UDC 004.4’24

DOROSHENKO A.,
BEKETOV O.,
YATSENKO O.

LOOP PARALLELIZATION AUTOMATION FOR GRAPHICS PROCESSING UNITS

A technology that allows extending GPU capabilities to deal with data volumes that outfit internal GPU’s memory capacity is proposed. It involves loop tiling and data serialization and can be applied to utilize clusters consisting of several GPUs. Applicability criterion is specified and a semi-automatic proof-of-concept software tool is implemented. The experiment to demonstrate the feasibility of the proposed technology is described.

Keywords: CUDA, general-purpose computing on graphics processing units, loop optimization, parallelization methods.

1. Introduction

The complication of computing problems and improvements of hardware, on the other hand, resulted in the appearance of multiprocessor systems, that are naturally suited to employ parallel algorithms. Nowadays, many new parallel platforms are emerging, one of the most popular is the direction using graphics processing units (GPUs) as general-purpose computing devices. As for now, GPUs have the highest level of parallelism compared to other computational devices. These devices can provide big performance boosts. However, efficient programming of GPUs is not an easy task. Developers should know many technical details about GPU architecture.

The interest in graphics accelerators is ever-growing due to their superior performance compared to conventional processors, availability, and low energy consumption. However, the development of appropriate tools is still a problem. In particular, we consider the problem of modeling parallel systems with heterogeneous components that contain both CPU and GPU. Along with spreading of GPGPU technology [1] that allows the employment of graphics accelerators for solving computational tasks, new challenges arise. As far as GPU is not a standalone device and is managed by a host operating unit, it should be considered within the context of heterogeneous computational platforms. Composing programs for such platforms demands knowledge in architecture and specific programming tools. Generally, concurrent software development passes through the stage of sequential implementation that becomes a starting point for further platform-dependent and hardware environment specific implementations.

Existing automatic code parallelizing tools [2–5] don’t account for the limited amount of GPU’s onboard memory space while real-life problems demand huge amounts of data to be processed. To embrace those cases of massive computational tasks that involve large amounts of data, we propose a technique that provides the ability to rip the loop and to split the data and calculation operations.

This paper considers the problem of automated parallelizing transformation of embedded loops for the target platform of a computing system of heterogeneous architecture that includes graphic processors. A technology for a semi-automated parallelization method of nested loops for graphics processors is proposed. A technique that allows to extend GPU capabilities to deal with data volumes that outfit internal GPU’s memory capacity is revealed and proved. The technology involves loop tiling and data serialization and can be applied to utilize clusters consisting of several GPUs [6, 7].

A formal transformation of the computation loop nest that allows the transition from sequential algorithm to parallel is illustrated on solving linear systems with Cholesky decomposition method, matrix multiplication, and *N*-body problems.

2. A formal model of the loop and parallelization technique

Loop parallelization is a long-standing problem of computational programming. Loops give a fair parallelization opportunity for numerous scientific modeling problems that involve numerical methods. This section introduces the idea of loop transformation.

Let us consider a set of finite sets I_k, $0 \le k \le N$, each set consists of an ordered set of elements, i.e.

$$I_k = \{i_{k,0} \prec_k i_{k,1} \prec_k \ldots \prec_k i_{k,\#I_k-1}\},$$

where $\prec_k$ is a partial order over the set I_k, $\#I_k$ indicates the number of elements of the set I_k.

The for loop operator

$$for \;\; i_k \in I_k : S(i_k)$$

is a form of notation of the following sequence

$$\{S(i_{k,0}), S(i_{k,1}), \ldots, S(i_{k,\#I_k-1})\},$$

where $S(i_k)$ is an expression containing the dependence on the loop iterator i_k.

Let D be a set of data with a subjective mapping $T : D \mapsto D$ over it. Let us also introduce subjective mappings $p : I \mapsto D$ and $q : I \mapsto D$.

Let us consider the pairs of elements of the set D of the form $(a, v) \in D^2$, for those elements $a \in D$, for which there exist preimages of the set I under mappings p and q, respectively. Sets of all such pairs are denoted as

$$P = \{(a, v) \mid \exists \bar{i} \in I : p(\bar{i}) = a \in D \mid v = T(a)\} \subset D^2$$

and

$$Q = \{(b, w) \mid \exists \bar{i} \in I : q(\bar{i}) = b \in D \mid w = T(b)\} \subset D^2.$$

Hereinafter, the pair $(a, v) \in D^2$ will be referred to as a data cell, $a \in D$ a cell address, and $v \in D$ a cell value. Mapping T provides a cell value by cell address. The set of all cells involved in the calculations is called memory.

Let $F : I \times D \mapsto D$ is the transformation mapping of the data set. Let us consider the nested loop of the following form:

$$\begin{array}{l} for \;\; i_N \in I_N : \\ \quad for \;\; i_{N-1} \in I_{N-1} : \\ \quad\quad \ldots \\ \quad\quad\quad for \;\; i_0 \in I_0 : \\ \quad\quad\quad\quad T \cdot q(\bar{i}) := F(\bar{i}, T \cdot p(\bar{i})), \end{array} \qquad (1)$$

where symbol $\bar{i} = \{i_0, i_1, \ldots, i_N\} \in I_0 \times \ldots \times I_N = I$ denotes a vector of iterators. The number of operators involved in the loop is called nesting depth.

Nesting depth for the loop (1) is equal to $N+1$. Sets P and Q are called a set of initial data and a set of final data of the loop (1), respectively.

The loop iteration is a calculation that executes a loop for a certain fixed value of the vector of iterators $\bar{i}$, which acquires value from the set $I_0 \times \ldots \times I_N$. We assume that iterations are independent by data:

$$\forall \bar{i}, \bar{j} \in I, \bar{i} \neq \bar{j} : p(\bar{i}) \neq q(\bar{j}), q(\bar{i}) \neq q(\bar{j}), \quad (2)$$

i.e. no iteration of the loop changes the initial data of other iterations, and different iterations do not change the values of the same cells.

Having numbered the elements of the sets I_k according to the lexicographic order, let us proceed to the loop with a linear counter by performing the following substitution:

$$for \;\; i_N \in I_N \mapsto for \; 0 \le i_n < \# I_n,$$

where counter i_n changes with a unit step. Let us perform decomposition of the loop nest. To achieve this, at first we perform the following substitution for each for operator:

$$\begin{aligned} &for \; 0 \le i_n < \# I_n \mapsto \\ &for \; 0 \le s_n < S_n : \\ &\quad for \; s_n \cdot L(\# I_n, S_n) \le i_n < \min((s_n + 1) \cdot L(\# I_n, S_n), S_n), \end{aligned}$$

where

$$L(a,\ b) = \left\lfloor \frac{a}{b} \right\rfloor + 1 - \delta_{0,\ a \bmod b},$$

$\lfloor \cdot / \cdot \rfloor$ denotes the integer part of a quotient, δ is Kronecker symbol, S_n is the desired number of steps to subdivide the loop, $1 \le S_n < \# I_n$. This transformation is commonly known as loop tiling [8]. After subdivision of subloops and regrouping, the loop takes the form, in which an internal loop is similar to the initial loop but of a reduced scale. We keep internal $N+1$ loops intact and group outer loops:

$$\begin{aligned} &for \quad 0 \le e < \prod_{i=0}^{N} S_i : \\ &\quad \vec{i} = g(e); \end{aligned} \quad (3)$$

Here $g(e)$ is a mapping that restores the vector of counters $\vec{i}$ and is constructed in the following way:

$$\begin{gathered} g_0(e) = e \bmod S_0, \\ g_k(e) = \left\lfloor \left(e - \sum_{j=k+1}^{N} g_j(e) \prod_{l=0}^{j-1} S_l \right) \Big/ \prod_{j=0}^{k-1} S_j \right\rfloor, \; 0 < k < N, \\ g_N(e) = \left\lfloor e \Big/ \prod_{j=0}^{N-1} S_j \right\rfloor, \\ 0 \le e \le \prod_{k=0}^{N} S_k . \end{gathered}$$

The obtained loop (3) maintains the sequence of the vector of counters equal to the sequence produced by the initial loop (1).

Let's denote the inner $N+1$ loops of the cycle (2) along with $g(e)$ as a $kernel(e)$. We intend to delegate the *kernel* execution to GPU and to run it concurrently thus diminishing the depth of the inner loop nest. As the GPU's memory space is isolated from the host's device one, we introduce *serialize* operation that is to prepare the input data required to perform calculations for the step *e* and *deserialize* operation to store the output data processed by GPU. The further implementation of these procedures is out of our scope and depends on the particular problem. Finally, we get:

$$\begin{aligned} &for \quad 0 \le e < \prod_{i=0}^{N} S_i : \\ &\quad serialize(e, inputData, dataPull); \\ &\quad transfer2device(inputData); \\ &\quad kernel(e, inputData, outputData); \\ &\quad transfer2host(outputData); \\ &\quad deserialize(e, outputData, dataPull); \end{aligned} \tag{4}$$

Iterations of the loop (4) can be distributed over concurrently running threads through involving several additional data exchange buffers. This approach could be applied to any distributed memory computational system, e.g. GPU cluster or heterogeneous cluster of any other computation empowered devices. To preserve equivalence in a sense of output results equality for the same given input data, Bernstein's [9] conditions must be met. This roughly means that iterations should not overwrite the other's iterations input data and should store their output data apart. The set of S_k ($0 \le k \le N$) is the transformation's tunable parameters that are chosen in a way to satisfy Bernstein's conditions and to optimize processing time that is to find a trade-off on time spent on data preparation, transfer, and kernel execution. These timings depend on the input and output data load size which is restricted by the total available amount of GPU's memory and hardware configuration parameters, such as input and output memory transfer rate and GPU compute capabilities.

The loop of the initial form is a form of notation of the data transformation sequence, given by the mapping F. However, a loop in this form is not suitable for execution by a parallel computing device, because for operator defines a sequential computational procedure for individual iterations. To perform the parallelization of the loop, it is necessary to properly distribute the iterations and corresponding data between the threads. The loop in the parallel form will be equivalent to the original loop if the above condition (2) holds. Let us prove this statement.

Let there is a procedure P performing data operations. Let us classify the memory cells used by this set of commands according to the mode of their use:

1) W, $W \subset D$ denotes a set of cells which are only read, i.e. $\forall d \in W$:

$$\forall \bar{i} \in I : q(\bar{i}) \ne d; \tag{5}$$
$$\exists \bar{i} \in I : p(\bar{i}) = d; \tag{6}$$

2) X, $X \subset D$ is a set of cells which are only written, i.e. $\forall d \in X$:

$$\exists \bar{i} \in I : q(\bar{i}) = d; \tag{7}$$
$$\forall \bar{i} \in I : p(\bar{i}) \ne d; \tag{8}$$

3) Y, $Y \subset D$ is a set of cells which are read and then written:

$$\forall d \in Y : \exists \bar{i}, \bar{j} \in I, \bar{i} \prec \bar{j} : \; p(\bar{i}) = d, \; q(\bar{j}) = d; \tag{9}$$

4) Z, $Z \subset D$ is a set of cells which are written and then read:

$$\forall d \in Z : \exists \bar{i}, \bar{j} \in I, \bar{i} \prec \bar{j} : \; q(\bar{i}) = d, \; p(\bar{j}) = d .$$

Let there are three procedures P_1, P_2 and P_3, the corresponding indices denote their sets of cells. Consider two algorithms that perform these procedures: the first calls the procedures sequentially one after the other, and the second performs the first two procedures simultaneously, and then the third. According to Bernstein [9], the following conditions must be met in order for the results of sequential and parallel algorithm calculations to coincide:

$$\left(W_1 \cup Y_1 \cup Z_1\right) \cap \left(X_2 \cup Y_2 \cup Z_2\right) = \varnothing,$$

that is, those memory cells that are read by the first procedure do not intersect with those cells that are written by the second procedure; condition symmetric to the previous one

$$\left(X_1 \cup Y_1 \cup Z_1\right) \cap \left(W_2 \cup Y_2 \cup Z_2\right) = \varnothing,$$

and

$$X_1 \cap X_2 \cap \left(W_3 \cup Y_3\right) = \varnothing,$$

that is, those cells that are used by both the first and the second recording procedures simultaneously will not be subsequently read without preliminary rewriting. Summarizing Bernstein's conditions for a set of procedures P_i, $0 \le i \le C$, we obtain:

$$\begin{gathered} \forall 0 \le i, j \le C : \\ (W_i \cup Y_i \cup Z_i) \cap (X_j \cup Y_j \cup Z_j) = \varnothing, \\ (X_i \cup Y_i \cup Z_i) \cap (W_j \cup Y_j \cup Z_j) = \varnothing, \\ X_j \cap X_j \cap (W_C \cup Y_C) = \varnothing. \end{gathered}$$

On the one hand,

$$d \in (W \cup Y \cup Z) \Leftrightarrow \bar{i} \in I : p(\bar{i}) = d .$$

On the other hand,

$$d \in (W \cup Y \cup Z) \Leftrightarrow \bar{i} \in I : q(\bar{i}) = d .$$

Therefore, the first two Bernstein conditions are equivalent to the following:

$$\forall \bar{i}, \bar{j} \in I : p(\bar{i}) \ne q(\bar{j}) .$$

Since condition (8) requires

$$\forall \bar{i}, \bar{j} \in I : p(\bar{i}) \ne p(\bar{j}),$$

then

$$X_i \cap X_j = \varnothing,$$

and therefore the condition (9) also holds. Thus, conditions (7), (8) are equivalent to the combined conditions of Bernstein and ensure that the memory status of the device does not depend on the order of operations. Therefore, the statement is true.

Example. As an illustration, consider parallelizing Cholesky decomposition. Cholesky decomposition method [10] of solving linear equation systems replaces the initial system with a square matrix by two systems with lower and upper triangular matrices. Cholesky method is used for solving systems with symmetric positive-definite matrices. Such systems may arise as a result of the least square method for example. LU-factorization method is more common, and it results in solving triangular systems as well. The following loop solves the system $Ax = B$, where A is a lower triangular matrix of size $N \times N$ with Gaussian elimination method [11]:

$$\begin{aligned}
&for\ \ i,\ 0 \le i < N:\\
&\quad s = 0;\\
&\quad for\ \ j,\ 0 \le j < i:\\
&\qquad s := s + A_{i,j} \cdot X_j;\\
&\quad X_i := (B_i - s) / A_{i,i}.
\end{aligned}$$

This form of the algorithm does not satisfy the property of iterations independency, as *i*-th iteration of the outer loop depends on X_{i-1} value obtained on the previous loop iteration. The properties don't comply for the inner loop as well as the iterations are linked through s variable. Moreover, this algorithm is not scalable, as splitting the matrix makes it rectangular and requires another solution algorithm. These drawbacks of the transformed algorithm would not preserve the correctness of the results of the calculation. So let's consider the algorithm of the following more convenient form:

$$\begin{aligned}
&for\ \ i,\ 0 \le i < N:\\
&\quad X_i := B_i / A_{i,i};\\
&\quad for\ \ j,\ i \le j < N:\\
&\qquad B_j := B_j - A_{j,i} \cdot X_i.
\end{aligned}$$

By introducing the parameter B_j, the inner loop of the previous algorithm was deprived of the dependency on the parameter s and thus created an opportunity to parallelize the inner loop by $N - i$ independent threads. Although this form of the algorithm maintains a dependency on the parameter X_i, and the conditions (7), (8) are not satisfied, as the matrix of the system remains square, the algorithm doesn't change and therefore acquires scalability. The transformation of the loop is valid if synchronization by the iterator i is held. The parallelization of an upper triangular matrix can be carried similarly.

3. Program execution flow

In this section, we observe the execution flow of a program parallelized with the help of the proposed method. Consider the computing node that consists of one multicore CPU and one GPU. Modern GPUs support direct memory access technology thus allowing to carry out data transfer and kernel execution asynchronously. To optimize the data exchange process, dual buffering is involved. Four buffers at both host and device sides are involved — two for the input and two for the output data exchange. On the host side, computations are made by two threads executing $kernel$, $serialize$ and $deserialize$ procedures simultaneously. One of them serializes the input data and fills the input data

buffer, then transmits the buffer to the GPU and launches the kernel, and the other receives output data buffer from GPU and deserializes it. Besides the calculations, GPU carries out bidirectional data transfers through the asynchronous data transfer mechanism. Computations are performed in three stages — initial, cyclic and finalizing.

At the starting point, data buffers are empty and GPU waits for the data transfer. It doesn't matter what thread will carry out the initial step, as all the operations are executed sequentially and asynchronous transfer mode is not involved. At the initial step, CPU serializes input data buffers of the first two iterations and transfers the buffer containing the first iteration data to the accelerator. After the initial step, the cyclic stage starts. The execution flow is shown in Fig. 1. The iteration number is given after the buffer's name. One step of the loop is divided into odd and even parts. Both odd and even parts of the first step skip deserialization as the host output buffers are empty yet. At the odd part of the first step, the accelerator-to-host transfer is omitted too. Meanwhile, the accelerator performs computations over a current buffer, host threads fetch data buffer from a previous step, deserialize penultimate step buffer, send input data buffer, and serialize buffer for the next step. In one step, two kernel launches are executed. After each part (odd or even) is finished, the processes are synchronized. Two final steps depend on the actual number of kernel launches. If the number of kernel launches is odd, the final step of the cyclic part excludes an even part and does not involve serialization and host-to-accelerator transfer, and the even part of the penultimate step skips serialization. Otherwise, if the number of launches is even, the last loop step is full, but the even part of the final step omits serialization. The final step deserializes output data buffer transferred at the last loop step and then fetches and deserializes the final output data buffer consequently finishing the computations.

4. Application of the proposed technique for constructing CUDA programs

In this section, we illustrate the application of the proposed technique to matrix multiplication and N-body problems. The time measurements were collected on the hardware system composed of Intel Core i5-3570 CPU (4 cores, 3.8 GHz) with 16 GB of host memory and NVIDIA Tesla M2050 GPU (3 GB of global memory, 384 bits memory bandwidth, connected through PCIe2.0 x8) running Ubuntu 16.04 host operating system.

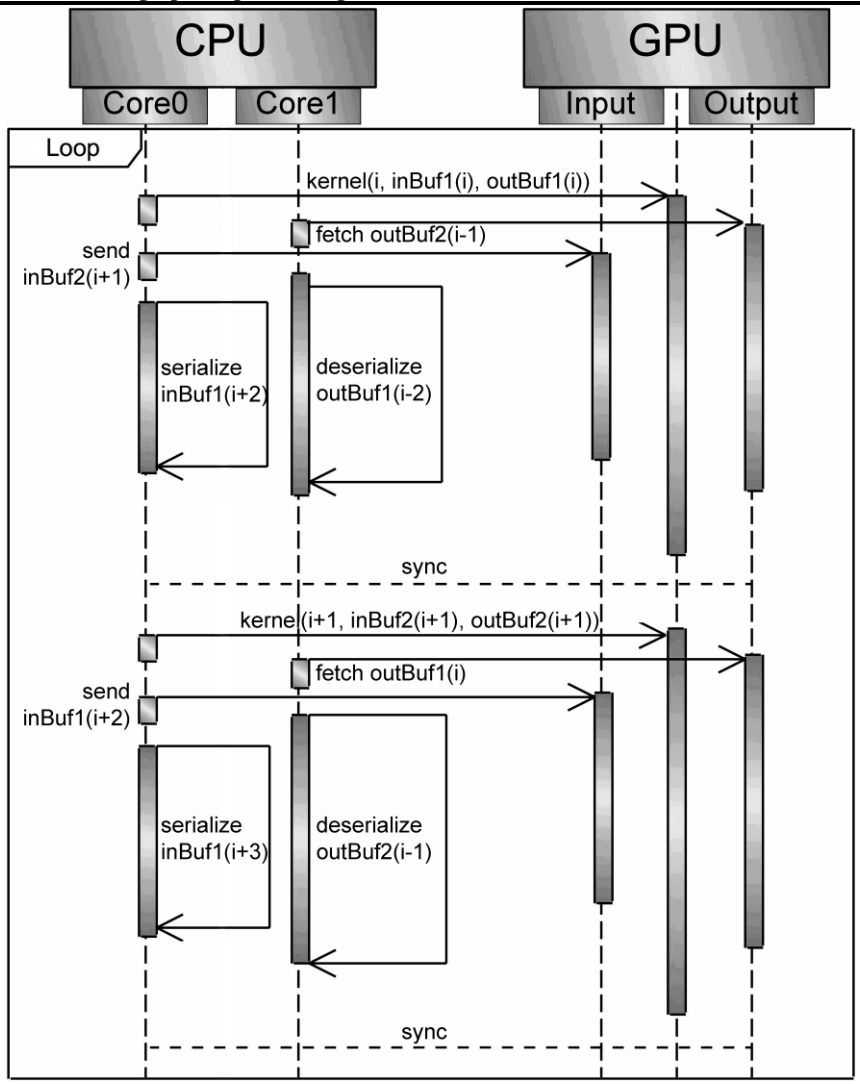

Fig. 1. Execution flow diagram of the cyclic stage of the concurrent program for the system of one accelerator and two control flow threads with four data exchange buffers.

A semi-automatic source-to-source code transformation tool based on the TermWare rewriting system [12] aiming to assist in constructing a new concurrent program was implemented. It takes the initial loop marked with pragmas, applies the transformations (3), and provides a template of the code of a new loop to be substituted. The remaining actions include serialization and deserialization routines implementation; the kernel could be implemented as well as generated by another tool and adapted in place.

The algorithm of the initial sequential matrix multiplication program involved a three-dimensional loop nest. It was transformed using the proposed technique and C-to-CUDA compilers PPCG and Par4All. Both of the programs generated by PPCG and P4A showed comparable results. After applying the slicing technique, the initial matrices were split into submatrices. The internal loop subdivision parameter S_0 was set to 1, the roles of parameters S_2 and S_1 is adjusting submatrices width. The schema with double data exchange buffering and two CPU threads were used. Even not involving GPU, adjusting the slicing number allowed to reach about 12 times acceleration over the initial loop due to CPU caching. For the GPU implementation, the parameterized PPCG generated kernel was used; the source codes of the constructed matrix multiplication program are available at GitHub [13]. The chart in Fig. 2 shows the constructed program's timings and the timings of the program obtained with PPCG relatively to the matrix dataset size.

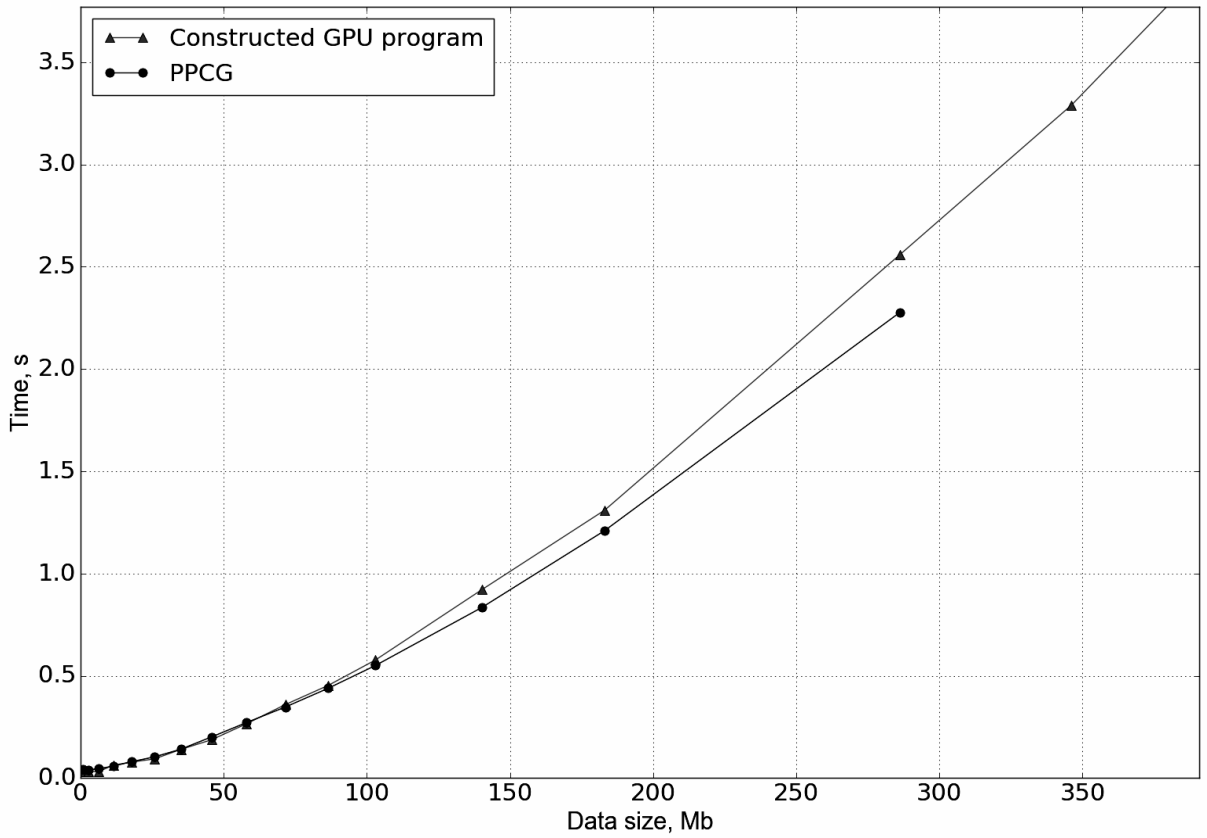

Fig. 2. The dependency of the execution time on the size of the input data for the concurrent PPCG-generated and constructed matrix multiplication programs

The relative acceleration of about 430 times in comparison to the sequential program executed on the CPU for the datasets of square matrices of the size of 5000×5000 single-precision floating point numbers was reached. It could be seen from the chart, that the PPCG generated program has achieved maximum data set size less than 300 Mb that is 10% of GPU's global memory available. This is because PPCG is limited to static memory usage, thus blocking to link programs with too large static arrays.

It is worth mentioning that involving two CPU threads is excessive as the part of serialization and deserialization is negligible compared to GPU kernel computation time, which can be seen from the GPU execution profile given in Fig. 3. Thus, involving just one thread instead won't decrease performance substantially, however, two concurrent threads are required to avoid gaps in kernel launches and to gain maximum performance from the GPU.

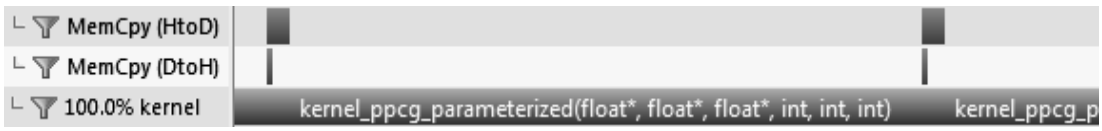

Fig. 3. The fragment of the profile of matrix multiplication execution

Another application examined was a predictor routine from the N-body problem with the predictor-corrector time-iterative [14] algorithm. The model of the system consists of a set of particles that interact pairwise PPCG applying caused a slowdown effect and led in about 500 times decrease in performance in comparison to the sequential CPU implementation. As for the constructed program with a self-implemented kernel not involving shared memory usage, the relative CPU to GPU acceleration at the selected data size range reached 13 times. The plot in Fig. 4 shows the dependency of sequential CPU and transformed GPU programs execution time on the data size that is scaled by the alteration of the number of particles N. The timings are measured for the one time-step of the prediction routine. The memory limit was not reached as it would take approximately 30 years to process one time-step of the algorithm

for a fully loaded GPU that was used in the experiment, however, the applicability of the technique is confirmed.

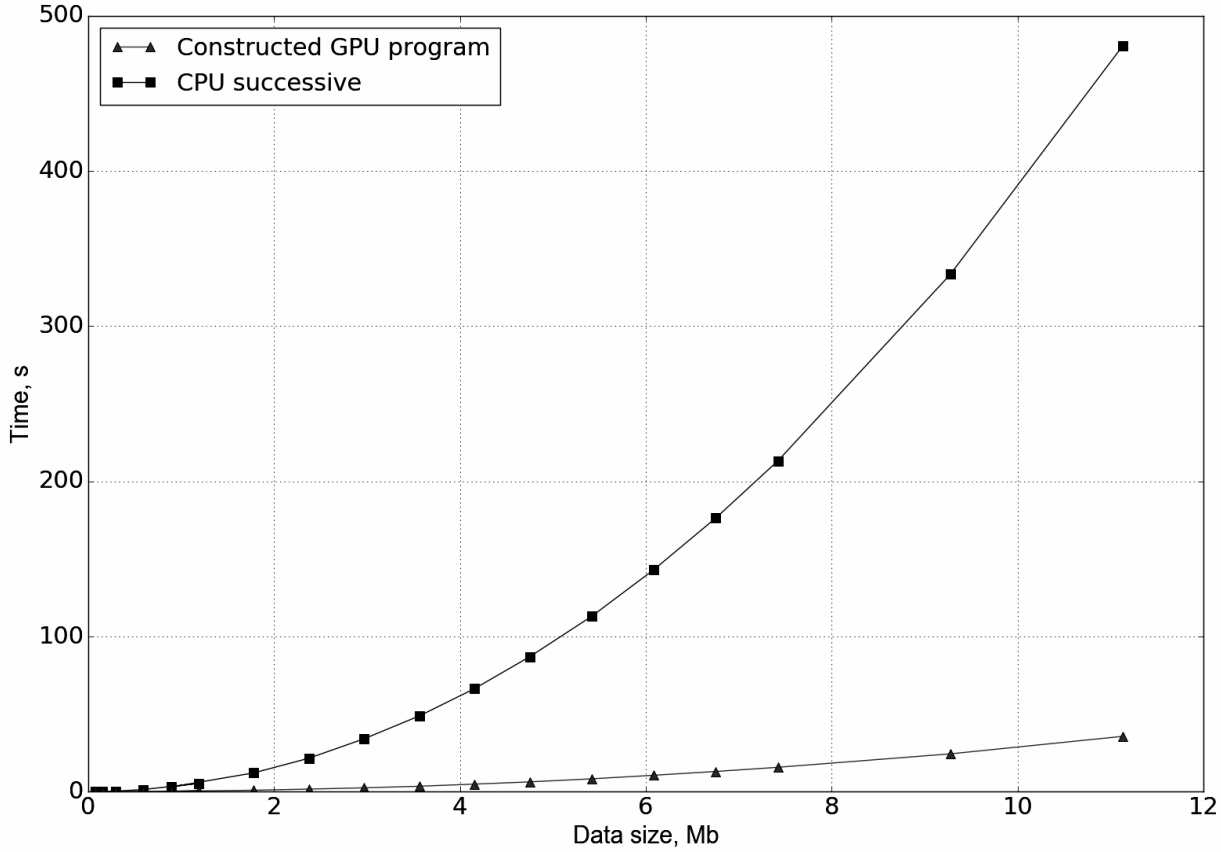

Fig. 4. The dependency of the execution time on the size of the input data for the sequential and constructed concurrent N-body programs

Thus, the difference in the constructed multiplication and N-body programs consists in the kernel, serializer, and deserializer implementation while the control flow structure remains identical.

5. Conclusions

A technology for semi-automated parallelization of nested loops for GPUs was proposed. The advantage of the proposed technology is the ability to process data volumes that exceed the GPU memory size and simultaneously use multiple GPUs. The usage of the technology is illustrated by the development of CUDA programs for solving matrix multiplication and N-body problems. An assistant semi-automatic code transformation tool was implemented. Further work is associated with developing unified methods and tools in loop parallelization.

References

1. Harris M. J. Real-time cloud simulation and rendering : a dissertation for a Ph.D. degree in the department of computer science. Chapel Hill, NC : University of North Carolina, 2003. 151 p.
2. PIPS: Automatic Parallelizer and Code Transformation Framework. PIPS4U : website. URL: http://pips4u.org (accessed: 17.08.2020).
3. Polyhedral parallel code generation for CUDA / Verdoolaege S. et al. ACM Trans. Architec. Code Optim. 2013. Vol. 9, № 4, Art. 54. P. 1–23.
4. Split tiling for GPUs: automatic parallelization using trapezoidal tiles / T. Grosser et al. GPGPU-6 : Proc. 6th Workshop on General Purpose Processor Using Graphics Processing Units, March 16, 2013. New York : Association for Computing Machinery, 2013. P. 24–31.

5. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs / P. Di et al. Proc. 41st International Conference on Parallel Processing, September 10–13, 2012. Washington, D.C. : IEEE Computer Society, 2012. P. 1–12.
6. Automated design of parallel programs for heterogeneous platforms using algebra-algorithmic tools / Doroshenko A., Beketov O., Bondarenko M., Yatsenko O. ICTERI 2019 : Post Proc. 15th Int. Conf. “ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer”, June 12–15, 2019. CCIS. Vol. 1175. Cham : Springer, 2020. P. 3–23.
7. Doroshenko A., Beketov O. Large-scale loops parallelization for GPU accelerators. ICTERI 2019 : Proc. 15th Int. Conf. “ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer”, June 12–15, 2019. Cham : Springer, 2019. P. 82–89.
8. Wolfe M. More iteration space tiling. Supercomputing’89 : Proc. ACM/IEEE Conference on Supercomputing, November 1989. New York : Association for Computing Machinery, 1989. P. 655–664.
9. Bernstein A. J. Analysis of programs for parallel processing. IEEE transactions on electronic computers. 1966. Vol. EC-15, № 5. P. 757–763.
10. Dereniowski D., Kubale M. Cholesky factorization of matrices in parallel and ranking of graphs. Proc. 5th International Conference on Parallel Processing and Applied Mathematics, September 7–10, 2003. Berlin : Springer, 2004. P. 985–992.
11. Gentle J. E. Gaussian elimination. Numerical Linear Algebra for Applications in Statistics. Berlin : Springer, 1998. P. 87–91.
12. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. Fundamenta Informaticae. 2006. Vol. 72, № 1–3. P. 95–108.
13. GitHub repository. github : website. URL: https://github.com/o-beketov/matmul (accessed: 17.08.2020).
14. Aarseth S. J. Gravitational N-body simulations. Cambridge : Cambridge University Press, 2003. 430 p.