ACADEMIA
Accelerating the world's research.

Accelerating Clustering Coefficient Calculations on a GPU Using OPENCL

Igor Mishkovski, Dimitar Trajanov

Cite this paper

Downloaded from Academia.edu

Get the citation in MLA, APA, or Chicago styles

Related papers

Download a PDF Pack of the best related papers

Mixing multi-core CPUs and GPUs for scientific simulation software
Ken Hawick

Accelerating text mining workloads in a MapReduce-based distributed GPU environment
Sándor Darányi

Best Practice Guide - GPGPU
Volker Weinberg

Accelerating Clustering Coefficient Calculations on a GPU Using OPENCL

Leonid Djinevski[1], Igor Mishkovski[1], Dimitar Trajanov[1],

[1] Ss Cyril and Methodius University, Faculty of Electrical Engineering and Information Technologies, ul. Rugjer Boshkovikj bb,
PO Box 574, 1000 Skopje, Macedonia
{ leonid.dzinevski, igorm, mite}@feit.ukim.edu.mk

Abstract. The growth in multicore CPUs and the emergence of powerful manycore GPUs has led to proliferation of parallel applications. Many applications are not straight forward to be parallelized. This paper examines the performance of a parallelized implementation for calculating measurements of Complex Networks. We present an algorithm for calculating complex networks topological feature clustering coefficient, and conducted an execution of the serial, parallel and parallel GPU implementations. A hash-table based structure was used for encoding the complex network's data, which is different than the standard representation, and also speedups the parallel GPU implementations. Our results demonstrate that the parallelization of the sequential implementations on a multicore CPU, using OpenMP produces a significant speedup. Using OpenCL on a GPU produces even larger speedup depending of the volume of data being processed.

Keywords: Complex Networks, Parallel, CPU, GPU, speedup, OpenMP, OpenCL.

1 Introduction

Traditionally, the majority of the current software is written as sequential programs. As new generations of processors are coming, historically is expected the same sequential programs to run much faster. These expectations have slowed down since 2003 onwards, due to energy consumption, limited increase of clock frequencies and level of productive activities that can be performed in each clock period within a single CPU [1], therefore almost all microprocessor manufactures have switched to multicore processors. Today a sequential program will not run much faster on a new generation processor, because it will be using only one core from the multicore processor.

In order to keep the software expectations of performance improvements with each new generation of microprocessors, the software applications have to turn to parallel programming.

The GPUs since their emergence as peripheral units have become probably the most powerful computational processor for the cost at which are being sold. Their architecture is making them much more superior than the CPUs regarding the

execution throughput. Much of the CPU resources are dedicated for non-computational tasks like branch prediction and caching and their focus is more into maintaining the execution speed of sequential programs, while increasing the cores in the meanwhile. This architectural difference allows the GPUs to have much bigger growth than the CPUs.

A recent survey on graphic hardware computation performance [2] gives an overview of the execution throughput of GPUs in comparison to CPUs. The overview states that the GPUs have a huge advantage over the CPUs, which is a very justifiable reason for the software developers to move their applications on GPUs.

A science discipline of our interest is the area of Complex Networks. Complex Networks by their nature are an interdisciplinary area of graph theory and natural and social sciences. One of the main features regarding complex networks is the computationally intense calculations of their measurements, which require lots of resources. This is why we decided to harvest the power of the GPUs in this science discipline.

The model of complex networks permeates our everyday life, due to its simplicity (a certain number of nodes representing individual sites and edges representing connections) and its ability to grasp the essence of many different systems. Commonly cited examples include social networks, technological networks, information networks, biological networks, communication networks, neural networks, ecological networks and other natural and man-made networks. Abundant study of their topology and models is presented in [3] [4] [5].

Each complex network has specific topological features, which characterize its connectivity and highly influence the dynamics of processes executed on the network. The analysis of complex networks, therefore, relies on the use of measurements capable of expressing the most relevant topological features. However, in order to find the topological features of a given real complex networks the researchers are using programs and packages, such as Pajek [6], Ucinet [7], matlab_bgl [8] etc. From our own experience the code for these measurements it is not so complex, but as the real networks become more and more complicated (enormous number of nodes and edges) we often encountered memory or computation related problems. More specifically, the simulations could not be done (or some tricks had to be used) because of memory constraints and sometimes the simulations took too much time. Thus, in this work we address and overcome these two problems, by representing the network via hash-table based structure instead of traditional adjacency matrices (see Section 3), and we speed up the simulations via parallel programming algorithms for CPU and GPU execution in OpenCL (see Section 4). The main contribution of this work is to make possible or easier to analyze large-scale networks mapped over real world examples, by harvesting the tremendous power of the GPU performance.

At the end of this paper, a comparison of the results is presented. The results are obtained by running the sequential, OpenMP and OpenCL implementations of the calculation of the clustering coefficient. Conclusions about the maximum accelerations are specified, according to the parallelization of portions of the sequential code.

2 Related work

Pawan Harish and P. J. Narayanan presented fast implementations of a few fundamental graph algorithms for large graphs on the GPU hardware [9]. Their implementations present fast solutions of BFS (breadth-first search), SSSP (single-source shortest path), and APSP (all-pairs shortest path) on large graphs at high speeds using a GPU instead of expensive supercomputers.

Joerkki Hyvoenen, Jari Saramaeki and Kimmo Kaski presented an article about a cache efficient data structure, a variant of a linear probing hash table, for representing edge sets of large sparse complex networks [10]. Their performance benchmarks show that the data structure is superior to its commonly used counterparts in programming applications.

Eigenvalues are a very important feature of Complex Networks. NVIDIA CUDA 1.1 SDK contains a parallel implementation of a bisection algorithm for the computation of all eigenvalues of a traditional symmetric matrix of arbitrary size with CUDA that is optimized for NVIDIA's GPUs [11]. V. Volkov and J. W. Demmel in [12] have improved the algorithm from the CUDA SDK.

Another useful feature is finding the shortest path between nodes. Gary J. Katz and Joseph T. Kider Jr in [13] describe a GPU implementation that solves shortest-path problems on directed graphs for large data sets.

3 Complex Networks

There are many relevant measurements that describe the complex networks like: measurements related with distance; clustering; degree distribution and correlation; entropy; centrality measurements; spectral measurements; and many others. These measures are important to capture the topological properties of real complex networks, which will further give insights of the robustness, efficiency, vulnerability, synchronization, virus propagation, etc. For this paper the clustering coefficient measure is chosen, because it is a property that is quite computationally intense and is also similar to other measures.

Clustering represents a local feature for a given node which is a measurement of how much its neighbours are grouped. The effect of clustering is measured by the clustering coefficient C which represents the mean value of the probability that two neighbouring nodes of a given node are also neighbours between each other. The equations for obtaining the clustering coefficient C_i for a given node, i and the clustering of the network $<C>$ are the following:

$$C_i = \frac{2E_i}{k_i(k_i-1)} . \quad (1)$$

$$\langle C \rangle = \frac{1}{N}\sum_{i=1}^{N} C_i . \quad (2)$$

In the equation (1), E_i stands for the number of links between the neighbours of the node i, k_i is number of neighbor nodes of the node i, and N is the total number of nodes in the network.

4 Implementation

Real complex networks are typically very sparse and large structures. Mainly, in the literature the complex networks are represented mathematically with an adjacency matrix A. The elements a_{ij} are equal to 1 if the nodes i and j are neighbours, or 0 if they are not, involves a lot of redundancy in the adjacency matrix. Having in mind the memory capacity that is needed for encoding the complex networks, the adjacency matrix is difficult or cannot be directly used as data structure.

From a hardware perspective, the problem lies in the latency of the memory chip. By the time the data from the main memory arrives to the processor registers, several hundreds of clock cycles would have been finished. The modern processors solve this problem in 2 ways. As mentioned in the introduction, the CPUs introduce cache memory between them and the main memory. The cache memory, which is quite faster [14] and more expensive then the main memory, keeps the recently accessed data. Another way to reduce the main memory latency is to increase the bandwidth by using per-fetching where adjacent locations are loaded simultaneously, which increases the cache hits.

As mentioned in the introduction, the GPUs advantage over the CPUs is the execution throughput, which hides the main memory latency, but only by ensuring that the processor is always busy with computations, while other computations are waiting on memory access. This latency hiding is useful when encountered with larger complex network load data, which results with higher number of calculation. For smaller complex network load data, the calculations number is lower, which makes the GPU's execution throughput not useful because just the time spent for transferring the data from host memory to the processor registers of the GPU is quite big compared to the time for execution on the CPU.

Nevertheless, for large complex networks, the cache does not improve the latency much. The data cannot be stored into memory such that the adjacent memory elements are neighbouring nodes in the networks and the cache misses being bound to happen. Introducing an efficient data structure based on hash-tables [10] for encoding the complex network data is a way to solve the problem in larger networks.

0	-1	-1	-1	-1	...	-1
1	2	-1	-1	-1	...	-1
2	3	10	12	-1	...	-1
3	7	15	-1	-1	...	-1
4	-1	-1	-1	-1	...	-1
5	6	6	17	-1	...	-1
6	-1	-1	-1	-1	...	-1
7	11	-1	-1	-1	...	-1
			...			
N	124	-1	-1	-1	...	-1

N x max_links

Fig. 1. Representation of a complex network using a hash-table based data structure.

As it can be seen from Figure 1, the data is represented by a hash-table based structure, where the index of each row represents the index of the appropriate node *i*, and *N* stands for the number of nodes and *max_links* for the maximum number of neighbours. Each row contains the neighbouring nodes of the node *i*, which are marked with a grey background. The width of the rows is fixed and the value of the width is determined by the maximum number of neighbours (*max_links*) that a node from the complex network can have. For the many cases where the number of neighbours is less than the maximum, the rest of the data elements are filled with a negative number, so there is a difference between nodes and empty data. The reduction of the redundancy of the adjacency matrix has a big contribution towards optimizing the parallel implementations in regards to memory bandwidth, resulting in less data being copied from host memory to the processor register. Also, with this structure the GPU eliminates few memory accesses to the main memory. The penalty for the exclusion is paid by transferring the extra padding of empty data to the shared memory, which in the end proves to be more efficient (See 4.1).

Currently there are 3 major GPU parallel programming languages DirectCompute [15][16], CUDA [17] and OpenCL [18]. For the GPU parallel programming OpenCL is used. OpenCL is a standardized programming language, formed by the major industry leaders Apple, Intel, AMD/ATI, NVIDIA, and others [19]. OpenCL is an open standard, parallel programming language of modern processors found in PCs, servers and embedded devices. It is very much similar to CUDA, but unlike CUDA it is agnostic and manufactures independent. Also an open standard, the source code is portable across implementations.

The OpenCL standard is designed to take advantage of all of the system resources available. Unlike GPU programming languages in the pass which were just specifically multimedia, the standard supports general purpose parallel computing. It is based on ANSI-C99 with additional qualifier, data types and build-in functions. When working with GPUs, the focus in OpenCL goes into data parallelisation.

4.1 Clustering Coefficient

The sequential implementation of the clustering coefficient is quite straight forward. From equation (1), the implementation needs to find the number of links E_i

between the neighbours, and the number of all neighbours k_i for each node i form the hash table structure. In order to calculate the clustering coefficient C_i for every node in the complex network, few nested loops are needed. The first level loop is iterated N times, where N is the number of nodes that the network contains. The second level nested loop iterates the vector for each node i, in order to obtain the values for k_i, which is the degree of each node i. Another second level nested loop, and a third level nested loop inside it, are iterated in order to find the value E_i. This is done by comparing if every pair of the neighboring nodes is connected. So because of the hash table structure, the third level nested loop iterates through other vectors according to the values in the data elements for the neighbors of the node i. Having the values for E_i and k_i, the final calculation describe in equation (1) is performed.

Developing the parallel OpenMP implementation is based on the sequential implementation and is performed quite easy because there are no data dependencies. All nested loops are under one OpenMP pragma *parallel for* directive [20], which defines the private, shared, and reduction properties, which can be seen in the Listing 1.

Listing 1. OpenMP implementation of Clustering Coefficient

```
#pragma omp parallel for default(none) \
shared(max_links, node_size, net_data, h_C, Ei, ki, \
search_index, search_row_size) \
private(i,j,k,z)
for(i = 0; i < node_size; i++){
  Ei = 0;
  ki = net_data[i*max_links + 0];
  for(j = 1; j <= ki; j++){
    for(k = j + 1; k < ki; k++){
      search_index = net_data[i*max_links + k] *
                                          max_links;
      search_row_size = net_data[search_index + 0];
      for (z = 1; z < search_row_size; z++)
        if(net_data[i*max_links + j] ==
                     net_data[search_index + z]) Ei++;
    }
  }
  h_C[i] = (float)(2*Ei)/(ki*(ki-1));
}
```

For each nested loop the compiler creates a separate team of threads only if the nested parallelism is enables, otherwise only the outer loop is parallelized, and the other loop are serialized. The nested parallelism is supported in OpenMP 3.0 [21] by adding the collapse clause in the pragma *parallel for* directive. For earlier versions [22], only the outer loop is parallelized, while the other nested loops are performed by each of the 1/p threads, where p is number of core that the CPU has. We recommend using the early version for nested parallelism because the version 3.0 may introduce some overhead.

The parallel OpenCL implementation harvests the power of the GPU by using the shared memory. Before executing on the GPU, the host CPU takes care of the

compiling and building of the OpenCL kernels (Just In Time – compiler), and initializes other OpenCL necessary objects. The complex network data, before is loaded from the host to global memory using OpenCL input buffer, an additional padding is introduced while allocation, so the number of nodes are a multiple of the number of assigned workgroup size [23]. A unit of work in OpenCL is called a work-item, which can be conceptualized as a thread, because each instance of a kernel execution is done by a single thread. A group of work-items form a work-group, which is the local size, and the minimal value is limited to 32 because of the hardware limitation of a warp. In our case the maximum number of local workgroup size 512 proved most efficient. The number of workitems is initialized with the number of nodes N in the complex network, which is our global size, thus N threads are executed. A fragment of the kernel is presented in the Listing 2.

Listing 2. OpenCL implementation of Clustering Coefficient

```
for(j = 1; j <= ki; j++){
  first = input[gid*max_links + j];
  for(k = j + 1; k <= ki; k++){
    search_index = input[gid*max_links + k]*max_links;
    for (z = 1; z < max_links; z += lsize){
      if((z + lsize)>max_links) lsize = max_links - z;
      if((z + lid) < max_links)
        sh_tmp[lid] = input[(search_index + z + lid)];
      barrier(CLK_LOCAL_MEM_FENCE);


      for(i = 0; i < lsize; i++)
        Ei += (sh_tmp[i] == first);
        barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
}
if (gid < node_size) output[gid] = (float)(2*Ei)/
                                        (ki*(ki-1));.
```

For obtaining the degree k_i of each node, in the kernel code, a simple parallel reduction is performed. In order to calculate the value for E_i, a first level nested loop iterates through the neighbors for each node i. A second level nested loop iterates through the rows, which are the values neighbors of the node i, with a step iteration *lsize*, where *lsize* is the size of the local memory. This is done in order to take advantage of the local memory, so when the synchronization barrier is passed, the local memory is loaded by all the free threads with the *lsize* elements. This allows for the third level nested loop to efficiently access the local memory and compare if the neighbors are connected, which is much faster than accessing the global memory. The number of connected neighbors is contained in E_i, which is a local integer register. The output, which is the calculation described in equation (1) is read into main memory using OpenCL output buffer.

5 Results

The results from the sequential and parallel implementation are system dependent. Therefore, this is a case-study to determine how much the performance of parallel implementations using both OpenMP and OpenCL are improving the sequential implementation. The implementations were executed on the computer which specification is presented on table 1:

Table 1. System specifications.

Devices	Specs
CPU	Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz 4 cores 8 thread Hyper-Threading Technology
RAM	12GB DIMM 1333 MHz (0.8ns) 64bit
GPU	NVIDIA GeForce GTX 285 1GB 240 CUDA cores @ 1476MHz

All executions were run on Ubuntu 10.04. The CPU implementations were written in C using standard and OpenMP libraries. Nvidia Graphics driver version 3.1 was used for OpenCL compatibility.

There are seven datasets that are generated for each of the three main network models. Each data set is generated with different number of nodes and links, thus obtaining different sizes of the networks. The sizes are noted as scaling factors 1, 2, 5, 10, 20, 50 and 100. For the scaling factor 1, 500 nodes are generated, which results in an adjacency matrix of 250000 elements. The other nodes are chosen by doubling the elements of the adjacency matrix, so for scaling factor 2 the adjacency matrix has 499749 elements, for scaling factor 5 it has 1249924 elements, and so forth until scaling factor 100 when the matrix has 25000000 elements.

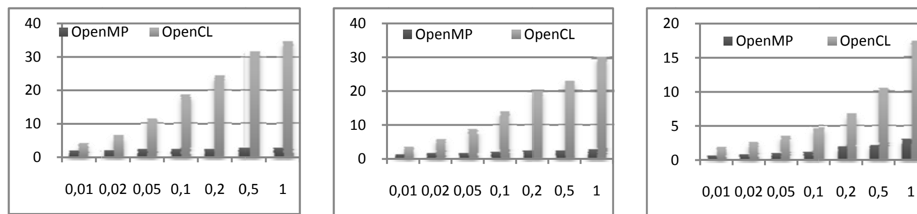

Fig. 2. The speedup of executions for the clustering coefficient calculations on CPU and GPU including the transfer time of the load data for random, small-world, and scale-free complex networks.

The obtained results for each of the implementations are summarized on figure 2 for each of the main complex network models: random, small-world and scale free model respectively. Each of the implementations is executed 10 times, and average execution times are obtained. Looking at the results for the times of the OpenCL

implementation, for all scaling factors, a conclusion can be made that for small volumes of data, executing the implementation on a GPU is not a smart way to go. For example in the executions for a scaling factor 1, the time spent for initializing OpenCL objects, building of kernels and programs, and allocating memory, is close or sometimes even bigger to the time spent for the CPU implementations. This proves that only by working with larger volumes of data, using the GPU is justifiable.

From figure 2, it can be seen that for different complex network models with the same number of nodes and appropriate number of links, the speedups are different. In average we obtained speedups of x3 for OpenMP, while for OpenCL we achieved x20. A conclusion can be made that for the random model of complex networks, the acceleration of calculations using the GPU is the greatest, while for the scale-free model the accelerations are smallest. The problem why we get worse speedups for the scale-free complex networks that the neighbours are concentrated around few nodes, because they have power law degree distribution, while in the other complex networks the distribution of neighbours is spread more equally. This makes us believe that another algorithm should be chosen for more efficient calculating of clustering coefficient for scale-free complex networks.

6 Conclusion

In this paper, we presented an implementation of an algorithm for calculation of the clustering coefficient measure of complex networks, for the three main complex network models. By utilizing hash-table base structure, we organized the complex network graph, which resulted of less data copied from host memory to the GPU processor registers. Also, because of the padding introduced of the hash-table base structure, more efficient use of the GPU's shared memory was achieved. We demonstrated the power of using the GPUs, providing a further evidence of effectiveness for accelerating complex networks calculations. The size of the GPU memory limits the size of the graphs handled on a single GPU. However, OpenCL provides for multiple devices to be interfaced, such that the work is distributed to each of them, thus expanding the capacity for calculating measurement for larger complex networks.

We have also performed comparisons with optimized implementations of CPU-based sequential and OpenMP parallel algorithms. The obtained acceleration of the measure calculation of complex networks is another example of the tremendous parallel power of the modern programmable GPU devices. These results of acceleration on the GPU provide a big interest. Seeing the GPU as high performance co-processing unit for any application eligible for data parallelization and having in mind the low cost of the GPU hardware, compared to the expensive CPUs of similar calculation power, points to GPUs as interesting area for future research and commercial development.

References

1. David B. Kirk , Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Published February 5, 2010
2. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, In Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.
3. Steven H. Strogatz. Exploring complex networks. Nature, 410(6825):268-276, March 2001.
4. R. Albert and A.L. Barabasi, Statistical mechanics of complex networks, Reviews of Modern Physics, vol. 74, no. 1, pp. 47-97, Jan 2002.
5. M. E. J. Newman, The structure and function of complex networks, SIAM Review, vol. 45, no. 2, pp. 167-256, 2003.
6. V. Batagelj, A. Mrvar: Pajek – Analysis and Visualization of Large Networks. In Junger, M., Mutzel, P. (Eds.): Graph Drawing Software. Springer (series Mathematics and Visualization), Berlin 2003. 77-103. ISBN 3-540-00881-0.
7. Borgatti, S.P., M.G. Everett, and L.C. Freeman, Ucinet 6 for Windows: Software for Social Network Analysis, H.A. Technologies, Editor. 2002
8. D. Gleich, Matlab BGL v1.0, April 27, 2006, retrieved April 2010, http://www.stanford.edu/dgleich/programs/matlab_bgl/.
9. Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In High Performance Computing (2007), vol. 4873 of Lecture Notes in Computer Science, Springer, pp. 197—208.
10. Joerkki Hyvoenen, Jari Saramaeki, Kimmo Kaski, Efficient data structures for sparse network representation, International Journal of Computer Mathematics, 85 (8) : 1219-1233, 2008.
11. Lessig, C. 2007. Eigenvalue Computation with CUDA, NVIDIA CUDA SDK 1.1.
12. V. Volkov and J. W. Demmel. LAPACK working note 197: Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. Technical Report UCB/EECS-2007-179, EECS Department, University of California, Berkeley, 2007.
13. Gary J. Katz and Joseph T. Kider Jr. All-Pairs Shortest-Paths for Large Graphs on the GPU. Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008.
14. Cantin, J. and Hill, M., Cache performance for selected SPEC CPU2000 benchmarks, ACM SIGARCH Computer Architecture News, 29, 13–18, 2001
15. DirectCompute Support on NVIDIA's CUDA Architecture GPUs, http://developer.nvidia.com/object/directcompute_home.html/.
16. Nigel, Dessau, senior VP and CMO at AMD about DirectCompute, http://developer.nvidia.com/object/directcompute_home.html/.
17. OpenCL Programming for the CUDA Architecture, Version 2.3, 8/31/2009.
18. The OpenCL Specification, Version 1.0, document Revision 43, 2009, retrieved February 2010 from http://www.khronos.org/opencl/.
19. The Khronos Group, Open Standard for Media Authoring and Acceleration, http://www.khronos.org/.
20. Barbara Chapman, Gabriele Jost, Ruud van der Pas, Using OpenMP, Portable Shared Memory Parallel Programming, The MIT Press Cambridge, Massachusetts London, England.
21. OpenMP Application Program Interface, OpenMP Architecture Review Board, Version 3.0 May 2008.
22. OpenMP Application Program Interface, OpenMP Architecture Review Board, Version 2.5 May 2005.
23. NVIDIA OpenCL, Best Practices Guide, Version 1.0, August 10, 2009.