Reachability Graph of IOPT Petri Net Models Using CUDA C++ Parallel Application

Carolina Lagartinho-Oliveira[1(✉)], Filipe Moutinho[1,2], and Luís Gomes[1,2]

[1] Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Caparica, Portugal
ci.oliveira@campus.fct.unl.pt, {fcm,lugo}@fct.unl.pt
[2] UNINOVA – CTS, Caparica, Portugal

Abstract. The construction of reachability graphs is suited to verify the properties and behavior of Petri net models based on the structure of the net and the initial marking. It allows checking whether a model conforms to the intended specification of a system and to obtain information about it. This paper proposes an algorithm to compute the reachability graphs of IOPT (Input-Output Place-Transition) nets, which is a Petri net class, using NVIDIA's CUDA (Compute Unified Device Architecture), which supports the co-processing using GPU and CPU. While CPU is used to schedule threads on GPU, GPU is used to calculate all the child nodes of the reachability graph, including the management of a hash-table for efficiently storing the new states and retrieving the states stored in the database. The presented algorithm takes advantage of CUDA memory functions to allocate and access data that can be used by code running on CPU or GPU, supporting the share of data between the two processor units. Six IOPT net models were used to validate the proposed algorithm.

Keywords: Co-processing · CUDA · GPU · IOPT nets · Reachability graph

1 Introduction

The increasing complexity of distributed embedded systems have been a motivation for the use of Petri nets [1, 2]. This graphical modelling formalism enables the explicit specification of concurrent systems, their synchronization and conflicts, and the share of resources [3]. As a result, they ensure that systems behavior conforms to the intended specification so as not to endanger people.

IOPT-Tools, which are online available at http://gres.uninova.pt/IOPT-Tools/ supports the development of embedded systems controller using Petri nets [4, 5]. This framework offers a set of tools to support the creation of IOPT-net models [1], their verification, and the automatic code generation (C and VHDL) [6, 7]. To enable the verification, an automatic code generator is used to compute the models' reachability graphs [8]. As real world applications can present exponential reachability graphs with a huge number of states, their generation can take a long time to compute [2], requiring high computational performance [9]. Some tools generate the condensed reachability graph of a Petri net model, preventing the exponential growth of the graph [10–12].

Published by Springer Nature Switzerland AG 2020
L. M. Camarinha-Matos et al. (Eds.): DoCEIS 2020, IFIP AICT 577, pp. 93–100, 2020.
https://doi.org/10.1007/978-3-030-45124-0_8

The work presented in this paper is focused on obtain the complete reachability graph of a model, based on a new model-checking algorithm to compute the reachability graphs of IOPT Petri net models using NVIDIA's CUDA. NVIDIA's CUDA supports CPU-GPU co-processing for parallel computing [13]. As a matter of fact, GPU calculates all the child nodes of the reachability graph and handles their storing with the support of a hash-table. In addition, the CPU schedules the threads to be launched on the GPU, which are needed to process a new set of unprocessed states of the graph.

Section 2 mentions how this paper contributes to life improvement. In Sect. 3 IOPT Petri nets and IOPT-Tools are briefly described, including the current reachability graph generation tool. In the following section, it is mentioned how CUDA Toolkit contributes to program and run parallel C++ applications on GPU. Section 5 presents the proposed algorithm, for the computation of reachability graphs, for IOPT Petri net models. In Sect. 6 are presented the results supported by an NVIDIA Titan V GPU, and finally in Sect. 7 the conclusions about the results and future work are presented.

2 Relationship to Life Improvement

Currently, technological advances enabled the creation of many types of distributed embedded systems that contribute to a significant improvement in people's quality of life across different areas, ranging from appliances and home products, medical and health solutions, surveillance and security equipment, to transportation and communication systems, among others.

Concerning to safety-critical systems, to make sure that they are safe and free of development errors, they must be formally verified, by checking all possible interactions and potential unwanted properties [14–16]. There are large number of Petri net tools [17], which support not only the models edition and simulation, but also their formal verification and analysis, to ensure that the system specification conforms to the desired properties or has no unwanted properties. This work, addressing the construction of reachability graph for IOPT nets, aims to contribute for the safety-critical systems properties verification.

3 IOPT Nets

The IOPT nets are a low-level and non-autonomous Petri net class proposed to develop automation and embedded systems, allowing the rapid prototyping of system controllers through IOPT-Tools framework [18–20]. The IOPT Petri net relies on signals and events to specify the interaction of the models with the environment: while input signals and events constraint the evolution of the net, directly associated with transition firing, outputs are updated according with the marking of the net and transition firing. In detail, a transition fires if it is enabled from the point of view of place marking, the associated events occur and if the associated guards are verified. When more than one transition is enabled, but not all of them are allowed to firing, transition priorities and

test arcs can be used [8]. As a result, each system's state is composed of a vector of all places' marking; and an output event signal vector, with the values of all signals associated with output events [19].

3.1 Reachability Graph Generator

The IOPT-Tools offer a tool to compute the reachability graphs [19] of IOPT Petri net models. During graph generation, the tool gathers information about the model, namely the influence of input signals and events on the firing of transitions, as well as all combinations of all enabled transitions. The automatic C code generated by the reachability graph generator provides libraries to compute a model's reachability graph, managed with a hash-table that allows ordering multiple states for each key, to help search for repeated states. At the end of the computation, the resulting reachability graph is stored in a hierarchical XML file, in which the connections between the states are represented.

The reachability graph algorithm [21] initiates with the creation of the database and initial state, obtained from the initial marking M_0 and the values of all output event signals presented on the net at that moment. After that, the initial node is added into the database and hash-table, and the algorithm proceeds from there or any other state by modifying the value of net's initial marking. The algorithm will continue until all the unprocessed states are treated, or the graph reaches the maximum size, which is specified according to the computation platform available resources. Each evaluation of an unprocessed state is carried out with the calculation of its child states (the next unprocessed states), by executing a function that recursively analyzes all transitions that are enabled to firing. Then, all child nodes are stored in the database and sorted in the hash-table if they are not repeated states, whose marking and outputs refer to previously existing nodes. Finally, the new child nodes are added to the set of unprocessed states, waiting to be processed. The generated reachability graph includes the nodes, the arcs that connect them, and links that represent existing nodes.

4 CUDA Architecture

To increase the performance of the IOPT reachability graphs generation, the use of GPU is proposed in this paper. The GPU is used to improve the processing of each state, parallelizing the calculation and analysis of its child nodes. For that, it was used a NVIDIA GPU and the CUDA Toolkit, which enable the co-processing of C++ programs in platforms with CPU and GPU [22, 23]. The execution starts and ends in the CPU, which exchanges data with the GPU and launch the kernels to running on the GPU.

A kernel is a sequential program that runs in parallel as many times as the number of threads running on the GPU distributed by blocks in a grid. Although, there is a limit to the number of threads per block, a kernel can be executed by multiple blocks in parallel, so the total number of threads launched is equal to the number of threads per block times the number of blocks thar compose the grid. A set of functions, such as __threadfence, __syncthreads, and atomic operations, can be used to ensure the correct

access to shared and global memory, avoiding hazards that can occur from simultaneous read and write operations at the same memory address [24].

5 Proposed Approach and Algorithm

The reachability graph generation at IOPT-Tools is mainly composed of two parts: for each state, it is calculated its child nodes; and for each child node it is inspected if it is a new independent state or if it is equal to a previously existing one. In the generator of the IOPT tool framework, this entire process is done sequentially in CPU, with each state being analyzed one at a time. An algorithm proposed in [25] reused part of IOPT-Tools generator code and adapted it to run on a GPU used to perform the calculation all the child nodes of unprocessed states in parallel, while the CPU schedules threads on the GPU, handles the hash-table and the categorization of states.

The algorithm proposed in this paper also uses co-processing, taking even more advantage of the GPU. For that, the initialization of the database and the creation of the initial state are handled by the CPU, and from that moment onwards the task of searching for new states it's responsibility of the GPU until all graph it's done. During the computation, the CPU receives feedback on the status of the graph through memory copies from the device to the host, receiving the update of the number of calculated states and the number of states to be processed. This allows the continuity of the algorithm that ends only when there are no more unprocessed states to process or the maximum number of states associated with the allocation of the database is reached. So, three kernels were implemented, each one responsible for a specific part of the algorithm, as described in Fig. 1.

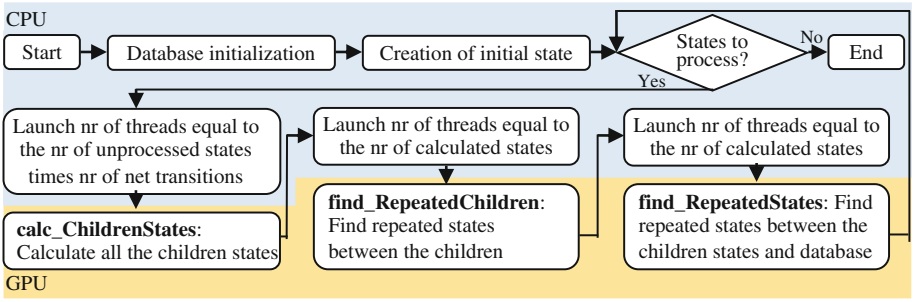

Fig. 1. Sequence of actions of the algorithm.

When the kernel calc_ChildrenStates, presented in Algorithm 1, is invoked, the values of the number of states to be processed and the current number of states stored in the database are passed. The first instruction performed is the assignment of an unprocessed state from the array states to the threads of a block. At this point, CPU has been launched as many blocks as the number of unprocessed states that need to be processed, and as many threads as the number of transitions of the Petri net. The way to

provide each thread the state to process is using the unique index of the block, the size of the array states, that stores all the states, and the number of blocks launched.

Algorithm 1. Kernel calc_ChildrenStates

```
program calc_ChildrenStates(unp_states, n_states)
  __shared__ state s*
  netMarking init_m, m, avail_m
  eventOutputSignals init_out, out
  outputSignalEvents ev
  begin
    IF threadIdx.x == 0
      s = &states[blockIdx.x+n_states-gridDim.x]
    __syncthreads()
    calc_FiringBlends(threadIdx.x, init_m, m, avail_m, unp_states,
      n_states, s->id, ev, out, init_out)
end.
```

Using the __syncthreads() function all the threads in the current block will synchronize, waiting for thread 0 to share the state that needs to be processed. The kernel continues with a recursively analysis of all enabled transitions of a state, calculating all the combinations between them. The transition with the highest priority will analyze the remaining ones; the second most priority will analyze the other ones except the first one, and so on. The latter transition will only consider itself. As there is one thread per transition, each one of them will analyze the combinations in parallel for each state, saving the founded new states inside the array childs.

After that, the kernel returns the number of child nodes calculated to proceed with the search of repeated children. The CPU launches a number of threads equal to the number of founded states, one for each thread to compare with the others. The comparison of the states is made at find_RepeatedChildren kernel, presented in Algorithm 2, by comparing the marking and outputs of the Petri net. If there is a repeated sibling, the value of the link flag and the id of the state are changed, followed by the storage of the state in the array links.

Algorithm 2. Kernel find_RepeatedChildren

```
program find_RepeatedChildren(n_links)
  For i=0; i<threadIdx; i++:
    int cmp = memcmp(childs[threadIdx.x].m, childs[j].m)
    IF cmp == 0:
      cmp = memcmp(childs[threadIdx.x].o, childs[j].o)
        IF cmp == 0:
          childs[threadIdx.x].link = -1
          childs[threadIdx.x].id = dev_childs[j].id
          memcpy(links[atomicAdd(n_links,1)], childs[threadIdx.x])
end.
```

The last kernel, presented in Algorithm 3, compare the child nodes that have not been copied to the array links with all existing nodes. For that, it was implemented a multivalue hash-table where multiple values for the same key are represented by

different key-state_id pairs. In the same way as the previous kernel, the CPU launches a number of threads equal to the number of founded states. Each thread will search on hash-table for repeated states starting by calculating the key based on the marking and outputs of the state, using it to limit the search to elements with the same key. Several threads could access the hash-table, including threads whose states have the same key. If there is no repeated state at the database, the value of the id of the child state is actualized, and the state is stored in the array states; if there is a repeated state it is stored in the array links.

Algorithm 3. Kernel find_RepeatedStates

```
program find_RepeatedStates(unp_states, n_states, n_links)
  IF childs[threadIdx.x].link != -1:
    key = calcHash(childs[threadIdx.x].m, childs[threadIdx.x].o)
    p = findHash(key, childs[threadIdx.x].m, childs[threadIdx.x].o)
    IF p < 0:
      state_id = atomicAdd(n_states,1)
      addHash(h, state_id, -p)
      childs[threadIdx.x].id = state_id
      memcpy(states[state_id], childs[threadIdx.x])
    Else:
      childs[threadIdx.x].id = p
      memcpy(links[atomicAdd(n_links,1)], childs[threadIdx.x])
end.
```

6 Results of Experiments

The algorithm presented was applied to six IOPT-net models. These models were also used to document the results at [25], and are available online at http://gres.uninova.pt/IOPT-Tools/, in the user account "models". Using these models, we could compare the results between the two approaches. The results are presented in Table 1. Considered the amount of time that GPU took to calculate the entire reachability graph and the time spent on co-processing we have obtained much better results with this algorithm. The results are, in some cases, two or three orders of magnitude better than the ones presented at [25].

Table 1. Results obtained with an GPU TITAN V.

Models	Trans.	Cycles	States	Links	Time CPU + GPU(ms)	Time on GPU(ms)
ICIT13_bldc_commut	24	2	7	0	0.8	0.6
PNSE-53b	8	9	21	8	2.3	1.8
ICIT13_denoise	14	9	14	11	2.5	2.0
concrete_mixer_6xA	11	56	110	108	17.5	14.7
ICIT13_quad_encoder	12	103	1025	1020	67.3	59.7
ICIT13_pwm_gen	6	1025	4096	12287	655.5	561.7

The number of cycles and nodes presented express the graph with the respect to its size and format. This characteristic can affect the execution time depending on the number of states processed in parallel as well as the number of transitions of the net. The models PNSE−53b and ICIT13_denoise presented the same number of cycles processed; for the first was calculated 29 nodes and for the second 25. Although the number of transitions in the second is almost twice that of the first, the execution times obtained were approximately equal, such that parallel threads were launched and used at the same time to exploit the computing power of the GPU.

7 Conclusions and Future Work

The proposed algorithm presents an improvement in the computation of the reachability graph for IOPT Petri net models in GPU. Threads were used to analyze combinations of transitions in parallel, improving the calculation of child states; and the use of a hash-table implemented in GPU prevented the time spent with memory management. As future work we intend to analyze the impact of the proposed algorithm using GPU as a function of the number of global states obtained, namely as a function of the initial marking. Additionally, the impact of computation efforts necessary for evaluating transition enabling conditions and output expressions is intended to be analyzed.

Acknowledgments. The work presented in this paper was partially supported by Portuguese Agency FCT ("Fundação para a Ciência e a Tecnologia"), in the framework of the project with the reference UID/EEA/00066/2019 and UIDB/00066/2020 (CTS – Center of Technology and Systems). We would also like to thank NVIDIA Corporation for the donation of the GPU used in this work, a Titan V.

References

1. Gomes, L., Barros, J., Costa, A., Nunes, R.: The Input-Output Place-Transition Petri net class and associated tools. In: Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Vienna, Austria, July 2007
2. Girault, C., Valk, R. (eds.): Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-05324-9
3. David, R., Alla, H. (eds.): Discrete, Continuous, and Hybrid Petri Nets. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-10669-9
4. Pereira, F., Moutinho, F., Gomes, L.: IOPT-Tools—towards cloud design automation of digital controllers with Petri nets. In: Proceedings of the 2014 International Conference on Mechatronics and Control (ICMC 2014), Jinzhou, China, July 2014
5. Gomes, L., Moutinho, F., Pereira, F.: IOPT-Tools—a web based tool framework for embedded systems controller development using Petri nets. In: Proceedings of the 23rd International Conference on Field Programmable Logic and Applications, Portugal (2013)
6. Pereira, F., Gomes, L.: Automatic synthesis of VHDL hardware components from IOPT Petri net models. In: IECON 2013 – 39th Annual Conference of the IEEE Industrial Electronics Society (IECON 2013), Vienna, Austria, November 2013

7. Gomes, L., Rebelo, R., Barros, J., Costa, A., Pais, R.: From Petri net models to C implementation of digital controllers. In: Proceedings of the ISIE 2010 - IEEE International Symposium on Industrial Electronics, Bari, Italy, July 2010
8. Pereira, F., Moutinho, F., Gomes, L.: A state-space based model-checking framework for embedded system controllers specified using IOPT Petri nets. In: Camarinha-Matos, L.M., Shahamatnia, E., Nunes, G. (eds.) DoCEIS 2012. IAICT, vol. 372, pp. 123–132. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28255-3_14
9. Pereira, F., Moutinho, F., Gomes, L., Rebelo, R.: IOPT Petri net state space generation algorithm with maximal-step execution semantics. In: Proceedings of the 2011 9th IEEE International Conference on Industrial Informatics, Caparica, Lisbon, July 2011
10. Jensen, K.: Condensed state spaces for symmetrical Coloured Petri Nets. J. Form. Method Syst. Des. **9**(1), 4–40 (1996)
11. Christensen, S., Kristensen, L.M., Mailund, T.: Condensed state spaces for timed Petri Nets. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 101–120. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45740-2_8
12. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
13. Nickolls, J., Dally, W.J.: The GPU computing Era. IEEE Micro **30**(2), 56–69 (2010)
14. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, USA, May 2002
15. Hsiung, P., Chen, Y., Lin, Y.: Model checking safety-critical systems using safecharts. IEEE Trans. Comput. **56**(5), 692–705 (2007)
16. Moutinho, F., Gomes, L.: Distributed Embedded Controller Development with Petri Nets. SECE, vol. 150. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-20822-0
17. Petri Nets Tool Database. https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html
18. Gomes, L., Costa, A., Barros, J.P., Lima, P.: From Petri net models to VHDL implementation of digital controllers. In: IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society, Taipei, Taiwan, November 2007
19. Gomes, L., Lourenco, J.: Rapid prototyping of graphical user interfaces for Petri-net-based controllers. IEEE Trans. Industr. Electron. **57**(5), 1806–1813 (2010)
20. Pereira, F., Moutinho, F., Ribeiro, R., Gomes, L.: Web based IOPT Petri net Editor with an extensible plugin architecture to support generic net operations. In: IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society, Canada, December 2012
21. Moutinho, F., Gomes, L.: State space generation algorithm for GALS systems modeled by IOPT Petri nets. In: IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society, Melbourne, VIC, Australia, November 2011
22. Parande, J.G., Kulkarni, M., Bawaskar, A.: GPGPU processing in CUDA architecture. Adv. Comput. Int. J. **3**(1), 105–120 (2012)
23. Jin, H., Li, B., Zheng, R., Zhang, Q., Ao, W.: *memCUDA*: map device memory to host memory on GPGPU platform. In: Ding, C., Shao, Z., Zheng, R. (eds.) NPC 2010. LNCS, vol. 6289, pp. 299–313. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15672-4_26
24. Cuda Toolkit Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/
25. Lagartinho-Oliveira, C., Moutinho, F., Gomes, L.: GPGPU applied to support the construction of the state-space graphs of IOPT Petri net model. In: IECON 2019 - 45th Annual Conference on IEEE Industrial Electronics Society, Lisbon, Portugal, October 2019