Optimizing ELARS Algorithms using NVIDIA CUDA Heterogeneous Parallel Programming Platform

Vedran Miletić, Martina Holenko Dlab, and Nataša Hoić-Božić

University of Rijeka, Department of Informatics
Radmile Matejčić 2, 51000 Rijeka, Croatia
{vmiletic,mholenko,natasah}@inf.uniri.hr

Abstract. Scalability is an important property of every large-scale recommender system. In order to ensure smooth user experience, recommendation algorithms should be optimized to work with large amounts of user data. This paper presents the optimization approach used in the development of the E-learning activities recommender system (ELARS). The recommendations for students and groups in ELARS include four different types of items: Web 2.0 tools, collaborators (colleague students), optional e-learning activities, and advice. Since implemented recommendation algorithms depend on prediction of students' preferences, algorithm that computes predictions was offloaded to graphics processing unit using NVIDIA CUDA heterogeneous parallel programming platform. This offload increases performance significantly, especially with large number of students using the system.

Keywords: e-learning, recommender system, ELARS, algorithm optimization, heterogeneous paralell programming, NVIDIA CUDA

1 Introduction

Recommender systems support users in identifying services in information-rich environments. These systems can provide a solution for the information overload problem by recommending items that are potentially useful for the target user or that are within the scope of his/her interests [1]. In addition, recommender systems ensure personalization since recommendations are generated according to user's characteristics. Therefore, recommender systems are often used across different domains like entertainment industry, e-commerce and e-learning [4].

The usefulness of items (utility) that can be recommended is expressed as a numerical value (rating). This value is determined by the user or it can be predicted. Accordingly, the recommendation problem comes down to the prediction of the unknown utility values in order to recommend item or items with the highest utility to the target user. Prediction algorithms are usually based on two commonly used methods: collaborative filtering and content-based recommendations [1,4]. Performance of the recommender system depends on the accuracy or

precision of the prediction algorithm. Therefore, appropriate algorithm should be chosen based on experiment in which a few algorithms are compared using some evaluation metric. Another important property of the recommender system that may affect user experience is scalability. The system should be able to scale well, ideally both horizontally and vertically, to keep up with increasing number of users navigating through ever-enlarging collections of items [18], producing large amount of data to be analyzed. ELARS, described below, is an example of such a system, and our research described in this paper is on scaling ELARS for a large number of users.

Horizontal scaling, or scaling out, implies adding more nodes to a distributed system, in this case adding more servers to a cluster running the application [17]. Vertical scaling, or scaling up, means adding resources to a single node in the system, in this case adding additional central processing units (CPUs), graphics processing units (GPUs), memory etc. When required to scale, one can opt for horizontal or vertical scaling, or combine both. There are advantages to each of two approaches. From the programming standpoint, vertical scaling is simpler, but tends to be limited by hardware specifications (i. e. one can only fit a limited number of CPUs and GPUs or limited amount of memory in a single node). Horizontal scaling enables addition of more resources. However, it requires a more complex programming model that may or may not fit a particular application. Also, larger numbers of nodes in a distributed system implies increased management complexity.

This paper presents our approach to performance optimization of ELARS algorithms using NVIDIA CUDA heterogeneous parallel programming platform enabling code to run on both GPU(s) and CPU(s). We found the preference prediction to be particularly demanding in terms of computation time. Suitable parts of preference prediction algorithms are moved to GPU for execution to improve overall performance. Meanwhile, CPU executes the parts not suitable for the GPU. Our approach utilizes a single node and allows vertical scaling with introduction of faster CPUs and GPUs. Future server systems running web application will be increasingly heterogeneous, with its computation power divided over a number of different processors [12]. Our approach contributes to promotion of heterogeneous parallel programming usage in web application development. To the best of our knowledge, this is also first application of NVIDIA CUDA in domain of e-learning.

The paper is organized as follows: first we present ELARS, then we introduce heterogeneous parallel programming approach with focus on NVIDA CUDA, then we describe our approach to algorithm parallelization. We do performance benchmarks, and conclude along with possible directions for future work.

2 E-Learning Activities Recommender System

E-learning Activities Recommender System – ELARS [10] supports collaborative e-learning activities in an online learning environment that consists of a learning management system (LMS) and 10 different Web 2.0 tools [3]. In such environ-

ment, students use LMS to study the learning content, read the instructions, solve online tests, communicate with others and similar. They use Web 2.0 tools for realization of e-learning activities like seminar writing, mind-mapping or WebQuests. Students use recommender system in parallel with other components of the learning environment to choose between recommended items.

2.1 Recommendation Algorithms in ELARS

The system provides personalization by recommending optional Web 2.0 tools, collaborators, optional e-learning activities (e-tivities), and offering advice to students and groups. Recommendations are based on several students characteristics [11]: preferences of Web 2.0 tools, preferences of learning styles, knowledge level and activity level. Preferences of Web 2.0 tools and learning styles are collected via questionnaires at the beginning of the course. Knowledge level is determined based on student's results on online tests and activity level, which represents quantity and continuity of student's (group's) contributions in e-tivities, is calculated based on activity traces collected from Web 2.0 tools.

From the scalability point of view, the most challenging task of the recommendation process is to predict not known Web 2.0 tools preferences that are used for calculating utility of potential collaborators, offered Web 2.0 tools or optional e-tivities. Web 2.0 tools preferences are predicted using hybrid approach [16]. Recommender switches between collaborative filtering and content-based recommendations based on the number of known preferences in the system's database. Using collaborative filtering technique preference of the target student for target tool is predicted based on preferences of similar students (nearest neighbours) for target tool [1]. n case nearest neighbours cannot be found, content-based recommendations technique [16] is used and preference is predicted according to target student's preferences for other tools.

2.2 System Performance Bottlenecks

Since the number of system's users is much bigger than the number of tools included in the learning environment, performance bottleneck was found in case of collaborative filtering. This technique is performed in two phases which can both be addressed using GPU: neighbourhood selection and target tool predicted preference computation.

Neighbourhood Selection. When performing collaborative filtering the recommender system uses knowledge about similar users' preferences regarding target tool [1]. Therefore, similarity of the target student with students for which target tool preference exists in the system database is calculated. Similarity between students is determined based on known preferences or learning styles preferences according to VARK model [6]. In that process we calculate cosine similarity [15] which is commonly used metric for collaborative filtering. To form the neighbourhood, k students who are the most similar to the target student are selected. The configuration parameter k is set to 20 using cross-validation.

Predicted Preference Computation. Preference value pp is predicted based on normalized preference values for nearest neighbours tp'_i [1] using formula 1. Normalization of neighbours' preferences using mean-centering method is performed to unify the criteria on the basis of which neighbours expressed their preference. Besides normalization, to increase the accuracy of prediction algorithm weighting factors w_i are used. Values w_i represent the similarity of neighbour i with target student. By using such weights the influence of preference from certain neighbour to the result value is bigger if he/she is more similar to the target student. This effect is further improved by introducing amplification factor $\alpha = 2$. The resulting value is normalized using expression in the denominator and added to the target student's mean preference $\overline{tp}$.

$$pp = \overline{tp} + \frac{\sum_{i=1}^{k} w_i^{\alpha} tp'_i}{\sum_{i=1}^{k} w_i^{\alpha}} \tag{1}$$

3 Heterogeneous Parallel Programming

Usage of graphics processors for general-purpose computing started with programmable shaders on the NVIDIA GeForce FX and AMD Radeon series of graphics cards in early 2000s [20]. Shaders were programmed using either High-level shading language (HLSL) from Microsoft DirectX, OpenGL Shading Language (GLSL), or NVIDA Cg. Despite the requirement to significantly change the algorithms to adapt them for graphics processing unit (GPU), programming non-graphics problems using shaders became popular soon afterwards.

NVIDIA recognized the potential for the utilization of the GPU for general purpose calculations, and with GeForce 8 series opened up the GPU using a custom application programming interface (API) named Compute Unified Device Architecture, or CUDA for short [13]. CUDA has been made available to the public in February 2007, and is supported by all NVIDIA graphics processors released since.

Despite the appearance of the open standard named OpenCL which has the same purpose as (proprietary) CUDA, CUDA and therefore NVIDIA continues to dominate the market. Beside being first to appear, it is also due to greater amount of literature available and better programming tools. While both standards are very similar, they are not compatible [5].

3.1 Related Research Efforts

GPUs have so far been used to solve problems in bioinformatics, chemistry, physics, mathematics, medicine, mechanical engineering, electrical engineering, computer science, and other science and engineering disciplines. Garland et. al. survey applications of CUDA to a diverse set of data parallel problems, finding varying speedups of GPU-enabled algorithms vs CPU-only versions depending on the algorithm properties [8]. Gregg and Hazelwood, as well stress that any benchmarks that lead to conclusions on speedup should provide information on

where the data is assumed to be, because copying data from CPU memory to GPU memory and back can take a significant amount of time [9]. It also is worth noting applications of CUDA that could be used in recommender systems; Garcia, Debreuve, and Barlaud implement brute force k nearest neighbours selection using CUDA C and conclude that GPU speedup can be up to 120 times compared to equivalent CPU code implemented in C [7], including data copies from CPU to GPU and back in computation time.

3.2 GPU Architecture and CUDA Programming Model

Single instruction, multiple data (SIMD) is a class of parallel processors that have a larger number of processing elements that can do the same operation on multiple data simultaneously. This feature exploits data-level parallelism. GPUs, unlike most central processing units (CPUs), are SIMD processors, which allows acceleration of suitable algorithms. Performance gains vary greatly, and can be anything from a couple of percent to one or even two orders of magnitude.

From now on, we focus solely on programming NVIDIA GPUs using CUDA programming language. CUDA began as an extension of programming languages C/C++ and Fortran. Special directives were added to both languages that allowed to offload parts of computation to GPU.

CUDA API exposes threads, which are grouped in blocks of threads, which are again grouped in grid of blocks. Each block allows indexing in three dimensions, while the number of dimensions for grid is limited to two. This programming model is intended to fit multidimensional arrays. Functions written in CUDA intended to run on the GPU are named kernels. On each kernel call, the number of blocks and threads on which the kernel will be executed is specified. Therefore each kernel can be written for data of varying shape and size.

3.3 CUDA Libraries

CUDA ecosystem offers a number of libraries that simplify programming and even provide highly optimized versions of frequently used algorithms, for example reductions and sorting. In this work we use Thrust [2] and PyCUDA [14] which we describe below.

Thrust. Thrust is a C++ template library for CUDA based on C++ Standard Template Library. Thrust offers a number of data parallel primitives such as scan, sort, and reduce. These primitives can be used along with existing C++ code to ease offloading parts of code onto the GPU and enable rapid prototyping of CUDA applications.

PyCUDA. PyCUDA is a Python module that enables programmers to access CUDA API. Due to Python's clean syntax, it is very suitable for prototyping software. With PyCUDA, it becomes possible to also for prototype software that uses CUDA. In addition, PyCUDA enables access to existing CUDA C/C++ libraries such as Thrust.

4 Algorithm Parallelization Approach

We found Python and PyCUDA to be very suitable for rapid prototyping and comparison of different approaches to parallelization. We ported preference prediction code from existing C# implementation to Python, utilizing NumPy [19]. NumPy is a Python module providing high-level interface to C-like arrays for efficient numerical computation. Large parts of NumPy are implemented in C and Fortran; this in and of itself resulted in significant performance improvement in implementation of our algorithms done in Python and NumPy compared to implementation done in C#. During prototyping stage we also simplified the resulting program by caching data which is normally retrieved from database.

4.1 Neighbourhood Selection Parallelization

Since number of potential neighbours grows with increase in number of students – system's users, neighbourhood selection is a performance bottleneck. For example, if the ELARS was deployed at University of Rijeka which has nearly 20000 students, a popular Web 2.0 tool could easily have 10000 or even 15000 entered preferences.

Brute force search is used to select nearest neighbours. Thrust function `thrust::sort_by_key()`, which sorts key-value pairs, was a good fit for GPU version of the algorithm. Somewhat counter-intuitively, key set is similarity expressed as floating-point number, while the value set consists of student IDs. Since neither hashing nor numerical operations are done using these floating-point numbers, usage of floats here does not lead to problems. In each iteration key and value arrays are copied to the GPU, and after sorting both arrays are retrieved from the GPU.

4.2 Predicted Preference Computation Parallelization

To optimize predicted preference computation, we opted to compute all the predictions in a single kernel execution. To achieve it, learning similarities matrix is copied to the GPU on program initialization. After that, pairs of target students and tools are collected and stored in a 2D array, as well as normalized target tool preferences for all pairs. Normalized target tool preferences are computed on the CPU. Since it is required to select only non-null values from the array, we expect such selection done on GPU would slow down computation significantly and compensate for potential benefits of parallelization.

Both arrays are copied to the GPU, and computation of sum elements for nominator and denominator of formula 1 is done. Sum reduction can then be done on either CPU or GPU. If reduction is done on the CPU, whole array is copied back; if it is done on GPU, only a single resulting value is copied back.

5 Performance Measurements

A system with AMD FX-8150 8-core CPU and NVIDIA GeForce GTX 660 GPU was used for testing and benchmarking. We should emphasize that neither 64-bit

floating point precision nor large amounts of GPU memory are required in this domain, so commodity GeForce GPUs can be used as well as more expensive ones from Tesla and Quadro series.

We measure the computation time required to get student's tool preference for all students for all tools, that is, to predict all unknown preference values. In dataset, loaded from flat text files, we used for testing 57,5% of preferences are unknown. We measure computation time for 10 tools for 640, 1280, 2560, 5120, 10240, 20480 students. Computation time for each number of students for CPU-only and CUDA-enabled GPU and CPU code is shown in Figure 1. CPU code in both cases uses no parallelization and runs on a single CPU core.

On the GPU side, we should note that both PyCUDA GPU Array module and Thrust library dynamically determine number of blocks and threads to be used for computation depending on data and GPU used, without needing to be manually specified by user.

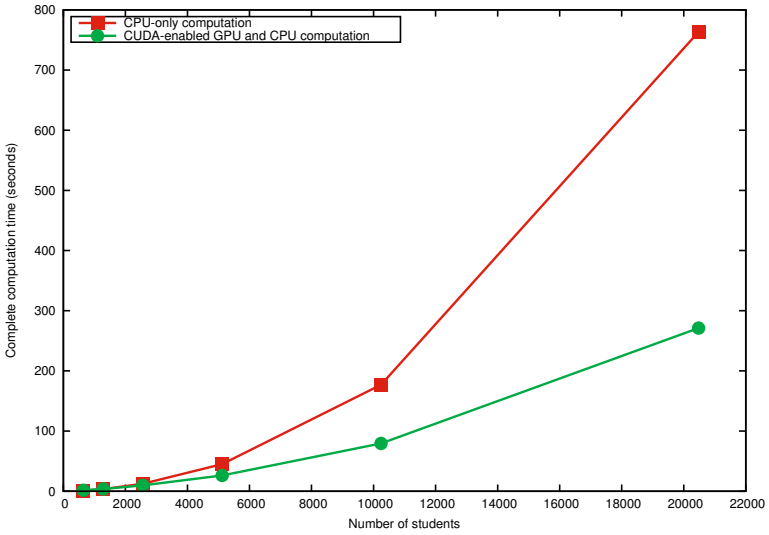

Fig. 1. Performance measurements for entire algorithm.

We can observe that for smaller number of students (640, 1280, and 2560) CPU-only and CUDA-enabled code are very close in terms of computation time. At 5120 students they start to visibly diverge, and difference becomes even greater in cases of 10240 and 20480 students. We can see that the gap widens with increasing number of students, arriving at nearly 3 times the speedup in favor of GPU at 20480 students.

Total execution time of sorting in k nearest neighbours selection, including copying of data from CPU to GPU memory and back, is shown in Figure 2. We can observe the exponential increase in computation time for the CPU, while the

for the GPU the increase remains linear for the number of students we measured. Difference in computation time starts to be apparent with 3 times GPU speedup over CPU at 5120 students, increase to nearly 7 times at 10240, and finally ends up being over 14 times at 20480 students. It is also apparent that k nearest neighbours dominates the computation time of the entire preference prediction algorithm in larger cases, taking nearly over half of computation time.

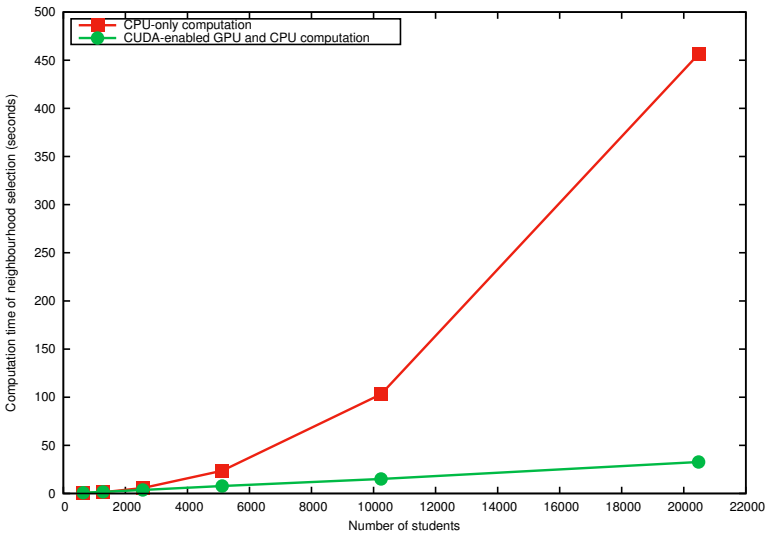

Fig. 2. Performance measurements for sorting in k nearest neighbours selection.

Total execution time of target tool predicted preference computation, again including copying of data from CPU to GPU memory and back, is shown in Figure 3. We can see how both CPU and GPU computation time increases linearly with the number of students, but GPU computation time consistently remains around two orders of magnitude smaller. In other words, we get GPU speedup over CPU between 75 and 100 times.

6 Conclusions, Discussion and Future Work

We presented an approach to optimization of recommender system performance by offloading parts of algorithms to GPUs. We find this approach to be reasonable considering the expectation that future machines will be increasingly heterogeneous, and their computing power will be divided across a range of chips with different characteristics targeting certain kinds of problems. We found the optimizations we used to improve performance significantly, and allow scaling for a larger number of users.

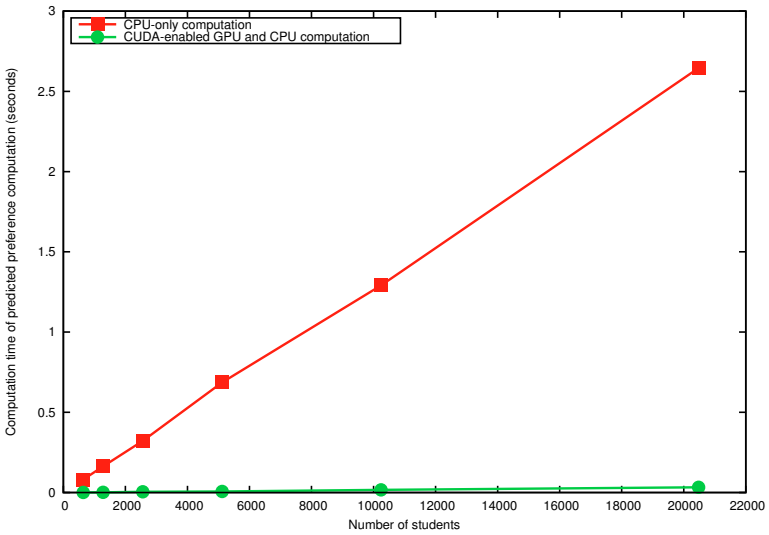

Fig. 3. Performance measurements for target tool predicted preference computation.

While CUDA dominates the market at present, it is realistic to expect that in the future OpenCL play a significant role. Intel and AMD, both supporting OpenCL, already ship most of their CPUs with integrated GPU, and recently with AMD Berlin APU making into the Opteron line of processors this trend moved this in the server domain as well. Both performance and power consumption of GPUs and APUs has the potential to make them an attractive choice in the server environment, even outside the usual scientific computing applications.

With these technology developments in mind, our other future plans include offloading even larger amounts of computation on the GPU, porting of CUDA-enabled code from Python to C# to ease integration within ELARS, and finally deployment in production at University of Rijeka.

Acknowledgments. The research has been conducted under the project "E-learning Recommender System" (reference number 13.13.1.3.05) supported by University of Rijeka (Croatia). Performance benchmarks were done at University CUDA Teaching Center provided by NVIDIA.

References

1. Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. Knowledge and Data Engineering, IEEE Transactions on 17(6), 734–749 (2005)
2. Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for cuda. GPU Computing Gems 7 (2011)

3. Dlab, M.H., Hoić-Božić, N.: An approach to adaptivity and collaboration support in a web-based learning environment. International Journal of Emerging Technologies in Learning (2009)
4. Drachsler, H., Hummel, H., Koper, R.: Identifying the goal, user model and conditions of recommender systems for formal and informal learning. Social Information Retrieval for Technology Enhanced Learning 10(2), 4–24 (2009)
5. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. Parallel Computing 38(8), 391–407 (2012)
6. Fleming, N.D.: I'm different; not dumb. modes of presentation (vark) in the tertiary classroom. In: Research and Development in Higher Education, Proceedings of the 1995 Annual Conference of the Higher Education and Research Development Society of Australasia (HERDSA), HERDSA. vol. 18, pp. 308–313 (1995)
7. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using gpu. In: Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on. pp. 1–6. IEEE (2008)
8. Garland, M., Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with cuda. IEEE Micro Magazine 28(4), 13–27 (2008)
9. Gregg, C., Hazelwood, K.: Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In: Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on. pp. 134–144. IEEE (2011)
10. Holenko Dlab, M., Hoić-Božić, N.: Recommender system for web 2.0 supported elearning. In: 2014 IEEE Global Engineering Education Conference (EDUCON) Proceedings (2014)
11. Holenko Dlab, M., Hoić-Božić, N., Mezak, J.: Personalizing e-learning 2.0 using recommendations. In: The Proceedings of MIS4TEL conference (2014)
12. Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., Glasco, D.: Gpus and the future of parallel computing. IEEE Micro 31(5), 7–17 (2011)
13. Kirk, D.: Nvidia cuda software and gpu parallel computing architecture. In: ISMM. vol. 7, pp. 103–104 (2007)
14. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. Parallel Computing 38(3), 157–174 (2012)
15. Lee, L.: Measures of distributional similarity. In: Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics. pp. 25–32. Association for Computational Linguistics (1999)
16. Manouselis, N., Drachsler, H., Vuorikari, R., Hummel, H., Koper, R.: Recommender systems in technology enhanced learning. In: Recommender systems handbook, pp. 387–415. Springer (2011)
17. Michael, M., Moreira, J.E., Shiloach, D., Wisniewski, R.W.: Scale-up x scale-out: A case study using nutch/lucene. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. pp. 1–8. IEEE (2007)
18. Shani, G., Gunawardana, A.: Evaluating recommendation systems. In: Recommender systems handbook, pp. 257–297. Springer (2011)
19. Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering 13(2), 22–30 (2011)
20. Wu, E., Liu, Y.: Emerging technology about gpgpu. In: Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on. pp. 618–622. IEEE (2008)