Patryk ORZECHOWSKI[1], Krzysztof BORYCZKO[1]

AGH University of Science and Technology (1)

doi:10.15199/48.2015.08.31

Effective biclustering on GPU - capabilities and constraints

Abstract. *This article presents the benefits and limitations related to designing a parallel biclustering algorithm on a GPU. A definition of biclustering is provided together with a brief description of the GPU architecture. We then review algorithm strategy patterns, which are helpful in providing efficient implementations on GPU. Finally, we highlight programming aspects of implementing biclustering algorithms in CUDA/OpenCL programming language.*

Streszczenie. *W artykule przedstawiono korzyści i ograniczenia związane z projektowaniem równoległego algorytmu biklasteryzacji, przeznaczonego na GPU. Zaprezentowano definicję biklasteryzacji oraz skrótowo opisano architekturę GPU. Zestawiono popularne wzorce strategii implementacji algorytmów, przydatne w projektowaniu efektywnych rozwiązań na GPU. Publikacja zawiera także praktyczne wskazówki programistyczne, w kontekście implementacji algorytmów biklasteryzacji w języku CUDA/OpenCL.* ***(Efektywna biklasteryzacja z wykorzystaniem GPU - możliwości i ograniczenia)***

Keywords: biclustering, GPU, pattern recognition, data mining, parallel programming, CUDA, OpenCL
Słowa kluczowe: biklasteryzacja, GPU, rozpoznawanie wzorców, klasyfikacja, programowanie równoległe, CUDA, OpenCL

Introduction

Growing demands of the industry pushed forward the design of faster and more efficient multicore chips. The observation formulated as *Moore's Law* [1], which predicted that the number of transistors on chip would double every two years, has proved to be right for over a half of century. Similarly, the performance of computations per watt grew exponentially with the miniaturization of transistor sizes, observed as *Dennard scaling* [2]. The law was originally formulated for MOSFET transistors and stated that the smaller the transistors get, the faster they may switch at reduced power consumption. Released in 2008, the Fermi family of Graphics Processing Units (GPUs) offered over 3 millions of transistors of 40 nm size [3]. The Kepler architecture, released by NVIDIA in 2012, reduced the size of transistors even further down to 28 nm CMOS. It has recently been recognized [4] that GPU-based and CPU-based multi-core designs are reaching their technological limits, considering the expected performance speed-up levels, and only radical microarchitecture innovations may allow the trend to continue.

Currently, one of the major manufacturing technology constraints is power consumption. Improvements in production technology enabled designers to produce thinner transistors, which are more energy efficient. More such cores may be placed on a wafer and still the device will require the same amount of power (not taking into account necessary external cooling).

The second important challenge of parallel processing, apart from maximizing power efficiency, is maximizing the performance [5]. This may be accomplished in two ways: by minimizing computation latency or by maximizing computation throughput. The design of CPU serves to minimize latency, meaning minimizing the amount of time needed to transfer a block of data from memory to CPU or minimizing the time that CPU stays idle. The philosophy of GPU is quite different, as it aims at maximizing computation throughput - the number of tasks completed per unit of time. Massively parallel, multiple-core GPU architecture allows the achievement of as much as super-linear speedup. Significant performance gains of GPU processing are present in particular with large datasets or heavy computation of simple operation on the same data.

Another problematic issue is the development of libraries and tools that would simplify programming the devices using massively parallel processors. This requires providing tools for the developer that would support programming for massively parallel devices. The most popular standards used in programming heterogeneous environments are the Compute Unified Device Architecture (CUDA) platform created by NVIDIA, available only for NVIDIA GPUs, and the Open Computing Language (OpenCL) framework, which was accepted by leading companies on the market and may be used on various devices. The development and standardization of both allowed computing operations on GPU that were typically reserved for CPU. Hence, GPUs became used for General Purpose computing on Graphics Processing Units (GPUs). This burst development of many libraries and tools for various purposes, such as linear algebra or FFT computation, matrix manipulation, random number generation etc. They involve tools for CUDA such as cuBLAS, cuFFT, cuRAND, cuSparse, and tools for OpenCL, such as clMath and ViennaCL.

The programmer has a final say and is supposed to come up with an optimised implementation. Exploiting parallelism requires making different assumption in programming and sometimes a complete redefinition and reorganization of sequential code. Optimised sequential algorithms may not be easily ported into parallel version, especially on GPU, where memory constraints become an issue. This is possibly one of the reasons why there aren't many parallel implementations of biclustering algorithms for GPUs.

In this article we describe hardware limitations and programming aspects that need to be addressed while designing biclustering algorithms for GPU. We present commonly used parallel patterns, which may help to provide effective optimised biclustering solutions for GPUs. We also discuss the purpose of parallel biclustering on GPU and scaling capabilities for GPU.

Overview of GPU architecture

A GPU device contains multiple Streaming Multiprocessors (SMs), which run independently in parallel. Initialization of the device involves providing lists of operations to be executed on the device, which are called *kernels*, and for each kernel - the number and size of thread blocks (in CUDA, work-group in OpenCL), that contain threads (in CUDA, work-items in OpenCL) which are going to run in parallel. The comparison of naming terms used in CUDA and OpenCL, as well as architectural equivalent is presented in Table 1.

Table 1. Comparison of naming terms used in CUDA and Opencl, based on [6, 7]

CUDA	OpenCL	Architecture
Thread	Work item	Scalar processor
Thread block	Work group	Multiprocessor (SM)
Local memory	Private memory	memory per thread
Shared memory	Local memory	memory per block

Allocation of thread blocks to SMs is GPU's responsibility, therefore multiple blocks may be run by the same SM, but each block may run only on a single SM. The programmer may not assume any order of execution of blocks, nor is she able to allocate a block to a specific SM. The only assumptions that may be made are that threads of the same block are run simultaneously. Kernels by default are run sequentially by GPU, though the recent Kepler architecture with compute capability 2.x allowed concurrent execution of kernels.

Threads of the same block are partitioned into groups of 32-threads (16-threads on pre-Fermi devices) called warps [7]. Warps independently execute one instruction at a time in an architecture called Single Instruction Multiple Threads (SIMT). If the code diverges ("if-else" instruction), warp executes each path serially, switching off threads that are inactive in each of the branches.

GPU memory overview

There are four levels of memory in GPU: global, shared, local and constant memory. Access to global, private and constant memory is cached by L1 and L2 caches. The exchange of data between host and device is possible only through global memory.

Global memory, which has a low bandwidth and a high latency, is used for data exchange with the host. It is accessible by 32-, 64- or 128-bytes memory transactions, which are coalesced by warp into one or many transactions, depending on a type of the data object and the distribution of addresses enquired by threads. The optimal throughput may be achieved when addresses are not dissipated, but point to the consecutive location in memory.

The second level of memory, called *constant memory*, resides in the same place as global memory and is cached similarly.

The third level of memory is *shared memory*, which is on-chip and serves the communication within blocks of threads (work-groups). Shared memory has high bandwidth and low latency and is divided into modules of equal size, called banks, which may be accessed simultaneously. Successive 32-bit words are assigned to consecutive banks (as the result bank 0 has all words divisible by 32, bank 1 words give the rest of 1 from division by 32 etc.). The situation in which threads requested addresses of memory that are located within the same bank is called a *bank conflict* and is resolved by access serialization, slowing down throughput. The only exceptions to this are *broadcast* and *multicast* where all or several threads respectively address the same memory location. Providing a single request from each bank per warp yields the best throughput for shared memory.

Private memory is assigned to each thread separately, but is located at device memory, so it has latency and throughput similar to global memory. Private memory is organized in such a way that consecutive IDs of threads access consecutive 32-bit words. As long as all threads within the warp access the same relative address in memory, this access is coalesced.

A standard heterogeneous computation using CPU and GPU involves writing input data from host into device memory, reading data from a global memory, copying read values into local/shared memory, performing operations on local/shared memory in parallel and afterwards writing the result from local/shared memory into global memory. This takes advantage of the fact that the access to local/shared memory is many times faster than access to global memory.

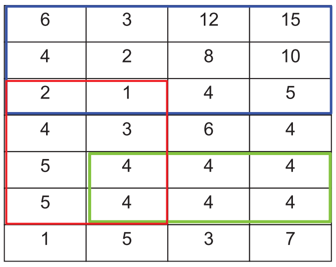
As threads on a device work together, they may overwrite each other's results. This is the reason why access to memory needs to be serialized. There are three levels of synchronization available on GPU: computation level (each thread executes instructions in a given order), barriers (points in a program where each thread waits until all threads within the block thread reach the barrier) and finally kernels (by default a kernel is executed after the previous kernel has ended, but this approach changes with Kepler GPUs).

Biclustering

Biclustering is a well-known data mining problem of finding one or many biclusters in a data matrix that meet a specific homogeneity criterion [8, 9, 10, 11]. Biclustering is considered to be very helpful method for detecting gene similarity in DNA microarrays and has been widely used in bioinformatics, genomics and medicine. It has gained attention in data mining as a local pattern recognition equivalent of clustering, which aims at detecting global similarities.

For a dataset $A = \{a_{ij}\}_{m \times n}$, which contains the following rows $X = \{x_{1,...,}x_n\}$ and columns $Y = \{y_{1,...,}y_m\}$, a bicluster is defined as a subset of rows I and subset of columns J. Formally, a bicluster is $B = (I, J) = \{a_{ij} \in A : i \in I, j \in J\}$, where $I \subseteq X$ and $J \subseteq Y$ and rows I exhibit the same (or similar) homogeneity criterion across the columns.

The purpose of biclustering is to obtain one or more biclusters at a time. Biclusters returned at each run of algorithm (or during a single run) may overlap with each other (on rows, columns or both) or be completely separated. The thorough analysis of biclustering issues is covered in [9] and [12].



6	3	12	15
4	2	8	10
2	1	4	5
4	3	6	4
5	4	4	4
5	4	4	4
1	5	3	7

Fig. 1. Examples of biclusters. (a) green - constant bicuster, (b) red - additive bicluster, (c) blue - multiplicative bicluster.

Biclustering algorithms identify one or more of the following major classes of biclusters [9]:

- constant biclusters - with exactly the same values within a bicluster - see Fig. 1 a.
- biclusters with constant rows (columns), where values in each row or column of a bicluster values are the same.
- coherent biclusters in an additive model, where each row or column may be calculated by adding a constant to values of other row or column - see Fig. 1 b.
- coherent biclusters in multiplicative model, where each row or column may be calculated by multiplying values of other row or column by a constant - see Fig. 1 c.
- biclusters with coherent evolutions, where the values in a matrix are symbolic. Coherent biclusters are identified regardless of the exact values in data matrix.

Initial results show that biclustering on GPU may provide significant speedup comparing to sequential versions of algorithms. Speedup of at least 9 up to 28 times is indicated by [13], depending on the dataset used. Therefore considerations of programming biclustering efficiently on GPU are fully justified.

Programming biclustering on GPU

Not many attempts have been taken to propose a biclustering program on GPU. One of the few approaches involve computing a task similar to biclustering, which is called betweenness centrality [14], or a recently taken approach proposed by [13]. Lack of articles in the field is not resulting from the spreading trend of parallel programming using GPU, but also major constraints connected with porting biclustering algorithms to heterogeneous environment.

The following aspects require programmer's attention when implementing parallel biclustering algorithms. The first issue derives straight from the definitions of biclustering and bicluster. The result of biclustering is usually some number of biclusters, unknown beforehand and determined during the process. The usage of GPU requires host to allocate a predefined amount of memory on GPU, what may happen to be insufficient or excessive, depending on the input dataset. A bicluster, which is a subset of rows and a subset of columns, requires rows to be compared with each other by values in corresponding columns. This requires repeatable reads of elements in different columns of each row, rendering memory coalescing very complex. Determining the data type to store biclusters needs consideration of whether a sparse matric, a static or dynamic object would be most suitable.

The second issue is type and size of the input data. Since many commonly used datasets in biclustering contain up to 50000 rows x 500 columns of floating-point numbers, there is an insufficient amount of private and shared memory to store all input data. Therefore, input data needs to be stored in global memory, limiting the throughput. The size of input memory is too large to be put in private/shared memory, on the other hand too small to hide global memory latency.

The third problem is determining the proper size of a grid. The maximum size of a workgroup is limited by hardware (usually 1024 threads), so it is impossible to cover all rows in input dataset with a single workgroup in such a way that each row would have its corresponding thread. Therefore division into multiple workgroups is required if the input matrix is to be processed by columns. Unfortunately, GPU does not specify the order in which various workgroups will be executed, bringing a risk of races between threads coming from different workgroups. There is hardly any way to ensure synchronization between threads of different workgroups, with the only three possibilities being the use atomic operations only (hardly acceptable), sticking to a single workgroup and placing a barrier (lowering the throughput of calculation) or providing synchronization at the kernels level (it is guaranteed that the next kernel in command queue will be run if and only if all the activities of the previous kernel have ended). Although the third option seems to be using the most GPU resources and seems to be the best choice, it has noticeable disadvantages. Data exchange between kernels may only be achieved by placing intermediate results in a global memory. As the access to the global memory is very time consuming, this memory overhead may undermine all the effect gained by using many workgroups.

Finally, the efficiency and performance may be improved by using best programming practices, such as patterns. The most common parallel patterns are summarized in the next chapter.

Parallel patterns for GPU

Different algorithm strategy patterns for parallel programming were described in details in [15]. In this article we present the most popular ones that are commonly used by GPU community [5].

The first pattern, called *map*, presents a one-to-one relationship between input and output data. More specifically, in *map* operation each task reads a single data element from an input array (for example: pixel from image), performs some operation or executes a function, and finally writes its result to a single element of an output array. Parallelizing *map* on the GPU may be achieved by assigning each task to a single thread and may be easily parallelized by avoiding bank conflicts or racing conditions between threads. Further optimisations may be achieved by memory coalescing (i.e. that threads read from the same chunk of memory). *Map* is commonly used for gamma correction and thresholding.

Transpose is another very popular one-to-one parallel pattern described in [5] that reorganizes the order of data elements in memory. A common usage involves reordering from array of structures (AOS) into structure of arrays (SOA) or vice-versa or transposing a matrix.

A simple generalization of *map* is called *stencil*. This operation uses several-to-one relation [5] For each task, data is read from a range of input elements (usually an element plus its neighbours, pointed by relative offsets using von Neumann, Moore or sparse neighbourhoods). The result of an operation is written into a single element of an output array. The larger the neighbourhood, the more data is reused, as multiple tasks read the same elements. Optimised parallel stencil for 1D arrays uses left- and right- shifting of read addresses. Performance gains may also be achieved by optimising *stencil* for cache by tiling (technique called strip-mining [15]) or by latency hiding (performing interior computations while waiting for calculation of cells). The common application of *stencil* pattern are computing a sum or average of elements or performing filtering, convolutions, differential equations etc.

Generalized *stencil* wherein many elements are used to calculate one element is called *gather* [5].

The *scatter* pattern is an inversion of *stencil*, wherein data is read by each thread from a single element and stored into multiple, overlapping locations. Therefore *stencil* reflects one-to-many relation. The problem of multiple writes into the same chunk of memory is a common situation where serialization of write access should appear, otherwise a racing condition between threads may occur. CUDA and OpenCL standard support atomic operations which serialize writes to memory, but the operation is not without cost.

Reduce pattern is an example of all-to-one case, where all processors cooperate with each other in order to obtain a single solution. Reduce shows noticeable speedup compared to the sequential version. An example of reduce is summing all input elements of an array, determining the minimum or maximum element,

Scan is one of the most primitive operations [5] which exhibits an all-to-all behaviour. The most popular parallel implementations of scan are described in [16] and [17].

Sort is a very complex problem for the GPU. There exist parallel implementations of mergesort or quicksort available for GPU. The most common parallel sorting algorithms are bitonic mergesort [18] and Radix sort [16].

Histogram pattern is another example of all-to-all situation, which shows a solution for a possible race condition between threads for updating the same values. One of the options is to use atomic operations available in CUDA and as extention of OpenCL, which serialize access to memory by default. The other method involves calculating histograms in private memory of each thread, which is followed by reduction in shared memory. The third method uses sorting and reducing by key strategy [5].

Summary of optimisation techniques

The programmer needs to remember the following aspects, in order to develop efficient programs on GPUs involving biclustering:

- Efficient access to global memory and exploiting data reuse. As access to global memory costs much, there is no need to repeatedly read elements from global memory, as long as it is possible to store elements in many times faster local/shared memory. Therefore redundant loads from global memory should be reduced to minimum.
- Exploiting coalescing and tiling. Avoiding non-unit stride global memory accesses should be avoided whenever possible [19]. Reads of successive elements from memory is executed simultaneously, whereas reads from different places in memory bring memory overhead.
- Reusing local/shared memory. Local memory may be also used to store partial results between threads;
- Load balancing between barriers. Each thread is supposed to be assigned similar amount of work, so that it finishes its task at similar time. Otherwise, threads that have already completed their tasks will have to wait at barrier point for others.
- Avoiding branches in code. The reason of this is not only idle threads waiting for other threads to complete operation. Threads in block are executed together in groups of 32 threads, called warp. If there are no branches in code, all threads are executed simultaneously, otherwise one branch is executed after another.

There are many other pitfalls that programmer should be aware of, for example the need to ensure safety when multiple threads access to memory. Bank conflicts (i.e. situation when multiple threads are accessing the same element in local memory) need to be avoided or serialized. Barriers may not appear within ”if-else” statements etc.

Conclusions

Parallelizing biclustering on GPU devices seems to be a challenging task. There are many constraints present in programming GPUs using CUDA or OpenCL that hindered the design of our prototypical parallel biclustering algorithm, which takes the advantage of some of the aforementioned optimization techniques. Basing on our experience, we believe that GPU architecture may successfully shorten the time of computation of biclustering methods. Nonetheless, designing fast and scalable biclustering algorithm on GPU requires thorough investigation. It is required though to consider different design schemes and perform multiple tests concerning the approach, grid sizes, communication between threads, blocks etc. in order to propose solution of acceptable efficiency.

A better insight of theoretical benefits of GPU high performance computing in heterogeneous environment may be obtained in comparison of GPU-based implementation of algorithm to other parallel implementations, such as using OpenMP programming standard, MIC architecture.

Acknowledgements

This research was funded by the Polish National Science Center (NCN), grant No. 2013/11/N/ST6/03204. This research was supported in part by PL-Grid Infrastructure.

REFERENCES

[1] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[3] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *IEEE Micro*, 31(2):50–59, 2011.

[4] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[5] D. Luebke, J. Owens, M. Roberts, and C.H. Lee. Intro to parallel programming. `http://www.udacity.com/course/cs344`. Online course in cooperation with NVIDIA.

[6] Opencl programming guide for cuda architecture. `http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf`, 2009. Version 2.3.

[7] Cuda c programming guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, 2014. PG-02829-001_v6.0.

[8] Y. Cheng and G.M. Church. Biclustering of expression data. In *Proceedings of the eighth international conference on intelligent systems for molecular biology*, volume 8, pages 93–103, 2000.

[9] S.C. Madeira and A.L. Oliveira. Biclustering algorithms for biological data analysis: a survey. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 1(1):24–45, 2004.

[10] A. Prelić, S. Bleuler, P. Zimmermann, A. Wille, P. Bühlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler. A systematic comparison and evaluation of biclustering methods for gene expression data. *Bioinformatics*, 22(9):1122–1129, 2006.

[11] K. Eren, M. Deveci, O. Küçüktunç, and Ü.V. Çatalyürek. A comparative analysis of biclustering algorithms for gene expression data. *Briefings in Bioinformatics*, 2012.

[12] P. Orzechowski. Proximity measures and results validation in biclustering – a survey. In L. Rutkowski et al., editor, *Artificial Intelligence and Soft Computing*, volume 7895 of *Lecture Notes in Computer Science*, pages 206–217. Springer Berlin Heidelberg, 2013.

[13] B. Liu, Yao Xin, Ray C.C. Cheung, and Hong Yan. Gpu-based biclustering for microarray data analysis in neurocomputing. *Neurocomputing*, 134(0):239–246, 2014. 19th International Conference on Neural Information Processing (ICONIP2012).

[14] A.E. Sariyüce, K. Kaya, E. Saule, and Ü.V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85. ACM, 2013.

[15] M.D. McCool, A. D. Robison, and J. Reinders. *Structured parallel programming patterns for efficient computation*. Elsevier/Morgan Kaufmann, Waltham, M.A., 2012.

[16] W.D. Hillis and G.L. Steele Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[17] G.E. Blelloch. Prefix sums and their applications. In *Sythesis of parallel algorithms*, pages 35—60. Morgan Kaufmann Publishers Inc., 1990.

[18] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[19] Nvidia's opencl best practices guide. `www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`, 2009.

***Authors**: Ph.D. Patryk Orzechowski, AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, Department of Automatics and Bioengineering, al. A. Mickiewicza 30, 30-059 Kraków, Poland email: patrick@agh.edu.pl; D.Sc. Krzysztof Boryczko, AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Kraków, Poland, email: boryczko@agh.edu.pl*