Automated GPU Support in LuNA Fragmented Programming System

Belyaev Nikolay[1] and Vladislav Perepelkin[1,2]□

[1]Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Novosibirsk, Russia
bl0ckzer01@gmail.com, perepelkin@ssd.sscc.ru
[2]National Research University of Novosibirsk, Russia

Abstract. The paper is devoted to the problem of reduction of complexity of development of numerical parallel programs for distributed memory computers with hybrid (CPU+GPU) computing nodes. The basic idea is to employ a high-level representation of an application algorithm to allow its automated execution on multicomputers with hybrid nodes without a programmer having to do low-level programming. LuNA is a programming system for numerical algorithms, which implements the idea, but only for CPU. In the paper we propose a LuNA language extension, as well as necessary run-time algorithms to support GPU utilization. For that a user only has to provide a limited number of computational GPU procedures using CUDA, while the system will take care of such associated low-level problems, as jobs scheduling, CPU-GPU data transfer, network communications and others. The algorithms developed and implemented take advantage of concerning informational dependencies of an application and support automated tuning to available hardware configuration and application input data

Keywords: Hybrid multicomputers · GPGPU · Parallel programming automation · Fragmented programming · LuNA system.

1 Introduction

When implementing large-scale numerical models on a supercomputer one can significantly improve performance by utilizing both CPUs and GPUs available. Unfortunately, development of such a program is often problematic due to necessity to distribute computational load between CPUs and GPUs, organize data transfer and computations' synchronization. The distribution depends on relative performance of CPUs and GPUs, RAM available, network topology and other architectural peculiarities of given hardware. Implementation of such a distribution is usually troublesome and requires skills in system parallel programming, thus impeding numerical programs development.

Despite the fact that such system programming skills are not expected from application programmers, their involvement is still necessary, because efficient workload distribution problem is far from being solved in general case. In particular, it requires

an understanding of application's data and computations structure and sometimes even understanding of peculiarities of the numerical model implemented (see [1] for an example).

Automation of construction of numerical parallel programs, which efficiently utilize available hardware, is a powerful way to hide the data distribution programming problem from application programmers, thus simplifying numerical programs development. Nowadays there are different systems and tools, aimed at simplifying GPU utilization.

OpenCL [2], for example, is an open standard and a library to support "kernel" development, which can be executed on CPU, GPU or FPGA. OpenCL employs a C-like language to define a kernel. Computational device is selected automatically, based on static analysis and profiling [3]. OpenCL is still a low-level programming tool, where a programmer has to program control manually. OpenCL also does not concern data locality of the application.

OpenACC [4] offers compiler directives to denote "GPU parts" and an API (Application Programmer Interface) to invoke them or transfer data. OpenACC does not concern application data locality, does not balance workload and only supports shared memory systems. DVMH [5] is similar to OpenACC, it allows tuning workload distribution for hybrid multicomputers, but does not provide dynamic load balancing

Charm++ [6] is a platform-independent programming system with a compiler and a run-time system. Charm++ program consists of "chares", which can execute simultaneously and interact with each other. A chare can be assigned to GPU or CPU by a run-time system depending on the strategy, chosen by a programmer.

It can be concluded, that different systems provide some automation of GPU usage, but either for a particular case, or at cost of a significant involvement of the programmer. This is caused by peculiarities of models these systems employ.

A programming system LuNA [7] is being developed in Institute of Computational Mathematics and Mathematical Geophysics SB RAS. LuNA is aimed at automation of numerical parallel programs construction and consists of LuNA language, compiler and a run-time system. LuNA system was chosen for this work, because it is designed for automation of tuning program to hardware resources, which makes is useful to examine algorithms of CPU-GPU load distribution algorithms. This paper is devoted to an attempt to provide automated GPU support for LuNA system.

2 LuNA-Program

In LuNA an application program is represented as a set of computational fragments (CF) and a set of data fragments (DF). Each DF is an aggregated immutable piece of data (say, a subdomain of a numerical mesh at given time step or iteration). Each CF is an operation on DFs, which takes a number of DFs as inputs and produces values of a number of output DFs. Each CF is implemented by a conventional sequential procedure without "side-effects". LuNA-program consists of two parts: a number of sequential procedures in C++ and a description of sets of CFs and DFs in LuNA lan-

guage. LuNA compiler translates programs into an internal representation, executable by LuNA run-time system.

3 CPU and GPU Workload Distribution Algorithm with Automatic Data Refragmentation.

The problem of workload distribution is formulated as follows. For each CF a device (CPU or GPU) must be assigned to be executed on. The goal is to reduce overall application execution time, mainly by providing load balance of available devices and by saving CPU-GPU data transfer "bottleneck."

The proposed algorithm is based on the Rope-of Beads [8] (RoB) algorithm, employed in LuNA system. In the RoB algorithm each CF and DF has a number n assigned ($0 \leq n < L$), where L is a parameter of the algorithm. Number n is called coordinate on the [0; L) segment. More than one fragment can share the same coordinate. The segment [0; L) is split into a number of sub-segments, one sub-segment for each computational node. All the fragments, mapped to a sub-segment of a node, are considered to be assigned to the node. Dynamic load balancing is possible through re-splitting the segment, causing CFs and DFs to migrate if their assignment has changed.

In the proposed algorithm an additional split within one node is proposed. A sub-segment of a node is split into three new parts. The fist part corresponds to CPU(s) of the node (the "CPU part"), the last part corresponds to GPU (the "GPU part"), and the middle part corresponds to fragments, stored in CPU memory, but executed on GPU (the "drag-through part"). For the drag-through part once a CF has to be executed, its input data are copied to GPU, then the CF is executed, and its output DFs are transferred to CPU, releasing occupied GPU memory (this allows running on GPU more CFs than fit in its memory). DFs of GPU and CPU parts never leave their devices in order to optimize CPU-GPU connection usage.

The reasoning behind this splitting a sub-segment into three parts is the following. To save GPU-CPU traffic, each of the devices should have its part of computations (CPU and GPU parts). In order to achieve load balance GPU may require more workload, than its memory can hold. To handle this case the drag-through part of the sub-segment is introduced. The drag-through workload occupies CPU memory, but is transferred in smaller portions to GPU for computations (trading off CPU-GPU bandwidth against CPU or GPU idle time). Although it is unclear, what proportions of parts 1, 2 and 3 would be the best for certain application and hardware, the optimum can be searched for (in particular, one or two parts can degenerate).

To reduce run-time system overhead, bound with number of fragments, static data refragmentation is suggested to be combined with the proposed algorithm. The refragmentation is performed as follows: All the GPU fragments (GPU part) are merged into one, all the CPU fragments (CPU part) are resplit into a number of fragments equal (or proportional) to the number of CPU cores, and the drag-through part is resplit into a number of fragments (portion size), which is a parameter of the proposed algorithm. Such refragmentation requires, that involved C++ procedures fit the

“merge requirements”, i.e. processed domain size must be a parameter of the procedures, which is annotated in code by the programmer.

The proposed algorithm concerns informational dependencies and data structure of the application algorithm (this is inherited from the original RoB algorithm). It can be tuned to properties of an application and hardware configuration using the parameters of the proposed algorithm. The parameters can be defined automatically on the basis of static analysis, hardware benchmarking and/or application profiling, but this is out of scope of the paper. The drawback of the proposed algorithm is the 1D refragmentable decomposition requirement.

4 Testing

The proposed algorithm was implemented as a part of LuNA programming system. To study performance characteristics of the algorithm a number of tests was performed. All the tests were conducted on single computing node with 2×Xeon 5670 (3 GHz) CPUs and GPU Nvidia Tesla M 2090. The application tested is a model finite scheme solver, where the number of computations per single data unit is a parameter. This parameter (called *load*) is used to represent different application classes with different volume of computations per data, which is one of the key properties of an algorithm.

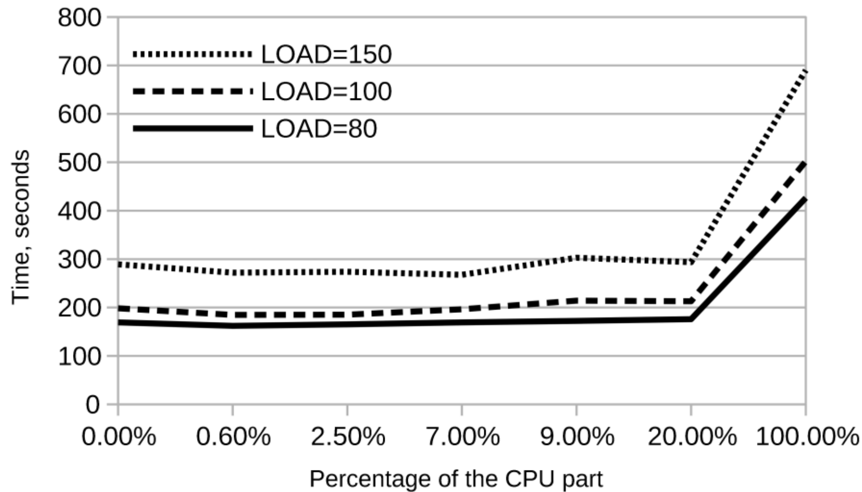

Fig. 1. Program execution time dependency on the amount of computations, assigned to CPU, for different computation-per-data intensity

The first test is devoted to finding an optimal CPU/GPU workload proportion (CPU workload percentage is the X axis). The drag-through parameter is degenerated to zero. It can be seen in Fig. 1, the optimal time is achieved when both CPU and GPU are used, despite the fact, that such execution requires extra CPU-GPU commu-

nications, as compared to CPU-only or GPU-only execution. Note, that the optimal proportion is different for different *load* value.

The second test is devoted to obtaining optimal value of the drag-through parameter. The X axis corresponds to different value of the parameter. It can be seen from Fig. 2 that optimal drag-through parameter is non-zero, which is an evidence of usefulness of the dragging-through part of the proposed algorithm. It also can be seen, that the optimal value of the parameter depends on the *load* parameter. It means, that different applications would require different value of the drag-through parameter.

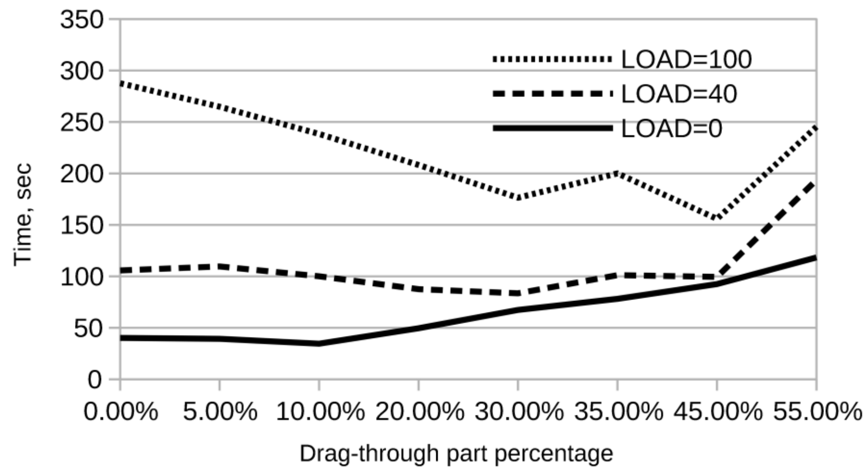

Fig. 2. Program execution time dependency on amount of the “middle-part” DFs for different computation-per-data intensity.

It is worth mentioning, that during the testing the absolute performance achieved is close to that of manually developed programs, which means that the conclusions made are essential to the proposed algorithm, and are not significantly affected by foreign factors, such as LuNA run-time system overhead.

5 Conclusion

An algorithm to distribute workload to CPUs and GPUs of a multicomputer is proposed. The algorithm possesses parameters, capable of tuning to application and hardware peculiarities to reduce program execution time. The algorithm was implemented as a part of LuNA system and performance tests were performed. The tests showed that the algorithm proposed allows automated efficient usage of hybrid (GPU+CPU) computing nodes of a multicomputer. The tests also showed, that the parameters of the proposed algorithm are essential.

Future work supposes solution of the problem of automatic (or at least automated) definition of parameters of the algorithm to allow LuNA tune to given hardware configuration and application peculiarities automatically.

References

1. Kraeva, M.A., Malyshkin, V.E. Assembly Technology for Parallel Realization of Numerical Models on MIMD-Multicomputers. In the Int. Journal on Future Generation Computer Systems, Elsevier Science. Vol. 17, No. 6, pp 755–765 (2001)
2. https://www.khronos.org/opencl/ (accessed May 2017)
3. Wen, Y., Wang, Z., O'Boyle, M. F. P. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. 21st International Conference on High Performance Computing (HiPC). pp. 1–10 (2014)
4. http://www.openacc.org/ (accessed May 2017)
5. Bakhtin, V.A., Chetverushkin, B. N., Krukov, V.A., Shilnikov, E. V. Extension of the DVM parallel programming model for clusters with heterogeneous nodes. DOKLADY MATHEMATICS, Moscow: Pleiades Publishing, Ltd, Vol. 84, Issue 3, P. 879-881 (2011)
6. http://charm.cs.illinois.edu/research/charm (accessed May 2017)
7. Malyshkin, V.E., Perepelkin, V.A. LuNA fragmented programming system, main functions and peculiarities of run-time subsystem. In: Proc. of the 11th conference on parallel computing technologies, LNCS 6873. Springer, New York, pp 53–61 (2011)
8. Malyshkin, V.E., Perepelkin, V.A., Schukin, G.A. Distributed Algorithm of Data Allocation in the Fragmented Programming System LuNA. In Proc 13th International Conference on Parallel Computing Technologies. LNCS 9251. Springer. pp. 80–85. DOI: 10.1007/978-3-319-21909-7_8 (2015)