Truly Parallel Model-Matching Algorithm in OpenCL

Tamás Fekete(✉) and Gergely Mezei

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

Abstract. The Model-driven Engineering (MDE) is coming into focus faster and faster nowadays because it can significantly simplify and accelerate the software development and maintenance processes. MDE can efficiently reduce resource requirements not only in development, but also in refactoring and maintenance tasks of complex software systems. There are several tools to support MDE. Although, these tools can deal with the average size of the currently applied domain models, the growing software systems can cause challenges in model manipulations. The growing size of systems can result in such a slow computation which cannot be accepted anymore. Therefore, more efficient model processing methods are needed. We are working on a complex, high performant model-transformation engine for MDE tools. Our solution can take the advantage of parallel computation available for example in modern GPUs. The engine is referred to as PaMMTE (Parallel Multiplatform Model-transformation Engine). In earlier publications, the architecture and functionality of our engine has been introduced and the functional correctness has also been proven. In this paper, we introduce a new pattern matching algorithm. The algorithm is truly parallel, it is scalable and more efficient than the previous version. Moreover, we analyze the current and the new pattern matching algorithms in general and the performance gain achieved. The new pattern matching algorithm can be effectively used not only in PaMMTE, but in any other cases, when high-performant pattern matching computation is required.

Keywords: PaMMTE · High-performant computation · Model-transformation · OpenCL framework · Software architecture · C++

1 Introduction

The Model-driven engineering (MDE) can efficiently simplify the software development which causes the sudden spreading of its usage in the software industry.

T. Fekete–This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project. This project is supported by the New Széchényi Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

R. Silhavy et al. (eds.), *Software Engineering Trends and Techniques in Intelligent Systems*, Advances in Intelligent Systems and Computing 575, DOI 10.1007/978-3-319-57141-6_13

MDE works with models, which are not only created for presentation purposes anymore, but transformed, processed and often used directly or indirectly as the basis of the code generation. Therefore, it is an important and challenging part of MDE to find and apply efficient model-transformation methods. Several techniques exist; the graph rewriting-based transformation (referred as a graph transformation) is one of the most popular among them. Graph transformation is based on an NP complete problem (subgraph isomorphism) and may need serious amount of time depending on the size of the input model and the pattern to search for.

Motivated by the increasing requests for high performant model transformation engines, we analyzed the capability of existing tools. There are many studies and surveys (e.g. [1]) that collect and classify the model transformation tools, like GREAT, IncQuery or MOLA. While all of them are efficient and flexible to some extent, none of them is capable of using GPU-based parallel execution. We have decided to fill up this gap and create a new model-transformation engine supporting both usual features of model transformation and efficient GPU-based parallelism. The engine is referred to as PaMMTE (Parallel Multiplatform Model-transformation Engine). The core algorithm of a graph transformation engine lies in the pattern matching, this is the most computation intensive part. Therefore, currently we are focusing on this part. We have analyzed the working mechanism of our original matching algorithm and the architectural structure of GPUs. We have found that our original solution is not GPU-specific enough, although it has several steps applicable in parallel, we are still heavily relying on the CPU-based computation, which is the bottleneck from the performance's point of view. Therefore, we have created a completely new algorithm, which fits much more in the GPU-based world. In this paper, we present this new, truly parallel algorithm. Besides the details of the algorithm, we also present a short comparison of the original and the new algorithms.

The rest of the paper is organized as follows: In Sect. 2, the base conceptions are introduced which were assumed during the creation of the engine. In Sect. 3, for the sake of simplicity, a short overview is given about the base architecture of PaMMTE focusing on the part which is modified. In Sect. 4, main novelty of the current paper can be found in details. In Sect. 5, our theoretical conception is validated by measurements applied in a case study. In Sect. 6, we conclude and give some directions for the possible future research.

2 Related Work

On the market, numerous kinds of GPUs and other hardware elements can be found which have the ability to apply highly-parallel computation. Using a vendor or model-specific language and framework would need a tremendous effort and each user would need to provide the proper environment according to the available hardware. To avoid this, the OpenCL framework has been released in 2009. OpenCL is a platform independent framework which can be applied to handle the most widely used hardware uniformly (CPU, GPU, FPGA, DSP).

OpenCL is an interface defined by Khronos Group [10]. Each product vendor has its own implementation. In addition to move the computation into OpenCL devices, the computation capacity of the primary hardware (CPU) is less used producing thus a more balanced system in general.

At the beginning of our research, it was the one of center questions to be examined whether the usage of the OpenCL framework can be as good as using any other hardware specific environment. We studied this point carefully and also searched in the literature for other's results. In [8], we compared and evaluated several GPGPU-based solutions, OpenCL-based libraries and applications. Papers, like [6] pointed out that OpenCL framework can be effectively used in our research too and there are lots of optimization points which are probably different in case of variant hardware. It indicated us to collect the optimization opportunities. There are further examples in [5], which gives details how important the graph processing components are. [5] also focuses on mapping algorithms between the host and the GPU devices which is a big challenge in the effective usage of GPUs. They mapped 12 graph applications into the GPU device, studied the performance and suggested several approaches to accelerate the performance of the GPU-based algorithms.

There are further new studies which have influenced our current research: In [2], the k-Nearest Neighbor algorithm is implemented using OpenCL and CUDA. There is a big difference between the two implementations, the most emphasized is that CUDA is strongly hardware-dependent, while the OpenCL framework can be used on many hardware platforms. There are several measurements and comparisons between devices with a single CPU and devices with a CPU and a GPU. Furthermore, there is also a comparison between OpenCL and CUDA: in some measurements OpenCL seems to be perform better, but not in all cases. The thesis [3] compares several hardware and software solutions. One of the main reasons of the rapid spreading of highly-parallel computation (and the growing number of the computation units) is the expensive computation requirements in computer games. [3] gives an overview and compares not only the main competitors of the OpenCL framework (e.g. CUDA), and it also presents a wide benchmark. It highlights several solutions for parallel computation (e.g.: CUDA, OpenGL, DirectX, OpenMP). [4] provides the usage of the OpenCL with some C++ and STL related features (meta-programming) as part of the official Boost library. This part of the Boost library can also be used as a thin wrapper.

Taking all advantages (e.g. platform independence) and possible disadvantages (e.g. performance loss) into account finally we have decided to build our engine based on OpenCL. Because of the importance of performance, we use the OpenCL interface directly (not through a high level wrapper) and fine tune the performance keeping in mind that our final goal is to create a high-performant model transformation engine.

3 Architecture of PaMMTE

The architecture and working mechanisms of PaMMTE are complex. In [7], we elaborated them in details, however, in this paper, we give only a short overview,

since the focus is on the new matching algorithm. The architecture of PaMMTE can be divided into three layers (Fig. 1):
(i) Model-transformation Logic Layer (MTLogic Layer): The model-transformation process is split up into three main model-transformation steps, all of them are implemented in the highest layer. Input model is read, processed and the output is also evaluated in the this layer. The three main steps of the model-transformation logic are the followings in the order of the first calls: pattern matching (PatternMatcher package), attribute processing (AttributeProcessor package) and finally the rewriting of the graph (ReWriter package). Note that: between the packages, in the highest layer, the model-transformation data (MTData) is passed again and again. MTData contains the processed input domain model, the pattern to be found and the results of the actual steps.
(ii) Model-transformation Library Layer (MTLib Layer): The middle layer contains the concrete pattern matching, attribute processing and model-rewriting algorithms which manage the core computations based on the OclAccessing Layer. Pattern-matching algorithm searches only for topological matching by using the symbol of the nodes in the input graph created from the input domain model. This is later extended and re-checked by the attribute processing step, where we check the attributes of the given nodes as well. The focus of the current study is the pattern-matching part of the engine. In MTLib, we introduced a common interface for pattern matching, attribute processing and model rewriting algorithms, thus each of them can be easily exchanged.
(iii) OclAccessing Layer: The lowest layer is a kind of abstraction layer. Other layers have no information about the type and the number of the currently used OpenCL devices, all of these are hidden from higher layers. Each OpenCL-based computation is managed via a general context provided by the OclAccessing Layer.

The presented architecture has several advantages: (i) The domain-related logic is implemented only on the highest level separated from the core algorithms (which are in the middle layer). Adding new model-transformation steps and changes in the hard coded configuration can be managed safely and easily. (ii) The core algorithms can be exchanged using a common interface and it can be replaced at run-time too. (iii) There is no hardware dependency, because the OclAccessing Layer provides a general context to the computation libraries. (iv) Implementation of the main interface of the modelPocessing package allows us to easily use new domain models. To achieve high-performant computation only C++14 is used in the implementation of PaMMTE and the only 3rd party dependency is the Boost library. To configure the engine, an XML file must be used following a predefined schema. The main development environment is the MS Visual Studio 2015 C++. During the development, we used elements from the Test-Driven Development (TDD) methodology to prove the correct functionality of the engine. In this paper, we focus on the improvement of the patternMatching package in order to create a generally usable truly pattern matching algorithm. We also compared two pattern matching solutions for OpenCL devices in [8]. In that research, we used the advantages of run-time information during

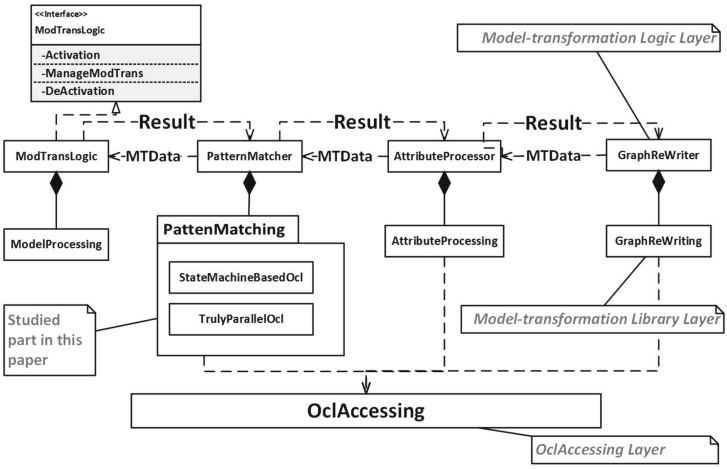

Fig. 1. The base architecture (packages) of PaMMTE.

building the kernel code. Now, the main goal is to achieve a truly parallel pattern matching to increase the performance of the whole model-transformation.

4 Towards the Truly Parallel Pattern Matching Algorithm

In [8], we introduced the importance of pattern matching algorithms and pointed out some open issues in pattern matching to be solved later. Since then, we managed to create a truly parallel pattern matching computation in our engine which is the main contribution of this paper. The main steps of both the old and the new algorithms are listed, as well as, the most important commonalities and differences.

4.1 Properties and Issues of the Old Algorithm

In the old algorithm, the OpenCL kernel was executed several times when result buffer overflow has occurred. We defined formulas to calculate the optimal size of the output result buffer and also designed and implemented a strategy, when the buffer is rarely used. The number of the threads equals to the number of the nodes in the input graph to be processed (each thread processes exactly one node). In case of buffer overflow, only those elements are re-used, which were not processed before, when the kernel is executed again. Avoiding to avoid processing one node two times does not significantly saves any time, because other threads must be waited. Although, in some cases, one node can have only a few neighbors, which results in a fast computation for those threads, there can be other threads with lots of neighbors. The kernel computation cannot be

finished from the viewpoint of the host until each thread has finished the task assigned to it.

As it can be seen, the old algorithm works with parallel threads, but not in an efficient way. In some cases, only a few threads work. Another viewpoint is that, each thread has a complex inner state. The state has to store the parent node, the currently processed neighbor number, the deepest level, which means how far the actual node is from the pivot point and so on. In the current paper, we use the term of candidate multiple times with the following meaning: *candidate is part of the input graph which is supposed to match to a part of the pattern. Moreover, if the size of the candidate equals to the size of the pattern and they are still matching, the candidate is already a matching result.* Each thread tries to create a small candidate at the beginning and checks whether that candidate is matching or not. If the candidate is matching the thread takes a new neighbor to extend the size of the actual candidate and checks again the matching state. This process is applied until then the size of the matching candidate equals to the size of the pattern. To find results, in the implementation of the first kernel, there are two nested state machines (the first digs deeper in the graph while the second finds each neighbors at current level) and two function calls (to validate the candidate) which are not the best way to achieve optimal performance for a data oriented computation model such as used in OpenCL. In short, the old algorithm was executed semi-parallel.

4.2 The New Pattern Matching Algorithm

Overview of the New Pattern Matching. To evaluate the matching algorithm truly in parallel and avoid buffer size estimation, we created a new kernel source code. There are some lower (hardware) level assumptions which are considered in order to achieve the truly parallel functionality and increase performance: (i) Compare two numbers is fast on hardware level (in general, hardware computation units use gates instead of bit evaluation one-by-one to compare two numbers). (ii) The memory allocation is fast both on the host and on the OpenCL side. However, on host side, handling of memory fragmentation is required because of the big amount of data processed. The new approach is illustrated in Fig. 2. To find all results, the kernel is executed several times (two executions in this particular case). We use four buffers, but only these buffers are changed during the computation. Others, graph or pattern related buffers do not change during matching. The four buffers are the followings: (i) FH1 - first helper buffer, (ii) FB1 - first result candidate buffer, (iii) SB2 - second candidate buffer, (iv) SH2 - second helper buffer. The role of the buffers is explained later.

Behaviour of the New Pattern Matching. Since in the general case, the kernel is executed several times, let us suppose that the current loop number is N (kernel is already executed N-1 times successfully). Now, the following steps evaluated:

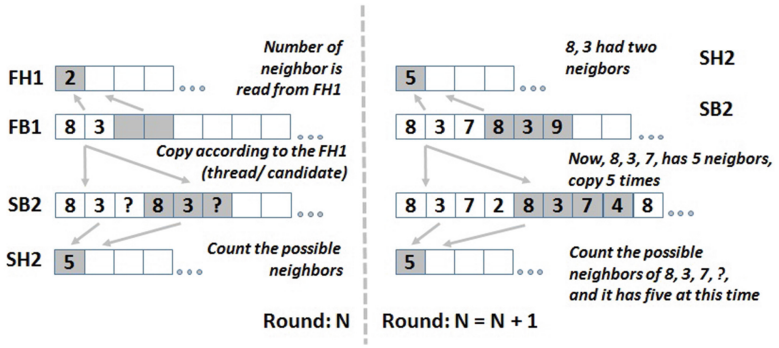

Fig. 2. Buffers in the new algorithm.

(i) Determine the size of the FH1 and FB1 buffers: We store the data of M candidates here. The length of the candidates is N, the size of the first-CandidateBuffer (FB1) is N*M. The values of the elements of FirstHelperBuffer (FH1) denote how many new neighbor each candidate can have. Since, we need only one value for each candidate, the size of FH1 is M.
(ii) Determine the size of the SB2 and SH2 buffers: As far as FH1 is cumulated on the host, let C refer to the last value of the last element of the cumulated helper buffer (FH1). In this case, the size of the secondCandidate-Buffer (SB2) is C*(N + 1). The meaning of secondHelperBuffer (SH2) is similar to that in the previous case: it shows how many new neighbors the candidates have. The size of SH2 is C size is C.
(iii) Copy candidates from the FB1 to the SB2: Each thread is responsible for exactly one candidate. The thread copies that candidate and adds the new neighbor. The thread checks whether the filled up candidate matches. If yes, it computes the number of the possible new neighbors and stores the number of neighbors in SH2.
(iv) Change the pointers on the host side: On the host side, the pointers of the FB1 and the SB2 are exchanged. Similarly, the SH2 is replaced with the FH1.
(v) Prepare the new result buffers: The first helper buffer is read from the global memory of the OpenCL device and cumulated on the host side. Based on the accumulation, we re-calculate the sizes of SB2 and SH2, moreover, the number of the threads received from the result of the cumulating are also recalculated. The SB2 and the SH2 are freed and new arrays are allocated with empty content.

Details About the New Pattern Matching. The kernel binary always works on the four buffers, it reads FH1 and FB1 and writes SH2 and SB2. The host manages two important steps before calling the kernel. Firstly, it cumulates the numbers in the first helper buffer, then it swaps the pointers of the first and second buffers. The kernel always works from the first buffers and saves the result to the second buffers: (i) The kernel copies the candidates from the first

Table 1. Average time values of several measurements are collected (only for the pattern matching).

Platform	Intel (time)	Nvidia (time)
New OpenCL Kernel	112	132
Old OpenCL Kernel	15761	15788
Host version of the old one	1264	

buffer to the second buffer and adds to the new neighbor using the helper buffer and the thread id (each thread uses the same formula to find the place of the old and the new neighbor candidate). The number of the threads equals 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 (each thread copies the same number of elements). (ii) The thread knows which neighbor must be taken from the input graph to the new empty place. (iii) The thread verifies whether the new candidate is matching according to the pattern. In case of mismatching, the thread sets that the number of possible new neighbors to zero. If the new candidate is matching, the thread adds how many new neighbors must be checked in the next loop. Finally, the new candidate buffer is created.

5 Case Study

To test and evaluate the performance gain of the new algorithm, we measured the computation time in a case study. We selected the Internet Movie DataBase (IMDb) [9] as the target domain. IMDb is the largest film and TV show related database which is publicly available. It has approximately 3.3 million titles and 6.5 million personalities (actors, directors, etc.). IMDB contains information on several domain concepts, like movies (subtitle, creation time, and rate), actors (with movies they played in) and producers (with their movies). A simple example for a pattern to be searched is: *"Three actors playing in the same movie. The movie has an attribute showing that the movie is made in the USA. Furthermore, the first name of the director is Jack and at least one actress (besides the three actors) must play in the same movie."*. As far as the first logical step of the model-transformation works only with the symbols of the nodes, the other part of the rewriting rules are not considered now. Similarly, only the pattern matching part is measured in the current case study.

The test environment used in the case study is a simple notebook with the following configuration: Intel Core i7 HD Graphic 5500 and Nvidia Geforce GTX 950M with a Windows 10. In Table 1, the results are collected. Note that the results are the average of ten measurements. In this case study, we measured and compared three pattern matching algorithms: (i) the old algorithm, (ii) the old version executed on the host (CPU) and (iii) the new algorithm. Although, we do not intend to compare two different kinds of architectures/technologies, it also can be seen that even a relatively week GPU can result in faster computation

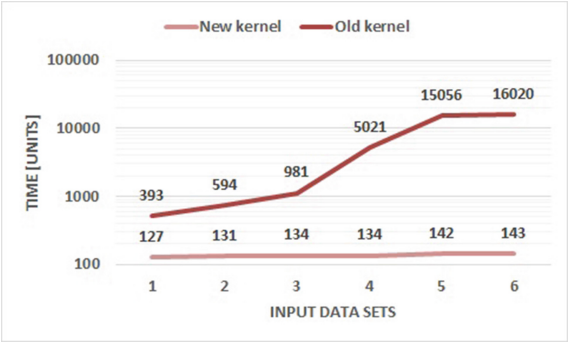

Fig. 3. Time results in case of different input models.

that a strong CPU. Among the algorithms, the first one uses the smallest input data and the simplest pattern. The last one uses the biggest input data and the most complex pattern.

In Fig. 3 the time of the execution is compared for the old and the new algorithms in case of different input domain models (the pattern to be searched is not changed). In case of the old algorithm, the computation time is directly proportional to the size of the input domain model. Bigger and more complex input graphs require more time to find each matching, because of the complex inner state machines and growing complexity. In contrast, the new algorithm is completed almost in the same time at each measurement. The reason of the almost constant time is the truly parallel behaviour of the new algorithm. However, it is suspicious that a remarkable portion of the time is caused by the constant time required by preprocessing steps (e.g. memory copy) and thus the real calculation needs increasing amount of time. We need a series of measurements in order to examine this question in detail. Note that this behavior does not change the fact that the new algorithm is several orders of magnitude faster than the old one.

6 Conclusion

The MDE approach is gaining more and more interest nowadays as we have to deal with bigger and bigger software systems. There are several tools to support model-based development however, existing tools does not support parallel execution natively. Our aim is to solve this issue. We have created an engine for model-transformation which is based on a new approach to achieve high-performant multiplatform computation. The base and the novelty of our engine is the usage of the OpenCL framework to significantly accelerate model-transformation. Pattern matching is one of the most important parts of a model-transformation engine. In the current research, we improved the pattern matching part of PaMMTE. The solution is presented, analyzed and validated by measurements using a case study. Other model-transformation steps like the attribute processing and the model rewriting are not changed in this paper.

By changing the pattern-matching algorithm, we could achieve a truly parallel model-transformation engine, which is deployed in PaMMTE, but it can also be used in any other similar cases. In the future, we are going to search for further case studies and to test the solution with other platforms to ensure that our solution works well with most of the OpenCL devices. Furthermore, we must study the advantages and the disadvantages of the algorithms.

References

1. Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedus, A., Herrmannsdorfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K., et al.: A survey and comparison of transformation tools based on the transformation tool contest. Sci. Comput. Program. **85**, 41–99 (2014)
2. Masek, J., Burget, R., Povoda, L., Dutta, M.K.: Multi-GPU implementation of machine learning algorithm using CUDA and openCL. Int. J. Adv. Telecommun. Electrotechn. Sig. Syst. **5**(2), 101–107 (2016)
3. Sorman, T.: Comparison of technologies for general-purpose computing on graphics processing units (2016)
4. 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 (2016)
5. 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 (2014)
6. Yan, X., Shi, X., Wang, L., Yang, H.: An openCL micro-benchmark suite for GPUs and CPUs. J. Supercomput. **69**(2), 693–713 (2014)
7. Fekete, T., Mezei, G.: Architectural challenges in creating a high-performant model-transformation engine. Subsequences. In: The 10TH Jubilee Conference of PhD Students in Computer Science, p. 20 (2016)
8. Fekete, T., Mezei, G.: Creating a GPGPU-accelerated framework for pattern matching using a case study. In: 24th High Performance Computing Symposium (HPC16), Pasadena, CA, USA (2016)
9. IMDb - Movies, TV and Celebrities - IMDb (2016). http://www.imdb.com/interfaces
10. OpenCL - The open standard for parallel programming of heterogeneous systems (2016). https://www.khronos.org/opencl