ELSEVIER

Procedia Computer Science 3 (2011) 396–400

Procedia Computer Science

www.elsevier.com/locate/procedia

WCIT-2010

Parallel wavelet-based clustering algorithm on GPUs using CUDA

Ahmet Artu Yıldırım [a] * , Cem Özdoğan [a]

[a] *Department of Computer Engineering, Çankaya University, Balgat, 06530 Ankara, Turkey*

Abstract

There has been a substantial interest in scientific and engineering computing community to speed up the CPU-intensive tasks on graphical processing units (GPUs) with the development of many-core GPUs as having very large memory bandwidth and computational power. Cluster analysis is a widely used technique for grouping a set of objects into classes of "similar" objects and commonly used in many fields such as data mining, bioinformatics and pattern recognition. WaveCluster defines the notion of cluster as a dense region consisting of connected components in the transformed feature space. In this study, we present the implementation of WaveCluster algorithm as a novel clustering approach based on wavelet transform to GPU level parallelization and investigate the parallel performance for very large spatial datasets. The CUDA implementations of two main sub-algorithms of WaveCluster approach; namely extraction of low-frequency component from the signal using wavelet transform and connected component labeling are presented. Then, the corresponding performance evaluations are reported for each sub-algorithm. Divide and conquer approach is followed on the implementation of wavelet transform and multi-pass sliding window approach on the implementation of connected component labeling. The maximum achieved speedup is found in kernel as 107x in the computation of extraction of the low-frequency component and 6x in the computation of connected component labeling with respect to the sequential algorithms running on the CPU.

© 2010 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.

Selection and/or peer-review under responsibility of the Guest Editor.

Keywords: GPU computing; CUDA; cluster analysis; WaveCluster algorithm

1. Introduction

Cluster analysis is one of the most widely used common technique for grouping a set of objects into classes of "similar" objects or clusters. As stated formally, let dataset $X \in R^{m \times n}$ be the set of objects x_i, $1 \leq i \leq m$, for any dimension of n. The goal of clustering is to map group of more "similar" objects x_i into K nonempty clusters $\{C_1, C_2, C_3, \ldots, C_K\}$. Cluster analysis is highly used in many fields such as data mining, bioinformatics and pattern recognition.

WaveCluster is an unsupervised clustering approach with multi-resolution feature based on wavelet transform for very large spatial datasets which has the ability to discover of clusters with arbitrary shapes and can deal with outliers (data points that don't belong to any cluster) effectively [1]. WaveCluster defines the notion of cluster as a dense region consisting of connected components in the transformed feature space. To transform feature space, wavelet transform is applied which is a contemporary signal processing tool for decomposing signal into high-frequency and low-frequency components. The low-frequency component represents a lower resolution approximation of the original feature space on which connected component labeling algorithm is carried out to detect clusters at different scales from fine to coarse. Hence, WaveCluster gains multi-resolution property by means of wavelet transform. In the last step, a lookup table is constructed to map the units in the transformed feature space to original feature space and cluster numbers are assigned to each object.

With the advent of many-core graphical processing units (GPUs), there has been a substantial interest to speed up

1877-0509 © 2010 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2010.12.066

the CPU-intensive tasks on GPUs to utilize its enormous computational performance in scientific and engineering computing community. One remedy is to adapt the computational task to the graphics APIs such as OpenGL or DirectX, but they are not convenient for non-graphics applications and impose many hurdles to the general purpose application programmer [2]. NVIDIA introduced CUDA (Compute Unified Device Architecture) in November 2006 [3] to enable data-parallel general purpose computations on NVIDIA GPUs in an efficient way. In the CUDA programming model [3], GPU runs the computationally intensive data-parallel parts of the application as a co-processor in a SPMD (Single-Program Multiple-Data) manner while allowing the CPU to conduct concurrent tasks and sequential parts of the application. In recent years, several but increasing CUDA studies have been conducted on the field of data mining to take advantage of the high performance of GPUs [4,5].

Despite effectiveness of WaveCluster algorithm, execution time of the algorithm has become a serious concern when dataset size is large. In the past, we already introduced our parallel WaveCluster algorithm based on the message passing model for distributed memory multiprocessors [6]. In this study, we present CUDA implementations of extraction of low-frequency component by means of wavelet transform and connected component labeling for shared memory system and present performance analysis respectively. The evaluations are performed on a synthetic dataset for varying dataset sizes.

2. CUDA Programming Model

From the point of programmer's view, GPU is regarded as a device that runs hundreds of concurrent lightweight threads with zero scheduling overhead. In this model, all threads execute the same instruction but perform on different data concurrently. The common function, called *kernel*, is executed by each thread that can be programmed by ANSI C language extended with several keywords and constructs. Besides, other languages such as CUDA Fortran, OpenCL and DirectCompute, are supported by CUDA software environment [3]. When the kernel is invoked, CUDA runtime creates a *grid* which is composed of blocks of threads. Each thread and block is distinguished by the built-in index variables which are automatically assigned by CUDA runtime and each thread accesses its memory region using these index variables. The runtime values of grid and block sizes are specified at the kernel invocation time via language extensions.

The CUDA memory model employs three levels of memory sharing to take advantage of high memory bandwidth of the CUDA device which are local, shared and global memories [7]. Local memory provides very fast access for threads with very limited size used to store non-array local variables that are private to each thread. Shared memory is allocated for all threads within the same block which has also fast access time and is used to store frequently accessed data to save global memory bandwidth. The synchronization mechanism of shared memory is succeeded by calling *__syncthreads()* barrier function which blocks until all threads within the same block have reached this routine. There is no intrinsic synchronization mechanism among the threads in different blocks to allow thread blocks to be scheduled in any order across any number of cores automatically (automatic scalability) and to avoid the possibility of deadlock [7]. Finally, global memory is the slowest but has highly large memory size when compared to local and shared memories and is the only accessible memory from all threads and the host application. The contents of global memory are retained during the lifetime of the application if not being freed intentionally.

3. Implementations of the CUDA Algorithms

3.1. Extraction of Low-Frequency Component of the Signal

Discrete wavelet transform is the core part of the WaveCluster algorithm which takes advantage of its multi-resolution feature. As mentioned previously, since WaveCluster tries to find dense regions over low-frequency component of the signal, we implemented only the extraction phase of the low-frequency component. The algorithm is designed for 2-dimensional feature space to ease algorithm demonstration, but it can be expanded to many dimen-

Algorithm 1 Decompose2DLFComponentKernel

Require: V^{j}, lastlevel, threshold
Ensure: V^{j-1}

Algorithm 2 ConnectedComponentLabelingKernel

```
 1: declare I [dim.y * 2][dim.x * 2] in shared memory
 2: declare H [dim.y * 2][dim.x] in shared memory
 3: load V^j into buffer I
 4: apply one-dimensional wavelet transform to each
    column of points of 2x2 field and store into buffer H
 5: apply one-dimensional wavelet transform to each
    row of points over H and store approximation value
    in local memory m
 6: if lastlevel = true then
 7:    if val > threshold then
 8:       m ← MAXFLOAT
 9:    else
10:       m ← threadindex
11:    end if
12: end if
13: V^(j−1)[threadindex] ← m
```

```
 1: if direction = RIGHT AND last thread in row then
 2:    return
 3: end if
 4: if direction = DOWN AND last thread in column
    then
 5:    return
 6: end if
 7: declare I [dim.y * 2][dim.x * 2] in shared memory
 8: calculate startPositionIndex of the local with respect
    to direction
 9: load disjoint 2x2 points (window) into buffer I using
    startPositionIndex
10: find minimum value among points in the window
    stored on buffer I
11: assign minimum value to foreground points
12: if anyvalueofpointsischanged then
13:    ischanged ← true
14: end if
```

Fig. 1. CUDA Algorithms (a) Extraction of Low Frequency Component; (b) Connected Component Labeling

sional wavelet transform unless the limit of shared memory allocation is exceeded. The algorithm can be regarded as simple convolution operation as well.

There are many wavelet types exist such as Haar, Daubechies, Morlet and Mexican hat. We used Haar wavelet, because its initial window width is two which leads us to less waste of shared memory usage. The host function calls kernel function as much as the value of *scale* and kernel function returns the lower frequency representation (V^{j-1}) of the feature space (V^{j}) at each iteration where the approximation spaces are nested [8].

We followed divide and conquer approach in the CUDA implementation of this process. Each thread is responsible to calculate one approximation value extracted from local disjoint 2x2 square-shaped points of feature space. Before kernel invocation, input feature space is transferred from host memory to global memory of the device and output buffer is allocated in device memory to store transformed feature space. In the kernel, input feature space is transferred from global memory into shared memory of buffer *I* to make data access efficient. Each thread firstly applies one-dimensional wavelet transform to each column of local feature space and stores "intermediate" values in the shared memory buffer of *H*. The final approximation value is eventually calculated by applying second one-dimensional wavelet transform to each row of points on *H*. So, buffer *H* is used to store temporal results. If the aim is only to calculate approximation representation of the signal, the operation can be halted here. However, our implementation also makes the data suitable for usage in the connected component labeling algorithm. For that purpose, it performs thresholding operation at the last iteration (last level of wavelet transformation) to assign maximum float value to background points and unique thread index value to foreground points with respect to threshold value. This operation also removes outliers on the transformed feature space.

3.2. Connected Component Labeling (CCL)

Several algorithms have been studied about the implementation of CCL on CUDA machine recently [9]. In this paper, we present multi-pass CCL algorithm based on sliding window approach which groups the points with respect to pixel connectivity by sliding many windows over feature space concurrently. Each thread is responsible of 2x2 square-shaped field forming a window where calculates the minimum value of these points and assigns the minimum value to its foreground points. As a prerequisite, each foreground point is expected to be assigned unique increasing value in a left-to-right and top-to-bottom manners. Maximum float number are assigned to background points to ensure consistency in the algorithm. Our algorithm consists of three sub-operations. In the first sub-operation, the windows stay steady and in the second and third sub-operation, windows move to the right and down respectively. Thus, the minimum value of connected points can be propagated in all directions. The algorithm continues to iterate until no point value changes in all three sub-operations. For that reason, additional 3 iterations are executed to detect this stopping criterion. If any thread changes any value of the points, the variable of *ischanged*

is assigned *true* value that kept in the global memory. The host function of the kernel keeps track of the *ischanged* variable for each sub-operation and then resets its value to false before kernel invocation. The execution time of our algorithm is highly dependent to the maximum distance between two points within cluster. Since GPU runs each operation very fast in a parallel manner, our algorithm finishes the execution in a reasonable amount of computation time. The algorithm demonstration is depicted in Figure 2 where each window is surrounded with bold line.

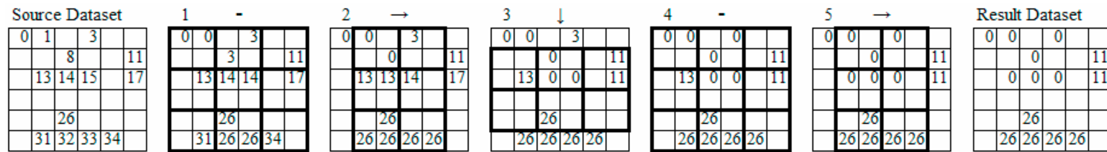

Fig. 2. Iteration Demonstration of CCL CUDA Algorithm

4. Performance Experiments

We have investigated the speedups of our CUDA algorithms compared to the sequential ones. The experiments were conducted on a Linux workstation with 2 GB RAM, Intel Core2Duo (2 Cores, 2.4 GHz, 4MB L2 Cache) processor and NVIDIA GTX 465 (1 GB memory, 352 computing cores, each core runs at 1.215 GHz) with compute capability 2.0 and runtime version 3.10. The sequential code of CCL is implemented using union-find data structure [10] which is highly fast and efficient with respect to linked-list implementation of CCL algorithm. We used two-dimensional synthetic dataset in the experiments and higher resolutions of this dataset are obtained by copying all points of the source dataset onto destination surface multiple times. Execution time (in microseconds) and kernel speedup results are presented in Table 1.

Table 1. Performance results of CUDA and CPU versions of algorithms for varying number of points and *scale* = 1 (times in microseconds)

			Extraction of Low-Frequency Component			Connected Component Labeling (CCL)			
Dataset Size	Number of Points	PCI-E Transfer Time	Execution Time (CPU)	Execution Time (GPU)	Kernel Speedup	Execution Time (CPU)	Execution Time (GPU)	Kernel Speedup	Aggregate Speedup
256	65536	54522	496	35	14.17	1082	1242	0.87	0.02
512	262144	55306	1987	53	37.49	4133	1503	2.74	0.10
1024	1048576	57904	7868	113	69.62	15555	3891	3.99	0.37
2048	4194304	67182	31045	319	97.31	57608	9689	5.94	1.14
4096	16777216	103664	123279	1151	107.10	174586	31357	5.56	2.18

We achieved about 107x speedup in the kernel of low-frequency component extraction and 6x in the kernel of connected component labeling with respect to the Intel Core2Duo processor, at most. The results show that data transfer time occupies the biggest proportion of total CUDA execution time. Hence, we achieved about 2x aggregate speedup which is calculated via this formula;

$$\text{Aggregate Speedup} = \text{Exec. Time of Sequential Algorithm} / (\text{Transfer Time} + \text{Exec. Time of Kernel}) \quad (1)$$

5. Conclusions and Future Research

In this study, the CUDA implementations of extraction of low-frequency component and connected component labeling algorithms are presented which are essential sub-algorithms in WaveCluster algorithm. Together with these

sub-algorithms, the lookup phase can be easily implemented using constant memory which is fast and cached on the device.

The reported results demonstrate that kernel algorithms expose good speedup values as dataset size increase (107x speedup in the kernel of low-frequency component extraction and 6x in the kernel of connected component labeling). Besides, as a time complexity of our algorithms, the execution times of both CUDA algorithms scales nearly linear with the number of points in the used dataset, as shown in the figure below. It is also observed that the data transfer time between CPU and GPU may introduce a considerable latency delay which is known as the main bottleneck on GPU computation.

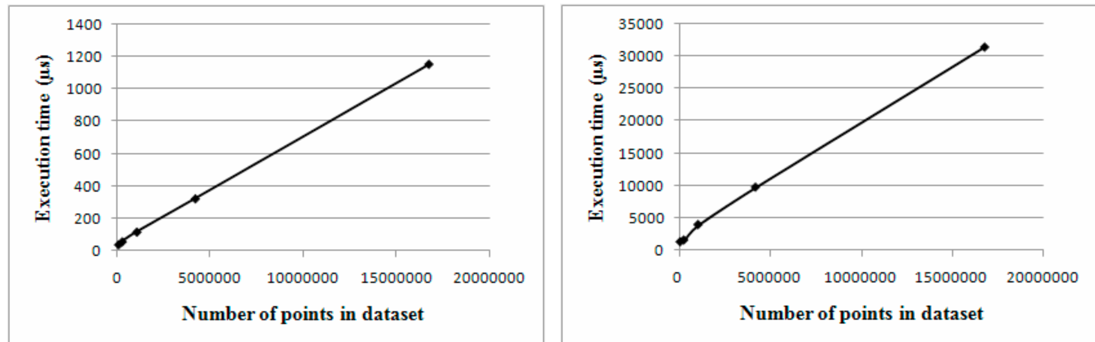

Fig. 3. Execution times of the kernel (in microseconds) for varying number of points in the dataset (a) Extraction of Low Frequency Component; (b) Connected Component Labeling

Further investigation studies on the behavior of our CUDA algorithms under different cluster shape complexities for varying scale values with detailed comparisons is planned as future study. Other algorithmic approaches to achieve better speed-up values for our developed WaveCluster algorithm on CUDA device will be another concern. Spectacular increasing amount of data and high demand to process this data in a fast and efficient way makes the GPU computation as a good promised solution due to its tremendous computational power.

References

1. G. Sheikholeslami, S. Chatterjee and A. Zhang, WaveCluster: A multi-resolution clustering approach for very large spatial databases, Proceedings of the 24rd International Conference on Very Large Data Bases, 1998, 428-439.
2. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, A performance study of general-purpose applications on graphics processors using cuda, J. Parallel Distrib. Comput. 68 (10) (2008) 1370-1380.
3. Nvidia, Cuda Programming Guide 3.0, 2010, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.
4. S. H. Adil and S. Qamar, Implementation of association rule mining using CUDA, International Conference on Emerging Technologies, 2009.
5. N.S.L. Phani Kumar, S. Satoor and I. Buck, Fast parallel expectation maximazation for gaussian mixture models on gpu using cuda, 11th IEEE International Conference on High Performance Computing and Communications, 2009.
6. A. A. Yıldırım and C. Özdoğan, Parallel WaveCluster: A linear scaling parallel clustering algorithm implementation with application to very large datasets, Submitted to Journal of Parallel and Distributed Computing, Elsevier, (2010).
7. J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable parallel programming with cuda, Queue 6 (2) (2008) 40-53.
8. E. J. Stollnitz, T. D. DeRose and D. H. Salesin, Wavelets for Computer Graphics: A Primer, Part 1, IEEE Comput. Graph. Appl. 15 (3) (1995) 76-84.
9. K. A. Hawick, A. Leist and D. P. Playne, Parallel graph component labelling with gpus and cuda, Massey University, Tech. Rep. CSTN-089, 2009.
10. L. Shapiro, G. Stockman, Computer Vision, Prentice Hall, 2001.