Investigation of the RAM access model in a heterogeneous computing system

Aleksander Kolpakov
Department of Electronics and Computer Science
Vladimir State University named after Alexander and Nicholay Stoletovs
Murom, Russia
ORCID: 0000-0001-9328-2331

Aleksey Belov
Department of Electronics and Computer Science
Vladimir State University named after Alexander and Nicholay Stoletovs
Murom, Russia
ORCID: 0000-0002-2072-7042

Yuriy Kropotov
Department of Electronics and Computer Science
Vladimir State University named after Alexander and Nicholay Stoletovs
Murom, Russia
ORCID: 0000-0002-9012-3092

***Abstract*—The issue of creating high-performance computing systems based on heterogeneous computer systems is topical, since the volumes of processed information, calculations and studies with large data sets are constantly increasing. The aim of the work is an experimental study of previously developed models for predicting the performance of heterogeneous computer systems in telecommunications. As a result, the study showed that the developed models allow us to obtain an adequate estimate of the possible time of the algorithm for various parameters of the GPU with some limitations.**

Keywords—graphic processor units, heterogeneous computing systems, parallel calculations

I. INTRODUCTION

One of the most dynamically developing areas in parallel programming at the moment is the use of computer systems with a heterogeneous architecture, in which there are computing devices with different architectures and, accordingly, with different methods of using parallel computing. The most common approach in the design of such systems was the use of graphic video cards or devices based on them as the main parallel calculator. This growing need for solving big problems stimulates research and innovation in the field of parallel computing in general and in the development of methods for graphic processors in particular.

II. RESEARCH AND DEVELOPMENT OF MODELS OF HETEROGENEOUS COMPUTING SYSTEMS BASED ON GRAPHICS PROCESSORS

Modern graphic processors (GPUs) are parallel processors. More precisely, they are known as stream processors because they are capable of performing various functions in the incoming data stream. They represent advanced architectures that are designed for parallel processing of data (primarily graphic). They are currently extremely powerful programmable processors, MIMD architecture capabilities with some limitations.

As technology, languages, and hardware evolved, researchers were able to leverage the added flexibility of GPUs when deploying non-graphical applications to the GPU (GPGPU), especially when processing images. A more detailed history of the development of GPGPU is presented in [1].

A further development momentum was the emergence of CUDA, the NVIDIA C-based development environment for GPGPUs. CUDA allows developers unfamiliar with graphical programming to write code that can be executed on the GPU. CUDA provides the necessary abstractions for the developer to write multi-threaded programs with little or no knowledge of the graphics APIs. Since then, many implementations of parallelized applications have been developed for GPUs, many of which offer significant acceleration compared to sequential implementations on the processor.

A model of the upper bound for the running time of an algorithm on a graphics processor in a CPU-GPU environment is presented in [2] and is based on an abstract PRAM model [3]. The upper estimate of the running time of the algorithm on a graphics processor in a CPU-GPU environment according to this model is calculated according to the formula below

$$T_{GPU}(N) = \frac{N_{HD}(N)}{S_{HD}} + \sum_{i=1}^{B(N)} T_{iG}(N) + \frac{N_{DH}(N)}{S_{DH}} \quad (1)$$

This model does not account for certain features of the graphics processor, such as the size of a warp-to GPU memory access time, etc., however, the GPGPU performance prediction model, which is a combination of known parallel computing models, was given in [4]. Given the complex architecture of the GPU, none of these models is complete, and it takes a combination of them along with a few extensions. The following models were used during development:

1) The PRAM model [3].
2) The BSP model [5].
3) The QRQW model [6].

The final equation of this model is:

$$T(K) = \frac{N_B(K) \cdot N_w(K) \cdot N_t(K) \cdot C_T(K)}{N_C \cdot D \cdot R} \quad (2)$$

All parameters of this model are shown in table 1.

TABLE I. THE LIST OF DEVELOPED MODEL PARAMETERS

Parameter	Description
D	The kernel pipeline depth
N_c	The number of cores per SM
R	GPU Clock
$C_t(K)$	The maximum number of ticks consumed by any thread in the kernel K
N_t	Number of threads in warp = 32
N_w	The number of warps per block
$N_B(K)$	The number of blocks per kernel
K_i	i-th kernel on the GPU
$T(K)$	Time spent by kernel K
$T(P)$	Time spent by program P

The performance of the CUDA kernel can vary greatly with slight changes depending on memory access strategies. Using shared memory can provide up to 20 times better performance than using global memory, and using shared

Copyright © 2020 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR-WS.org/Vol-2667/paper3.pdf

global memory access can lead to 5 times better performance compared to non-coalesced access. Arithmetic operations also require a different number of cycles to perform, for example, operations such as integer summation require 4 cycles, while calculating an integer module takes 48 cycles [7,8]. Any model that does not capture these changes is unlikely to be accurate.

III. USING GPU MEMORY ACCESS PATTERNS TO IMPROVE HETEROGENEOUS COMPUTING SYSTEM PERFORMANCE

Since the access delay when reading data from the global memory of the GPU is up to 200 times higher, than reading data from the registers, providing the most efficient way to access the GPU RAM is crucial to improve the performance of the GPU in particular and the heterogeneous system as a whole. Therefore, optimizing global memory access is becoming the single most important programming factor for the GPGPU architecture.

The GPU global memory reads and writes data in half warp threads (16 threads), which are optimized by the device in just one global memory transaction if certain access requirements are met. For the GTX 280 video card, the following protocol is used to determine the number of transactions used by the halfwarp:

1) Implemented search memory segment, which contains the address inquiry from the active thread with the lowest number. The segment size is 32 bytes for 8-bit data, 64 bytes for 16-bit data, and 128 bytes for 32-, 64- and 128-bit data.

a) if the transaction size is 128 bytes and only the upper or lower half of the segment is used, the transaction size is reduced to 64 bytes;

b) if the transaction size is 64 bytes and only the lower or upper half is used, the transaction size is reduced to 32 bytes.

2) A transaction is in progress, serviced threads are marked as inactive.

3) Repeated until all threads in the halfwarp are serviced.

When more than one thread requests data from addresses, that fall into the same segment, one transaction can satisfy all such threads. This service of multiple requests in a single transaction is called coalescing. Thus, it is obvious that certain GPU memory access patterns will necessarily have a positive effect, while others will gradually increase the delay as the memory addresses requested by the halfwarp threads diverge.

The shared memory of the GPU is divided into memory modules of the same size, called banks, which can be accessed simultaneously by several threads. Thus, any request to read or write to the memory, consisting of n addresses, that fall into n separate memory banks, can be carried out simultaneously, which gives an effective throughput that is n times higher than the throughput of one module. However, if two memory request addresses fall into the same memory bank, a bank conflict occurs and access must be serialized. The GPU divides memory access requests with bank conflicts into as many individual requests without conflicts as necessary, reducing the effective bandwidth by a factor equal to the number of individual memory requests. For GTX280, the size of the warp is 32, and the number of banks is 16. Access to shared memory for warp is divided into one request for the first half of warp and one request for the second half of warp. As a result, there can be no conflict of the bank between a thread belonging to the first half of the warp and a thread belonging to the second half of the same warp.

Shared memory also has a broadcast mechanism that allows you to read a 32-bit word and broadcast it to multiple threads at the same time with one transaction to read from memory. This reduces the number of conflicts in the bank when several threads of the halfwarp are read from the address within the same 32-bit word. More precisely, a memory read request made to several addresses are served in several stages over time – one step every two cycles, serving one conflict-free subset of these addresses per step, until all addresses are served. At each step, a subset is constructed from the remaining addresses that have yet to be served using the following procedure:

1) Select one of the words indicated by the remaining addresses as the translated word.
2) Include in a subset:

a) all addresses that are within the broadcast word;

b) one address for each bank indicated in the remaining addresses.

The space of constant memory is cached, so reading from constant memory is delayed as in the case of reading from global memory only if there is no cache, otherwise a transaction from constant cache is performed. For all threads, halfwarp reads from the constant cache as fast as reads from registers if all threads read the same address. Access time scales linearly depending on the number of different addresses read by all threads.

Texture memory allows to cache data present in global memory. If a cached item is requested, then it is served in a single request. The lack of a cache results in a global memory read operation, which takes much longer.

To compare the access time to different types of memory, 1,000,000 read operations from each type of memory were performed. Access was performed both sequentially and randomly. Tests were performed using the NVIDIA GTX280 GPU. The results are presented in Table 2. As you can see, shared memory provides the best performance, followed by a constant cache, and then a texture cache. Global memory shows the highest latency.

TABLE II. COMPARISON TABLE OF ACCESS TIMES FOR DIFFERENT MEMORY TYPES

Memory type	Serial access, ms	Random access, ms
Global	969	21777
Shared	51	86.6
Constant	35	192
Texture	140	247

IV. EXPERIMENTAL MODELING OF GPU MEMORY ACCESS PATTERNS

For most parallel computing platforms, modeling memory access patterns and their associated costs is the most complex and most important part. The following algorithm is used to experimentally verify statements about the process of accessing memory on the GPU:

Algorithm 1 Global Memory Access Benchmark

Input data: number of elements N, step *stride*, offset *offset*, array A in global memory.

1: Calculate the number of elements in the thread, N_{thread};

2: Calculate the data range of this thread, using *stride* and *offset*;

3: while *index* is in the range do

4: Reading *A*[*index*] to variable *R*;

5: Increment *R* and save back to *A*[*index*];

6: *index* = *index* + *stride*;

7: end while

Using the above algorithm, the experiment was simulated, that shows how much gain from sharing depends on the number of threads in the warp. This is controlled by the *stride* variable. *Stride* denotes the gap between elements, that are accessed sequentially by a single thread. Consequently, threads in halfwarp can take advantage of coalesced access, if the *stride* value is large. For example, when *stride* = 32, each warp thread receives consecutive elements, which ensures complete union. When stride is 1, each thread reads one element that is offset by 32, so they are not completely merged and require 16 memory transactions that will be serviced for halfwarp. To ensure a fair comparison, in the above code the number of hits on the stream does not depend on *stride*.

In the code shown in algorithm 1, the number of calculations per iteration is very small compared to the memory access delay for stride = 1. However, as the stride value increases, memory access and calculations take approximately the same number of clock cycles. Using the MAX model, we can assume the runtime of this kernel and compare it with the actual one in Figure 1. The program execution graph is shown for various stride values. It should be noted that the basic access code only for memory, that is, with a small number of calculations, differs from the model, due to limited information about the hardware [9,10].

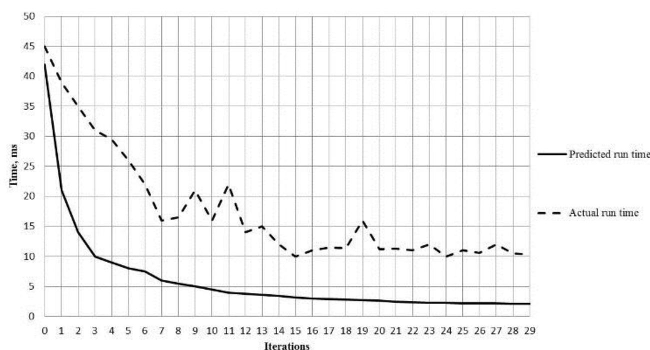

Fig. 1. The results of experimental studies of modeling access to global memory using the MAX model.

As can be seen from Figure 1, the value of stride significantly affects the execution time of the algorithm. Because the amount of operations of the actual calculations in the algorithm is very small, then from the graph you can see that to increase the performance of the parallel algorithm it is necessary to read the array elements located at a distance from each other by no less than the value of warp to take advantage of the combined access to memory. It can also be seen from Figure 1 that although the graph of the predicted lead time does not coincide with the graph of the actual time, it also allows the predicted lead time to obtain enough information about the decrease in performance while decreasing the distance between the read elements. This suggests that the model presented above adequately describes the performance of GPGPU.

V. Experimental Verification of the Impact of Access Conflicts When Using Shared Memory

In this experiment, while maintaining the general structure of global memory accesses, as in the previous experiment, each thread writes an element to the shared memory. The shared memory access pattern is controlled by the bank variable, which can be set to a value from 0 to 16. With a larger bank value, we can thus increase the number of access conflicts.

Algorithm 2 Shared Memory Access Benchmark

Input data: number of elements *N*, step *stride*, offset *offset*, control variable *bank*, array *A* in global memory, array *B* in shared memory.

1: Calculate the number of elements in the thread, N_{thread};

2: Calculate the data range of this thread, using *stride* and *offset*;

3: while *index* is in the range do

4: for *i* = 0 to 10000 do

5: Reading *A*[*index*] and save it back;

6: *B* [*IDthread* × *bank* (mod *sizeblock*)];

7: end for

8: end while

The kernel in this algorithm has about 16 cycles of calculation per iteration, and there are 64,000 iterations. The number of clock cycles required to access the memory is about bank × 4 per iteration [11,12,13]. Actual lead time and lead time predicted by the developed model are shown in Figure 2.

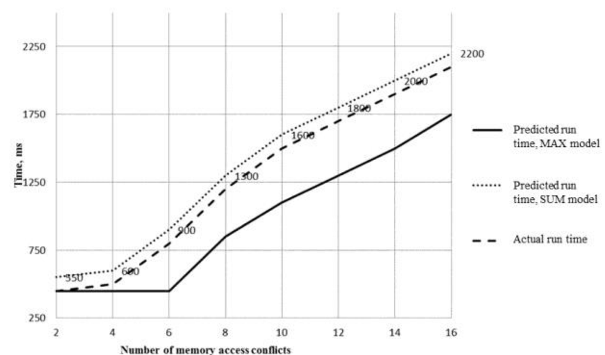

Fig. 2. The results of experimental studies of the access conflicts impact, when using shared memory.

As can be seen from Figure 2, there is a linear dependence of the number of conflicts on the execution time of the program. Thus, to increase the performance of parallel algorithms, it is necessary to exclude the intersection of the read data for different threads [14]. Also from Figure 2 we can conclude that the presented model allows us to adequately estimate the execution time of the algorithm in the presence of memory conflicts, although it does not provide accurate information due to the closed hardware platform.

VI. CONCLUSIONS

This paper presents an experimental study of previously developed models for predicting the performance of a heterogeneous computer system in telecommunications. The study showed that the developed models allow us to obtain an adequate estimate of the possible time of the algorithm for various parameters of the GPU. However, it is worth noting that the estimate obtained using the developed models is not accurate, because average access time is used for all levels of the memory hierarchy. In the future, it is planned to finalize the models taking into account the use of the amount of memory access time, obeying a rank distribution, for example, Pareto or Zipf.

ACKNOWLEDGMENT

The work was carried out under a grant from the president of the Russian Federation for state support of young Russian scientists № MK-2378.2020.9.

REFERENCES

[1] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," Computer Graphics Forum, vol. 26, no. 1, pp. 80-113, 2007.

[2] A.A. Kolpakov and Y.A. Kropotov, "Development of a model for predicting the performance of a heterogeneous computer system in telecommunications," Proceedings of ITNT, Samara National Research University, pp. 2265-2274, 2018.

[3] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," Proceedings of 10th Annual ACM Symposium on Theory of Computing (STOC), ACM New York, NY, USA, pp. 114-118, 1978.

[4] Y.A. Kropotov and A.A. Kolpakov, "Experimental study of the model for predicting the performance of a heterogeneous computer system in telecommunications," 12th International Scientific and Technical Conference Dynamics of Systems, Mechanisms and Machines, Dynamics, 2019. DOI: 10.1109/Dynamics.2018.8601478.

[5] L.G. Valiant, "A Bridging Model for Parallel Computation," Communications of the ACM, vol. 33, no. 8, pp. 103-111, 1990.

[6] P.B. Gibbons, Y. Matias and V. Ramachandran, "The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms," SIAM Journal of Computation, vol. 28, no. 2, pp. 733-769, 1999.

[7] CUDA C Programming Guide [Online]. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[8] D.R. Helman and J. JaJa, "Designing Practical Efficient Algorithms for Symmetric Multiprocessors," Lecture Notes in Computer Science, International Workshop, vol. 1619, pp. 37-56, 1999.

[9] A.A. Kolpakov and Y.A. Kropotov, "Advanced mixing audio streams for heterogeneous computer systems in telecommunications," CEUR Workshop Proceedings, vol. 1902, pp. 32-36, 2017.

[10] A.A. Kolpakov, "Theoretical estimate of the growth performance of a computer system using multiple computing devices," The world of scientific discoveries, no. 1, pp. 206-209, 2012.

[11] Y.A. Kropotov, A.A. Belov and A.Yu. Proscuryakov, "Issues of processing experimental time series in an electronic system of automated control," Electronics Issues, vol.1, no. 1, pp. 95-101, 2010.

[12] Y.A. Kropotov, "The algorithm for determining the parameters of the exponential approximation of the law of the probability distribution of the amplitudes of a speech signal," Radio engineering, no. 6, pp. 44-47, 2007.

[13] Y.A. Kropotov and A.A. Bykov, "The model of the law of the probability distribution of signal amplitudes in the basis of the exponential functions of the system," Design and technology of electronic tools, no. 2, pp. 30-34, 2007.

[14] Y.A. Kropotov, A.Yu. Proscuryakov and A.A. Belov, "A method for predicting changes in time series parameters in digital information management systems," Computer Optics, vol. 42, no. 6, pp. 1093-1100, 2018. DOI: 10.18287/2412-6179-2018-42-6-1093-1100.