken edges are ignored to avoid infinite loops.

In this paper, we focus on accelerating the inference TPG, that is, the execution of a TPG whose graph topology and *programs* have been fixed during a training process. The consideration of a TPG with a fixed topology creates many opportunities for accelerations:

- *Removing overhead for dynamic TPG structures.* During the training process, the topology of the TPG *teams*, its number of *programs*, and the *instruction* list of each *program* may change as a result of the genetic mutation process. In order to support these constant mutations, flexible data structures must be used to support the creation and the destruction of random *teams*, *programs* and *instructions*.
- *Removing TPG instruction decoding and learning environment data fetching overhead.* Because during training, TPG *programs* contain a dynamic list of *instructions*, referencing operands that can also be mutated during the genetic evolution process, execution of these programs requires a software *instruction* decoding and data fetching mechanisms. Even in their simplest forms, these decoding and fetching mechanisms introduce an indirection overhead during the program execution.
- *Benefiting from compiler optimization.* By translating pretrained TPG into standalone standard C code, the inner working of the TPG and its *programs* are directly exposed to C compilers, making it possible to benefit from their powerful optimization passes.

III. Related Work

The need for fast and low-power AIs is the subject of many studies in the scientific literature [8], [13], [17]. A common way to create lightweight and frugal AIs is to exploit the resilience of DNNs to approximations, using techniques such as pruning, low-precision, and approximate computing [13], [17]. By using such techniques, it is possible to simplify the trained DNN model while retaining most of its accuracy, with *typical* gains up to 1 order of magnitude in terms of computational complexity and 2 orders of magnitude in terms of memory usage [13], [17]. Even with these techniques, only relatively small and less powerful DNNs, such as MobileNet or EfficientNet [16], can be embedded in tight memory-, energy-, and time-constrained systems.

Creating dedicated design flow for inferring pre-trained AI agent has also been explored in the scientific literature. For example, PyTorch, a popular DNN framework, offers the possibility to generate so-called TorchScript, which is a standalone C++ code for inferring the DNN. A similar approach is the use of the ONNX format and associated inference runtime that can be exported from TensorFlow. These techniques considerably reduce the complexity of DNNs inference, especially when batch sizes are small, with *typical* acceleration by a factor 2×

to 10× on various DNN models and hardware [1], [12]. Other C++ code-generation frameworks have also been proposed, notably to remove all dependences of the generated code to third-party library, which is not the case with TorchScript or ONNX [2], or to target specific many-core hardware through the generation of OpenCL code [5].

Although DNN currently dominates the world of machine learning, many other techniques have proven their worth in the past, and continue to be useful today, such as random forest, support vector machines, or principal component analysis. As for DNN, while powerful frameworks exist to train these models, like the scikit-learn Python module, code generation technique have also been created to create inference code of pre-trained models. MicroML Gen, which supports a wide variety of non-DNN model, is an example of framework generating standalone C code for embedded devices with very limited resources, such as ARM Cortex M3 microcontrollers [14].

The TPG model was created half a decade ago, and most scientific publication focus on extending the capabilities of the model [11], [15], but not on accelerating its performance on modern hardware. Hence, available open-source frameworks for TPGs are mostly coded in standard Python, Java, and C++ [7], [10]. This paper introduces the first effort to translate the topology and *programs* of a pre-trained TPG into standalone C code. As in [2], to maximize the portability and lightness of generated code, the generated C code can be compiled without any dependency to third-party library.

IV. Code Generation for TPG Inference

The C language was selected as the target language for the TPG code generation as it is the de facto reference for programming embedded systems. Hence, this choice ensures the portability of generated code on a wide variety of target hardware, ranging for ultra-low power microcontrollers, to high-end CPUs. The following subsections describe how the different parts of a pre-trained TPG are translated into standard C code.

A. Code Generation for Programs

In a TPG, a *program* is a list of *instructions* using data from the learning environment or results of previous *instructions* as operands. As presented in [7], training a TPG with different *instruction* sets results in RL agent with different complexity and fitness. For this reason, it is interesting for TPG frameworks to allow developers to customize the *instruction* set for each training with dedicated instructions.

```
auto addInst = LambdaInstruction<int, int>(
    [](int a, int b)->double {return a + b;},
    "$0 = $1 + $2");
auto accuInst = LambdaInstruction<double[2][1]>(
    [](double[2][1] t)->double {
        return t[0][0] + t[1][0];},
    "$0 = $1[0][0] + $2[1][0]")
```

Listing 1: `LambdaInstruction` usage examples

In a C++ framework, this customization can be supported using lambda functions to specify instructions with their operand number and types, and the lambda function to execute when this instruction is called. Listing 1 presents two examples of C++ code declaring such custom *instuctions* where template arguments define the number and data types of operands accepted by this *instruction*, and the lambda function is the code to execute.

To generate the C code corresponding to a *program*, each *instruction* must be translated into C code. For this reason, as shown in Listing 1, a template string is used when declaring the *instruction*. This template string, which adopts the syntax of regular expressions, uses the `$0` placeholder for the register storing the result returned by the instruction, and the `$n` placeholder for the name of the nth operand of the instruction.

```
1  double P0(int* in0    /* 1D array */,
2            double* in1 /* 4x4 2D-array */)
3  {
4    double reg[8] = { 0 };
5    { /* 1st Instruction: addInst */
6      int op0 = in0[0];
7      int op1 = reg[2];
8      reg[7] = op0 + op1;
9    }
10   { /* 2nd Instruction: accuInst */
11     double[2][1] op0 = {{in1[5]}, {in1[9]}};
12     reg[0] = op0[0][0] + op0[1][0];
13   }
14   ... // Following instructions
15   return reg[0];
16 }
```

Listing 2: Program `P0()` generated code

For each *program* of the TPG, a dedicated C function is printed, as shown in Listing 2. At lines 1-2, the printed function receives as arguments, the pointers to the data sources used to observe the current state of the environment. At lines 4, it declares the registers used to store the results of *instructions* throughout the *program*. Then, at lines 4-14, the *instructions* of the *programs* are printed one by one. For each instruction, the operands are first retrieved from the environment data source. For simple data types, this is achieved through simple pointer dereferencing, as done at line 6-7. For complex operand types, the container class managing the environment data may provide more complex code generation schemes for fetching the operands, as shown at line 11 where a 2D subregion of a 2D array of `double` is extracted automatically. Finally, the value held in the first register is returned as the *bid* for the *program* at line 15.

B. Code Generation of TPG Structure

In this work, we choose to represent the traversal of the TPG graph directly in the generated code, exposing the graph structure to the compiler to perform additional optimizations.

The TPG is encoded as an Finite State Machine (FSM) using a **switch structure**. An extract of the TPG switch structure is represented on Procedure 1. Each `case` represents a *team*, containing *program* executions, as *Team1* at line 5. Transitions

in the FSM represent the edges of the graph. The traversal of *teams* is saved in a specific array initialized at line 2 to avoid executing *programs* multiple times. Scores are set to $-\infty$ (e.g. line 13) to record edge traversals during graph execution in order to avoid falling in an infinite loop. The traversal of a leaf *team*, returns the integer value of the corresponding action, as in line 18.

Procedure 1: ExecuteTPG

```
   Input: Data sources: data
   Output: Action
 1 team = rootTeam
 2 visited[] = { False }
 3 while True do
 4 |  switch team do
 5 |  |  case Team1 do
 6 |  |  |  if !visited[team] then
 7 |  |  |  |  visited[team] = True
 8 |  |  |  |  T1Scores[0] = P0(data)
 9 |  |  |  |  T1Scores[1] = P1(data)
10 |  |  |  |  ... // Outgoing programs of the team
11 |  |  |  end
12 |  |  |  best = bestProgram(T1Scores)
13 |  |  |  T1Scores[best] = -∞
14 |  |  |  team = T1Next[best]
15 |  |  end
16 |  |  ... // Other teams of the TPG
17 |  |  case Action0 do
18 |  |  |  return 1
19 |  |  end
20 |  |  ... // Other actions of the TPG
21 |  end
22 end
```

A **stack based structure** has also been explored and is evaluated in Section V. With the stack structure, each *team* is represented as a function, itself containing an array with all *program* pointers and edges. An execution function called within each team is in charge of executing the *programs* and selecting the next *team* to execute according to the winning *bid*. The *team* calls recursively their next *team*, hence the stack structure. Once reaching an action, the value of the action is returned to the caller. The main limitation of this method is the potential maximum stack size, which can be as large as the number of edges in the TPG.

V. Evaluation of Generated Inference Code

To assess the benefits of the proposed code-generation strategy for inferring TPGs, the contribution was evaluated on a diverse set of computing platforms and learning environments, all presented in Section V-A. Analyses of the results of these experiments are then discussed in Section V-B. To ensure the reproducibility of the presented results, all library code, pre-trained TPGs, execution traces and analysis code is made available as open-source artifacts [6].

A. Experimental Setup

1) Training and Learning Environments:

The proposed code generation strategy has been implemented as part of the open-source C++ GEGELATI framework for TPGs [7]. Generic Evolvable Graphs for Efficient Learning of Artificial Tangled Intelligence (GEGELATI) was proven to give similar performance, on a single CPU core, for the training and inference of TPGs as the reference C++ code from Kelly [9]. Version `1.1.0` of GEGELATI was used for all experiments described in this paper, notably to provide the reference time for inference without code generation.

The Arcade Learning Environment (ALE) (version `0.6.1+d3f2b25`) was chosen as the learning environment for training TPG [3]. The Arcade Learning Environment (ALE) is a collection of 55 Atari 2600 video games, with diverse complexity. In these experiments, 5 games with diverse degrees of difficulty were chosen to compare assess the performance of generated code: `alien`, `asteroids`, `centipede`, `fishing_derby`, and `frostbite`.

For the following experiments, 10 trainings with different seeds for the pseudo-random evolution process were run for each of the 5 selected games. Each of the 50 TPG graphs was trained during 400 generations, using the meta-parameters described in [9]. The characteristics of the trained TPGs are summarized in Table I. These TPGs cover a wide range of use cases, with the smallest TPG consisting of only 3 *teams* and 8 *programs*, and the largest one consisting of 25 *teams* and 50 *programs*. It is important to note that all *teams* that were never visited during a game, and all *programs* that never produced a winning *bid* were all removed from the TPG at the end of the training process.

2) Platforms:

The performance of the 50 pre-trained TPGs has been evaluated on 4 different platforms, including both low-end, embedded and high-end CPUs. Characteristics of the 4 CPUs are summarized in Table II.

Id	Platform	Core type	Frequency	GCC
`rpi2`	Raspberry Pi 2	ARM A7 32bit	0.9 GHz	v11.2.0
`jetson`	Jetson Nano TX2	ARM A57 64bit	1.2 GHz	v7.5.0
`laptop`	i7-8650U	Intel x64	1.9 GHz	v9.3.0
`xeon`	Xeon E5-2690	Intel x64	2.9 GHz	v7.5.0

TABLE II: Characteristics of the 4 target CPUs.

B. Experimental Results

On each target platform, the inference time of each of the 50 TPGs summarized in Table I was measured 5 times with GEGELATI, 5 times with switch-based code generation inference, and 5 times with the stack-based code generation. All measures were done using standard C++ `std::chrono::high_resolution_clock` timer. For each {game, platform, TPG, inference mode} configuration, only the average inference time is kept, as the observed relative standard deviation for the 5 runs of each configuration are very low, with an average of 0.06% with the library, and 0.004% with generated code.

Game	Teams	Programs	Instr. / prog.	Actions / game	Teams / action	Prog. / action
alien	15.40 ± 30%	34.10 ± 23%	4.73 ± 46%	4747.60 ± 22%	6.38 ± 18%	19.78 ± 19%
asteroids	12.20 ± 27%	31.70 ± 21%	4.81 ± 19%	11848.60 ± 25%	5.91 ± 28%	20.99 ± 26%
centipede	14.00 ± 42%	28.70 ± 37%	4.80 ± 18%	17992.00 ± 0%	6.03 ± 41%	18.02 ± 37%
fishing_derby	9.70 ± 58%	20.80 ± 57%	4.14 ± 17%	8477.00 ± 11%	5.00 ± 47%	12.87 ± 35%
frostbite	9.90 ± 39%	24.20 ± 32%	4.56 ± 34%	8534.60 ± 41%	4.87 ± 39%	13.74 ± 30%

TABLE I: Characteristics of TPGs used in experiments. For each characteristic (columns), the table contains the average value and the relative standard deviation for the 10 TPGs trained on each game (rows).

When measuring the inference time, only the time spent executing the TPG is measured. The time taken to compute the evolution of the environment state following an action of the TPG is not part of the measured inference time. Although the time taken to parse the TPG file and build the TPG into the library could also be measured, and compared with the time needed to load the compiled generated code, this time strongly depends on the chosen file format which, in our case, would be unfairly to the detriment of the library.

1) Stack- vs Switch-based code generation: To assess which of the switch- and stack-based code generation produces the best performance, the measured inference time were compared for the 50 TPG inferred on all platforms. Figure 3 presents the speedup in inference time of the switch-based code generation with respect to the stack-based code generation. On average, inference time with the switch-based code generation is 8.87% faster than with the stack-based structure, and is faster in 95% of the configurations. This result can be explained by the function calls and stack memory usage overheads of the stack-based structure. Moreover, the switch-based structure is compiled to a jump table, creating an efficient code for the TPG inference execution.

In the reminder of this section, all code generation results are obtained with the switch-based code generation.

2) Inference performance with generated code and library: The speedups in inference time of the generated switch-based code with respect to the library is presented in Figure 4. They are based on the total TPG inference time during a game, and are presented as a ratio between the library and generated code. On average on all games, the observed speedups are 44× on xeon, 24× on laptop, 45× on jetson, and 85× on rpi2. There are many possible causes to these difference in average speedup between platforms, notably: different hardware complexity (in-order vs out-of-order pipeline, bitwidth, instruction & data cache sizes), different compiler versions, etc. Nevertheless, the obtained results are very good, especially for the rpi2 which is the most lightweight CPU and benefits the most from the acceleration brought by the code generation.

Interestingly, the per-game variations of the speedups observed on every platform are very similar. For example, speedups obtained on all platforms for the asteroids game are on average 56% larger than speedups for the fishing_derby game. This results seems to indicate that the measured speedup per-TPG depends on the intrinsic complexity of the TPG itself, which derives from its number of *teams* or *programs*.

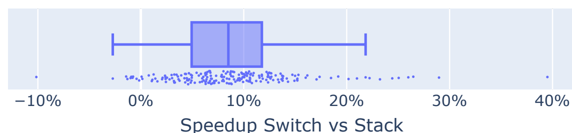


Fig. 3: Speedup in inference time of the switch-based generated code with respect to the stack-based generated code. The box-plot represents the statistics obtained over 200 speedups (5 games×4 platforms×10 TPGs).

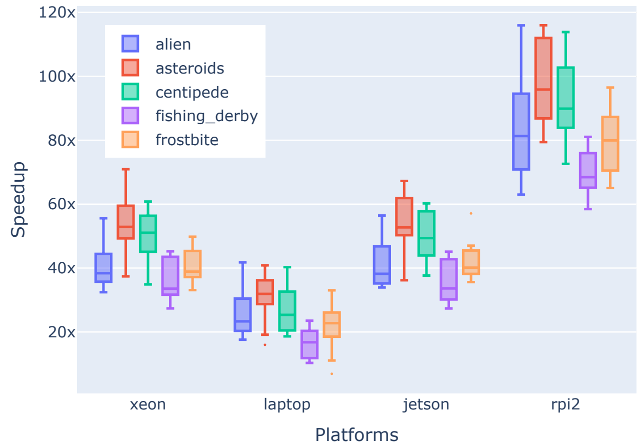


Fig. 4: Speedup in inference time of the generated code with respect to the library. Each box-plot represents the statistics for the 10 TPGs trained for a given game, and inferred on a specific platform.

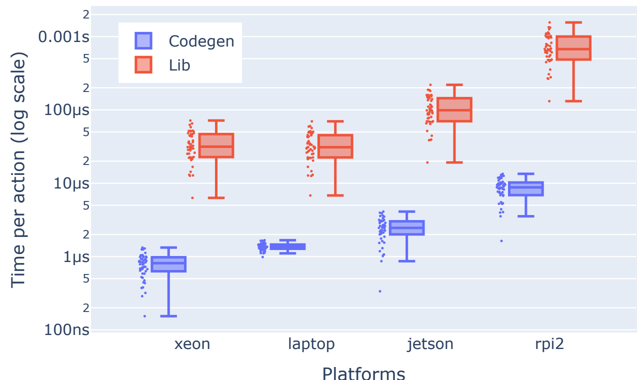


Fig. 5: Average time per TPG inference with the generated code and with the library. Each box-plot represents the statistics for the 50 TPGs (5 games×10 TPGs) run on a platform.

The average times per inference of the TPG, that is per action, for the different platforms are presented in Figure 5. These results show the impressive absolute performance of the generated code, which on average performs an inference of the TPG within 782ns on `xeon`, 1.36µs on `laptop`, 2.41µs on `jetson`, and 8.60µs on `rpi2`. While these result confirm the benefit of using generated code for infering TPGs, they also reveal the important spread of inference time on most platforms, with an average relative standard deviation of 34% for generated code (excluding `laptop`), and 42% for inference within the library.

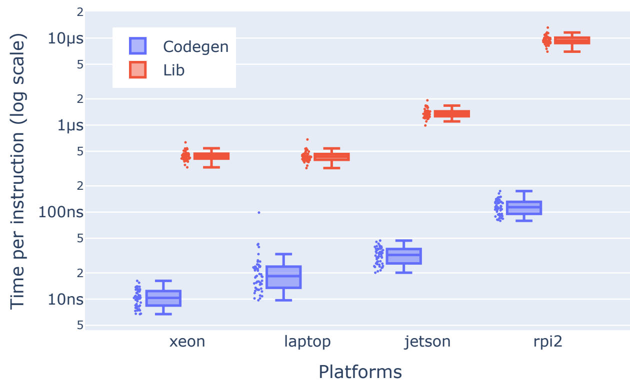


Fig. 6: Average time per instruction execution with the generated code and with the library. Each box-plot represents the statistics for the 50 TPGs (5 games×10 TPGs) run on a platform.

Figure 6 depicts the average execution time taken per line of *programs* of the TPG on the different platforms. With the library, these results show that the inference time of a TPG strongly correlates with the number of lines of *program* to execute, with a relative standard deviation of only 12%. Except on the `laptop` platform, for yet unknown reasons, these results also reveal the correlation between the average number of *program* lines executed and the inference time with generated code, with a relative standard deviation of 21%. The larger spread of per-line execution time for generated code can be explain by the finer granularity of the measured time, and the stronger dependency on the nature of executed instructions and potential compiler optimizations. Indeed, with the library, the time spent decoding instructions and fetching operands in software most likely dominates the time actually spent executing a program line, which hides the complexity difference between different instructions.

VI. Conclusion & Future Work

This paper introduces a code generation workflow to accelerate the inference of pre-trained Reinforcement Learnings (RLs) agent based on the Tangled Program Graph (TPG) model. Acceleration of the TPG inference is achieved by getting rid of the algorithmic and software overhead that is unavoidable for training TPGs but dependable when focusing on TPG inference. Experiments on four computing platforms, ranging from embedded processors to high-end CPUs, result in an acceleration by a factor 50, on average, of the inference time, compared to a traditional framework. For a state-of-the-art visual RL environment and for performance equivalent to DNNs, the obtained inference times range from 782ns to 8.60µs on single-core CPUs, making this approach a very promising one for embedding RL agent in ultra-low power edge AI systems. Potential directions for future work includes the generation of code matching the instruction set of the target computing platform in order to further improve the complexity-performance tradeoff offered by this technique.

References

[1] M. Alluin, "An empirical approach to speedup your bert inference with onnx/torchscript," Feb. 2021. [Online]. Available: https://towardsdatascience.com/an-empirical-approach-to-speedup-your-bert-inference-with-onnx-torchscript-91da336b3a41

[2] S. An and L. Moneta, "C++ code generation for fast inference of deep learning models in root/tmva," in *EPJ Web of Conferences*, vol. 251. EDP Sciences, 2021, p. 03040.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *CoRR*, 2012. [Online]. Available: http://arxiv.org/abs/1207.4708

[4] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint*, 2016. [Online]. Available: https://arxiv.org/pdf/1605.07678.pdf

[5] B. de Dinechin, J. Hascoët, J. Le Maire, and N. Brunie, "Deep learning inference on the mppa3 manycore processor," in *Embedded World Conference 2020*, 2020.

[6] K. Desnos, T. Bourgoin, N. Sourbier, M. Dardaillon, O. Gesny, and M. Pelcat, "Sips22 artifacts," Nov. 2022. [Online]. Available: https://github.com/gegelati/SiPS22-Artifacts

[7] K. Desnos, N. Sourbier, P.-Y. Raumer, O. Gesny, and M. Pelcat, "Gegelati: Lightweight artificial intelligence through generic and evolvable tangled program graphs," in *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*, ser. International Conference Proceedings Series (ICPS). Budapest, Hungary: ACM, 2021.

[8] J. Ichnowski, Y. Avigal, V. Satish, and K. Goldberg, "Deep learning can accelerate grasp-optimized motion planning," *Science Robotics*, vol. 5, no. 48, 2020.

[9] S. Kelly, "Scaling genetic programming to challenging reinforcement tasks through emergent modularity," Ph.D. dissertation, Dalhousie University, Halifax, Nova Scotia, Canada, 2018.

[10] S. Kelly and M. I. Heywood, "Emergent tangled graph representations for atari game playing agents," in *European Conference on Genetic Programming*. Springer, 2017, pp. 64–79. [Online]. Available: https://web.cs.dal.ca/ mheywood/OpenAccess/open-kelly17a.pdf

[11] S. Kelly, J. Newsted, W. Banzhaf, and C. Gondro, "A modular memory framework for time series prediction," in *Genetic and Evolutionary Computation Conference*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 949–957. [Online]. Available: https://doi.org/10.1145/3377930.3390216

[12] M. Levental and E. Orlova, "Comparing the costs of abstraction for dl frameworks," *arXiv preprint arXiv:2012.07163*, 2020.

[13] K. Ota, M. S. Dao, V. Mezaris, and F. G. D. Natale, "Deep learning for mobile multimedia: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 3s, pp. 1–22, 2017.

[14] S. Salerno, "Introducing microml generator," Online, Nov. 2019. [Online]. Available: https://eloquentarduino.github.io/2019/11/you-can-run-machine-learning-on-arduino/

[15] R. J. Smith, R. Amaral, and M. I. Heywood, "Evolving simple solutions to the cifar-10 benchmark using tangled program graphs," in *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 2061–2068.

[16] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*. PMLR, 2019, pp. 6105–6114.

[17] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating deep convolutional networks using low-precision and sparsity," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2017, pp. 2861–2865.