Introduction of an OpenCL-Based Model Transformation Engine

Tamás Fekete(✉) and Gergely Mezei

Budapest University of Technology and Economics, Budapest, Hungary
{fekete,gmezei}@aut.bme.hu

Abstract. As model-driven engineering (MDE) became a popular software development methodology, several tools are built to support working with MDE. Nowadays, the importance of performance is getting higher as the size of the systems grow. New solutions are needed that can take advantage of modern hardware components and architectures. One step towards this goal is to use the unique processing power of GPUs in model-driven environments. Our overall goal is to create a graph transformation framework that fits into the parallel execution environment provided by GPUs. Our approach is based on the OpenCL framework and it is referred to as PaMMTE (Parallel Multiplatform Model-transformation Engine). This paper presents an overview of our tool and the description of the implementation. We believe that this new approach will be an attractive way to accelerate MDE tools efficiently.

Keywords: MDE · GPU · Graph transformation
High-performance computation · Parallel computation · OpenCL

1 Introduction

Model-driven engineering (MDE) can simplify the software development processes that caused the sudden spreading of its usage in various domains. MDE works with models that are no longer created only for presentation purposes but transformed, processed and often used directly or indirectly as the basis of code generation. Hence, it is an important and challenging part of MDE to find and apply suitable model transformation techniques. The graph rewriting-based model transformation (or graph transformation for the sake of simplicity) is one of the most popular among them [1]. Besides CPU, there are other hardware components to accelerate the execution of these algorithms. The advantages of platform independence are obvious here, since the hardware available to the users is quite heterogeneous. In order to handle this, the OpenCL framework[1] is used in our approach. OpenCL is platform independent and can be used to handle the most widely used hardware components uniformly (CPU, GPU, FPGA, DSP). In this paper, we show that by using an OpenCL-based solution, a promising way

[1] https://www.khronos.org/opencl.

© Springer International Publishing AG 2018
M. Seidl and S. Zschaler (Eds.): STAF 2017 Workshops, LNCS 10748, pp. 14–19, 2018.
https://doi.org/10.1007/978-3-319-74730-9_2

to accelerate model transformations efficiently can be found. Although OpenCL is rarely used in MDE tools, we provide reasons why OpenCL is moving forward in a promising direction. We also introduce the base architecture and model transformation logic of our tool.

2 Related Work

Paper [2] studies the most widely used MDE tools: GREAT, IncQuery, Fujaba, Groove, Henshin, MOLA, Viatra2 in order to understand them. Although the performance is not the most important property, these tools manage model transformations efficiently. For example, in [3] IncQuery uses the so-called incremental evaluation of queries to accelerate. Moreover, there are tools (like GMTE[2]) that use a C++ implementation to achieve better performance. However, none of the existing tools can efficiently use the benefit of the parallel execution architecture offered by the GPUs. OpenCL is a popular way to use the computation power of GPUs, FPGAs and many other devices. The key aspect to achieve high-performance computation is to apply appropriate scalability techniques. Improving the scalability in different contexts is an actively researched area as seen in [4]. Papers [5,6] showed that OpenCL can be efficiently used with graphs. However, mapping graph algorithms from CPU version to OpenCL is a significant challenge. In [7], the k-Nearest Neighbor is implemented using the multi-GPU OpenCL. We should mention at this point that the paper also provides a CUDA-based implementation. CUDA is another major GPU programming platform, however, it is strongly hardware-dependent while OpenCL is not. The measurements in this paper show that the efficiency of the two platforms varies. Taking everything into account, we choose to continue working with OpenCL mainly because of its platform independence. Paper [8] shows the usage of the OpenCL with some C++ and STL related features as part of the official Boost.

3 Parallel Multiplatform Model-Transformation Engine

In this section, we introduce our solution: the Parallel Multiplatform Model-Transformation Engine (PaMMTE)[3]. PaMMTE is implemented in C++ 14 to maximize performance. We should note that the approach is currently limited to execute the graph transformation rules separately; no control flow support is given. However, specifying a pivot node where the match should be started can help matching.

3.1 The Representation of the Domain Model

At the beginning of the model transformation, the input domain model is read and converted using a domain specific adapter. Using the adapter, we split the

[2] http://homepages.laas.fr/khalil/GMTE/.
[3] https://www.aut.bme.hu/Pages/Research/VMTS/PaMMTE.

input model into two sets of data. The first set is a graph representing the topology of the original model, while the second set represents the attributes attached to the model entities. During the transformation, these representations are used and the changes are evaluated on the original domain model as the last step of the transformation. Although we need to create an adapter for each domain, we provide a template to simplify the task. In the topology graph, all elements are represented by an elementID and a typeID. The elementID is a unique identifier of the node generated by the adapter creating the graph representation. Type information on domain elements is expressed by the typeID that contains the unique identifier of the type (metaelement) of the given element. Both elementID and typeID are integer values in order to accelerate their use on the GPU. In case of attributes, we create an array of data referring to the container entity by using its elementID. From a technical point of view, we use a hash table to build the graph and create the inner topology/attribute representation from the input domain model. The main benefit of using a hash table is its ability to find entities quickly (in O(1) time). Practically speaking, matching requires several orders of magnitude more time than rewriting. Therefore, the costly operation of modifying the hash table has no serious affect on performance. The graphs are further processed by the host (the CPU) just before working with it on the GPU. This transformation is not complex, however, it is advantageous in order to simplify and accelerate the algorithms running on the GPU. The original graph is mapped into two one-dimensional structures using the elementIDs of the nodes: (i) The first structure contains the list of the neighbors one-by-one from the first to the last node. (ii) The second structure contains the starting positions of the neighbor list and is a helper structure to process the first. Using these two arrays and the size of the second array, all graphs can be passed to the OpenCL device. The structure of attributes is much simpler in that arrays refer to their container entity by using elementIDs.

3.2 Steps of the Approach

Unlike most of the tools in our approach, the execution of graph transformation rules is divided into three major logical steps (Fig. 1): (i) pattern matching, (ii) attribute processing and (iii) graph rewriting. The three logical steps are connected to each other and are executed sequentially. (i) Pattern-matching is responsible for searching for topological matches according to the user defined rewriting rules. In this step, only the aforementioned topology graph is used. (ii) Attribute processing works on the result of the first step and it filters the matching candidates by evaluating attribute constraints on them, which are evaluated separately and sequentially. If a certain constraint fails, the candidate is dropped. We have created several dedicated kernels for the most typical constraint types (e.g., regular expressions in strings, simple numeric operations, etc.). In addition to these dedicated kernels, we support using custom atypical constraints, however, they must be specified in OpenCL. To simplify this task, we are continuously working on extending the range of built-in constraints. At this point, domain attributes are also needed; thus, attribute arrays are copied

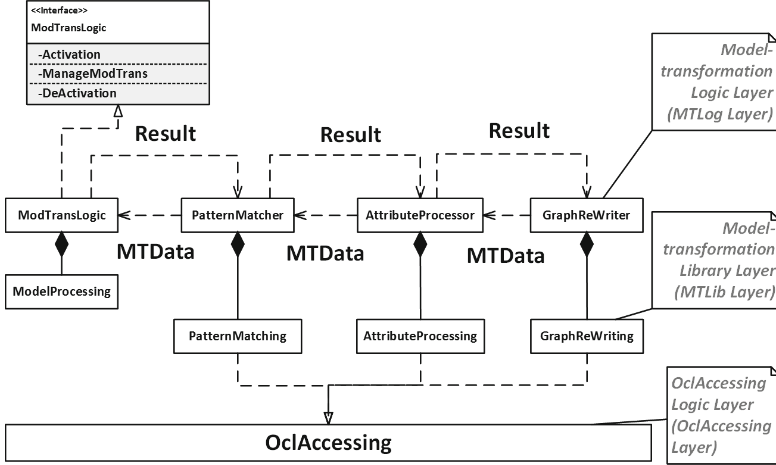

Fig. 1. The main designing concept and the three logical steps

to the GPU. It is important that kernels working on attribute constraint evaluation must receive only the necessary attribute data in the concatenated format. Otherwise, the cost of transferring the data from the CPU to the GPU would seriously degrade the performance of the approach. (iii) Rewriting applies the modifications defined in the rewriting rules by modifying the data sets representing the domain model. To avoid inconsistencies caused by parallel execution, the result is verified just before rewriting. After rewriting, we also have to decide whether the graph transformation is finished, or another rewriting is required (for "as long as possible" rules). Finally, if applying the rewriting rule is finished, the input domain model is changed based on the data sets of the transformation using the domain adapters. All three steps have input and output data, which is not stored but rather is temporally used by steps. Each step obtains an input data and then processes it and generates the output. The data is composed of three parts: (i) the model (accessed via modelProcessing package), (ii) the transformation rules (iii) and the temporal results. By rigidly separating the steps, a highly modular and easily extendable design is achieved. The logical steps have several kinds of responsibilities like supporting the scalability issues of the actual step, and preparing and configuring of the core algorithms, which belong to the Model-transformation Library Layer (Fig. 1). Library components can be easily exchanged to vary the dynamic behavior of the engine by using template programming. The common interface of the steps and the modularity also support the testability.

3.3 Illustrating the Topological Match

In order to illustrate the truly parallel behavior of the engine (same running time results are received in several case studies), the pattern matching logical

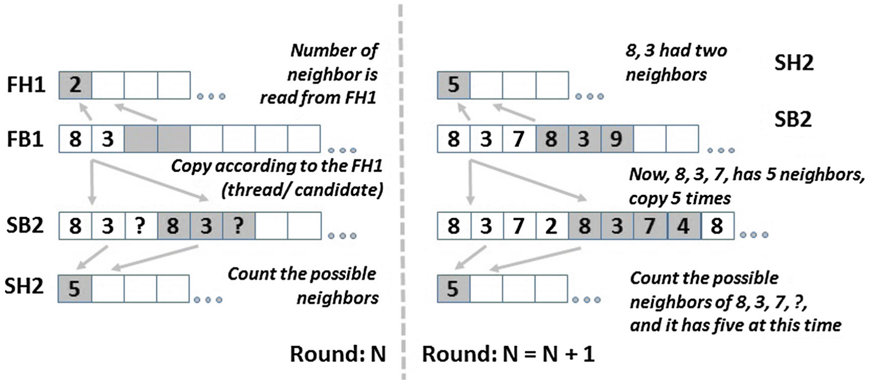

Fig. 2. Handling the buffers during pattern matching

step is detailed. The main concept is that we start a kernel from each potential matching point. Initially we try to find the first entity in the pattern, then the submatch is extended with the second entity, etc. Each kernel obtains a submatch already found and returns with its possible completion. Thus, pattern matching is applied in several steps. Four temporary buffers are used (Fig. 2) during the process: (i) FH1 - first helper, (ii) FB1 - first result candidate, (iii) SH2 - second helper, (iv) SB2 - second candidate. The kernel binary reads FH1 and FB1 and writes SH2 and SB2. The host applies two important steps before calling the kernel. First, it cumulates the numbers in the first helper buffer to provide information about the index of the candidates, then it swaps the first and second buffers. The kernels always work from the first buffers and save their result to the second: (i) The kernel copies the candidates from the first buffer to the second buffer and also takes the new neighbor using the helper buffer and the kernel worker thread ID. The number of threads started is equal to the number of new candidates. Each new thread knows its base candidate and copies the candidate from the first buffer to the second buffer. (ii) The thread knows which neighbor is to be taken to the new empty position. (iii) The thread validates whether the new candidate is matching. In the case of a mismatch, the thread sets the number of possible new neighbors to zero. If the new candidate is matching, the thread adds the number of potential new neighbors that must be checked in the next loop. Finally, the new candidate buffer is created. We have built our tool by following the principles of Test-driven development. Many test cases were created and applied from the beginning. This method helped us to find implementation issues and avoid degeneration of the code. Later on, we have searched for a domain that can be used to apply transformations. The Internet Movie Database[4] (IMDb) was chosen. Because of its size, IMDb data is perfectly suited for scalability measurements and for performance tests. We applied several tests on the database in our earlier researches [9].

[4] http://www.imdb.com/interfaces.

4 Conclusion and Future Work

The continuous growth of modeled systems is driving the focus on high performance model transformation solutions. We believe that using the remarkable potency in computing power of GPUs provides a solution to this issue. We are currently working on an OpenCL-based model transformation engine. In this paper, we introduced our framework PaMMTE by showing the basics of our approach and the most important parts of our engine, as well as illustrating the mechanisms by elaborating the steps of the pattern matching in more detail. Although our results are already promising, there are further acceleration and optimization points to discover and apply. The tool supports only the application of a single rewriting rule, not a complete sequence of rules. Our current research involves implementing a control flow that allows defining the sequence of rules and data transfer between them. The usage of further real-life case domains and studies can bring new challenges to solve. In the meantime, the achieved results can be used in MDE tools to accelerate their performance. Processing data, like Ecore, is a task for the future and it will give us a chance to create practical comparisons to other MDE tools.

References

1. Ehrig, H., Rozenberg, G., Kreowski, H.J.: Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific, Singapore (1999)
2. Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedüs, A., Herrmannsdorfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K., Lepper, M.: A survey and comparison of transformation tools based on the transformation tool contest. Sci. Comput. Program. **1**(85), 41–99 (2014)
3. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16145-2_6
4. Strüber, D., Kehrer, T., Arendt, T., Pietsch, C., Reuling, D.: Scalability of model transformations: position paper and benchmark set. In: Workshop on Scalable Model Driven Engineering, pp. 21–30 (2016)
5. Yan, X., Shi, X., Wang, L., Yang, H.: An OpenCL micro-benchmark suite for GPUs and CPUs. J. Supercomput. **69**(2), 693–713 (2014)
6. Xu, Q., Jeon, H., Annavaram, M.: Graph processing on GPUs: where are the bottlenecks? In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp. 140–149. IEEE, 26 October 2014
7. Masek, J., Burget, R., Povoda, L., Dutta, M.K.: Multi-GPU implementation of machine learning algorithm using CUDA and OpenCL. Int. J. Adv. Telecommun. Electrotech. Sig. Syst. **5**(2), 101–107 (2016)
8. Szuppe, J.: Boost. Compute: a parallel computing library for C++ based on OpenCL. In: Proceedings of the 4th International Workshop on OpenCL, p. 15. ACM, 19 April 2016
9. Fekete, T., Mezei, G.: Generic approach for pattern matching with OpenCL. In: Proceedings of the 24th High Performance Computing Symposium. Society for Computer Simulation International, p. 15. ACM, April 2016