Journal of Electrical Engineering & Technology Vol. 6, No. 4, pp. 573~579, 2011 **573**
DOI: 10.5370/JEET.2011.6.4.573

High-Performance Korean Morphological Analyzer Using the MapReduce Framework on the GPU

Shiwon Cho* and Dong-Wook Lee†

Abstract – To meet the scalability and performance requirements of data analyses, which often involve voluminous data, efficient parallel or concurrent algorithms and frameworks are essential. We present a high-performance Korean morphological analyzer which employs the MapReduce framework on the graphics processing unit (GPU). MapReduce is a programming framework introduced by Google to aid the development of web search applications on a large number of central processing units (CPUs). GPUs are designed as a special-purpose co-processor. Their programming interfaces are typically formulated for graphics applications. Compared to CPUs, GPUs have greater computation power and memory bandwidth; however, GPUs are more difficult to program because of the design of their architectures. The performance of the Korean morphological analyzer using the MapReduce framework on the GPU is evaluated in comparison with the CPU-based model. The proposed Korean Morphological analyzer shows promising scalable performance on distributed computing with the GPU.

Keywords: GPGPU, MapReduce, Korean morphological analyzer, Natural language processing, Distributed processing

1. Introduction

With the recent increase in the demand for scientific analyses that require intensive computation and data compilation, developing a parallel and distributed framework that can process the large data is of utmost importance. Growing interest in efficient parallel algorithms and frameworks in the field of scientific data analyses, such as bioinformatics and information retrieval, has been observed. Parallel data analysis and processing systems are designed to take advantage of shared pools of processors to increase performance. Generally, data are scattered over individual processors and computed using a message passing or grid services library. This process is repeated until the desired results are obtained. The huge volume of data is not the only cause of these exhaustive, compute-intensive operations. The complexity of this process and the variety of available computing resources require the provision of a generic framework for developers to implement tasks easily, correctly, and efficiently.

In 1971, Intel Corporation released the Intel 4004, the first central processing unit (CPU), and the first commercially available microprocessor. Intel 4004 employed a 10 μm silicon-gate enhancement load pMOS technology and could execute approximately 92,000 commands per second. Thirty years after, CPUs have evolved into processors with the capacity to handle hundreds of millions of instructions per second with 100 million transistors. The performance of CPUs has developed rapidly in accordance with Moore's law. The trend shows that the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years. This development continues to the present, although the trend has shifted from developing CPUs to developing multi-core processors. Instead of being used to improve CPU performance, many transistors are currently used to create multi-core processors. In the 1990s, a 10 GHz processor was anticipated for release by the 2010s. However, Intel NetBurst had a critical limitation which prevented realization of this kind of processor. NetBurst had severe heating problems caused by high clock speeds which jeopardized its capacity to improve performance. This problem is mitigated in multi-core processors which, unlike Intel NetBurst, have higher thermal design powers. Thus, the development of most modern CPUs has been refocused on increasing the number of processor cores. While single-thread performance has remained important, interest in parallel processing has grown in recent years. Many resources such as processors, programming and operating systems, and applications are used to find ways to take advantage of parallel processing.

Graphics processing unit (GPU) is emerging as a promising computing device. Traditionally, the functionality of the GPU has been very limited. For many years, the GPU was used mainly to accelerate certain parts of the graphics pipeline. Modern GPUs demonstrate faster

† Corresponding Author: Division of Electronics and Electrical Engineering, Dongguk University, Korea. (dlee@dgu.edu)
* Division of Electronics and Electrical Engineering, Dongguk University, Korea. (stsolaris@gmail.com)
Received: September 11, 2010; Accepted: February 26, 2011

performance than typical CPUs when performing special computations such as processing simple graphics while acting as an auxiliary processor. The desktop quad-core processor of Intel is composed of four cores and the modern GPU. The architecture is relatively simple, containing just over 100 cores. Data-parallel co-processors that demonstrate particularly high performance can be incorporated into new, multi-core CPUs to create a more powerful, heterogeneous computing solution. Data parallel processors such as GPUs are designed to provide certain types of performance, thus enabling their utilization in various domains such as high-performance computing. General-purpose computation on graphics processing units (GPGPU) programming is used to achieve mass processing in a single computer.

Processors now are much faster than they were decades ago. The number of cores is consistently increasing and the GPU has a greater number of cores than multi-core processors. However, programming tools and methodology are not much different from those of decades ago. Programming methodology has not yet changed and has failed to take full advantage of parallel processors. The demand for parallel programming, which optimizes multi-core processors, has increased. GPUs are designed specifically for graphics, and thus are extremely restrictive in terms of operations and programming. GPUs are only effective at addressing problems that can be solved using stream processing and their hardware can only be used in specific ways.

The preliminary research focuses on the parallelism of multiple CPUs or multiple cores within a single CPU. Inspired by the CPU-based MapReduce frameworks, we develop a heterogeneous architecture and implement the MapReduce framework using the GPU and MPI called Hyper Stream.

This paper is divided into the following sections: a brief overview of the GPU, related literature on the GPU with MapReduce framework and semantic analysis model, design and implementation of Hyper Stream framework and Korean morphological analyzer based on Hyper Stream framework, evaluation of the proposed Korean morphological analyzer using Hyper Stream framework, and conclusion.

2. Background and Related Literature

In this section, we introduce the GPU architecture, discuss related works on the GPU, and review the MapReduce framework.

2.1 MapReduce

This section summarizes the basic principles of the MapReduce programming model. MapReduce, proposed by Google for large-scale data processing in a distributed computing environment, is a parallel programming technique derived from functional programming concepts [1]. MapReduce has been applied to various domains such as data-intensive scientific analyses, information retrieval, machine learning, and bioinformatics. Fig. 1 shows the data flow and programming model of the MapReduce framework.

MapReduce is a framework for parallel and distributed processing of batch jobs on a large number of compute nodes. Each job is based on two functional programming primitives, namely, map and reduce. They are defined as follows:

Map: $(k_1,v_1) \rightarrow (k_2,v_2)^*$
Reduce: $(k_2,v_2^*) \rightarrow (k_2,v_3)^*$

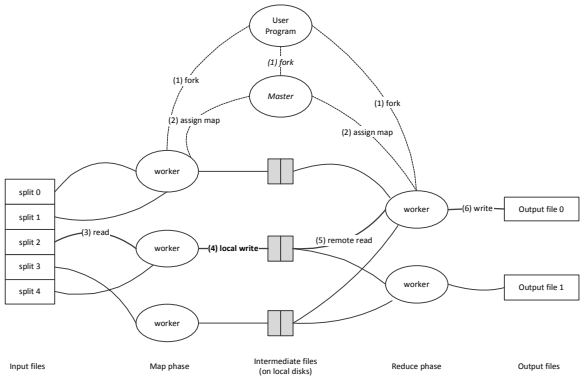

Fig. 1. MapReduce programming model

The map function takes input *key/value* pairs (k_1,v_1) and produces a list of *key/value* pairs (k_2,v_2). The map phase partitions the input data by associating each value with a key. The reduce function takes all values associated with the same key and produces a list of *key/value* pairs independently. Data are processed based on *key/value* pairs: the map function processes a *key/value* pair and produces a set of new *key/value* pairs; the reduce function merges all intermediate values with the same key to derive the final results. Programmers implement the application logic using these two primitives. The parallel execution of each primitive is managed by the system runtime. Computational processing can occur on data stored either in a file system or within a database. The MapReduce programming model is inspired by functional languages and targets data-intensive computations. The input data format is application-specific and user-specified. The output is a set of *key/value* pairs. The developer expresses an algorithm using two functions: map and reduce.

The following sample code represents a program written using MapReduce. This program counts the number of occurrences of works in a collection of documents.

```
map (key, value):
     // key: line number
     // value: triple
     emit(value.subject, blank);     // emit a blank value, since
     emit(value.predicate, blank);   // only amount of terms
```

```
matters
     emit(value.object, blank);
reduce (key, iterator values):
     // key: triple term (URI or literal)
     // values: list of irrelevant values for each term
     int count=0;
     for (value in values)
         count++; // count number of values, equaling occurrences
         emit(key, count);
```

2.2 General-Purpose Computation on Graphics Processing Units (GPGPU)

GPGPU is also known as GPU Computing [2]. GPUs are high-performance multiple core processors capable of very high computation and data throughput. GPUs, which were previously designed for computer graphics and difficult to program, have become general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C.

The first popular GPUs, designed for 3D graphics, used a pipeline that provided only fixed functionality. The recent addition of programmability allows GPUs to target a broad range of application domains other than 3D graphics. The modern GPU contains over 100 processors and is no longer a 3D pipeline with some programmable elements; instead, it is a programmable parallel processor with some 3D fixed-function hardware. GPGPU focuses on the replacement of pixel-operation programs with general-purpose computational programs.

GPU vendors have provided the programming languages including graphics application programming interfaces (APIs) such as OpenGL and DirectX and have proposed GPGPU languages such as nVIDIA Compute Unified Device Interface (CUDA) and AMD/ATI Stream SDK. Unfortunately, GPU architectural details are highly vendor-specific and generally insufficient. Moreover, the lack of high-level programming abstractions in many of the non-graphics APIs limits the scalability and portability of programs on new architectures.

The GPU architecture model is illustrated in Fig. 2. A GPU is a special purpose processor, known as a stream processor, specifically designed to perform a very large number of floating point operations in parallel. Typically, current high-end cards contain gigabytes of dedicated memory and several hundreds of processors running thousands of threads all dedicated to performing floating point math. The GPU consists of many multi-processors and a large amount of device memory. At any given clock cycle, each processor of a multi-processor executes the same instruction but operates on different data. The device memory has high bandwidth and high access latency. Moreover, the GPU device memory communicates with the main memory and cannot directly perform the disk I/O.

Stream processing is one of the most common features of the GPU and multi-core. It is a technique used to achieve a limited form of parallelism known as data level parallelism. The concepts behind stream processing

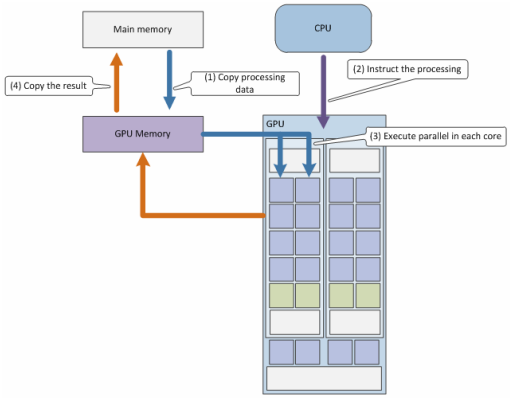

Fig. 2. GPU architecture model

originated from the mainstream of the supercomputer. Applications running on a stream processor can use multiple computational units, such as the floating point units on a GPU, without explicitly managing allocation, synchronization, or communication among the units. Not all algorithms can be expressed in terms of a data parallel solution. However, for algorithms that can be expressed, significant performance gains can be achieved by running them on a GPU and taking advantage of the massive parallelism of the device.

2.3 Open Computing Language (OpenCL)

OpenCL is a new standard for task-parallel and data-parallel heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL is independent of GPU type and provides access for any application to the GPU for non-graphical computing 0.

The need for increased computational performance in software development has led to the use of heterogeneous computing with GPUs and other processors for arithmetic-intensive data-parallel tasks. OpenCL supports a general-purpose and massively multithreaded parallel computing architecture. OpenCL provides a programming environment similar to multithreaded C/C++ and easy-to-use abstractions. More recent general-purpose interfaces include nVIDIA's CUDA 04] and the new standard OpenCL. These interfaces provide a more natural programming environment. In particular, they allow integer variables, pointer manipulation, and arbitrary memory reads and writes on the GPU cores.

However, it is very important to design software interfaces that can survive hardware changes because new hardware cease to be state of the art in only a few years, yet parallel software remain useful for decades. We focus primarily on OpenCL, which is emerging as the dominant interface for scientific GPGPU programming. By providing a common language as well as common programming interfaces and hardware abstractions, OpenCL allows developers to accelerate applications with task- or data-parallel computations in a heterogeneous computing

environment consisting of the host CPU and any attached OpenCL devices.

3. Design and Implementation

In this section, we present the design and implementation of a Korean morphological analyzer using Hyper Stream. The Message Passing Interface (MPI) is a standard API for distributed-memory parallel CPU programming. MPI is used by parallel applications on hardware ranging from multi-core laptops to super-computers. Correspondingly, many implementations of MPI exist from vendor-tuned high-performance imple-mentations to open-source implementations such as OpenMPI [9] and MPICH2 [10]. Hyper Stream uses OpenCL to provide the interface between CPU and GPU.

3.1 Design of Hyper Stream

The MPI on data-parallel architectures and heterogeneous computing platforms has a unique problem set and philosophy; principles from CPU-based message passing do not necessarily transfer over. Perhaps, the largest problem for GPU-based MPI is maintaining flexibility and dynamic communication. Developers should not be forced into a single model of communication and/or computation. While CPUs serve as first-class computa-tional and communication resources simultaneously, GPUs do not have this capability. Programming parallel GPU machines is difficult, at least in part because parallel and individual GPU programming is difficult to undertake. However, a small class of GPU applications are easy to parallelize because minimal or no communication is required during the application run. With the large computing capacity of GPUs, it is likely that GPU-based clusters will become important in designing high-performance computing platforms [Fig. 3].

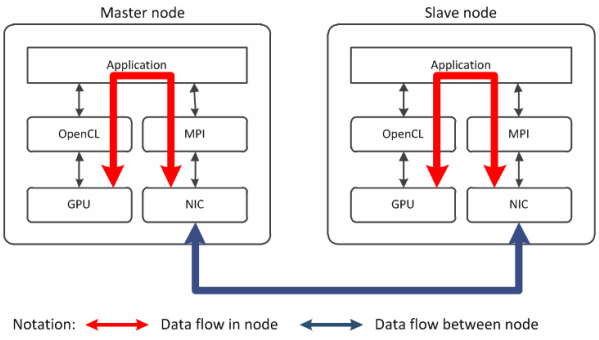

Fig. 3. Master and slave node decomposition of Hyper Stream

A typical MapReduce framework includes a master server that handles job scheduling and resource management for a number of back-end resources. The front-end node is a general purpose multi-core server with a large amount of RAM, which acts as cluster master (or manager). The back-end compute node is a general-purpose multi-core server with GPUs. The master distributes jobs to the compute nodes (or slave nodes) and handles job scheduling. The compute node processes the input data on the GPUs according to the request of the master. The system architecture of Hyper Stream is shown in Fig. 4.

The master and all the other compute nodes are connected via a high-speed network, e.g., Gigabit Ethernet. Application data are hosted on a distributed file system such as PVFS [11] or NFS. The file service can be hosted either on a dedicated server or in less demanding setups on the master. The master is responsible for invoking the jobs at the compute nodes, distributing data and allocating work between compute nodes, and providing other support services as the front-end of the cluster.

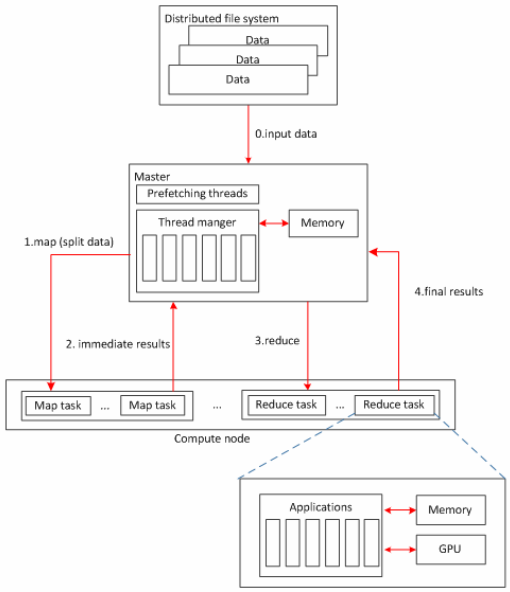

Fig. 4. System architecture of Hyper Stream

3.2 Korean Morphological Analyzer

Korean is one of the most complex languages to perform computational analysis on. Morphological analyzer is one of the most essential parts in natural language processing (NLP). A variety of approaches have been proposed for Korean morphological analysis 07]. Previous research focused on accuracy and faster algorithm within a single machine 07]. Our proposed model focuses mainly on voluminous data processing. To meet the scalability and performance requirements of these data analyses, efficient parallel algorithms and frameworks are essential.

In Korean, the word phrase (어절; *eojeol*), which is the element of a sentence, is separated by a space. Korean is one of the languages that is partially free of word order. Most of the words are free in terms of order, although there are

restrictions for several words. For example, verb and verbal endings must always come in the final position of a sentence and adjectives must always precede the modifying noun. Therefore, dependency grammar provides a clear description of word order than structure grammar in Korean.

As the Internet expands, the size of documents posted on the Internet is rapidly increasing. Therefore, high processing speed has become more crucial than ever before. MapReduce framework is a suitable solution to accelerate processing performance. Thus, we propose the Korean morphological analyzer based on Hyper Stream, a GPU-based MapReduce framework.

Tokenization is a process of breaking a stream of text into words, phrases, symbols, or other meaningful elements called tokens. Korean morphological analyzer processes a sequence of tokens into a list of morphemes in map function. A morpheme is composed of phonemes, the smallest linguistic unit of sound, and by graphemes, the fundamental unit of a written language. If the morphological analyzer fails to analyze a token, the unrecognized token is tagged with the *unknown pattern*. The unknown patterns are caused by a new word or user errors such as grammatical errors, word spacing, and mistyping. Each map function creates an intermediate pair of tokens as the key and a result of the morphological analyzer as the value. Meanwhile, the reduce function creates a list of unique tokens as the key. The value of reduce function is the frequency count and the result from the morphological analyzer.

Korean morphological analyzer has four dictionaries: *pre-analyzed dictionary*, *morpheme dictionary*, *run-time cache dictionary,* and *unknown pattern dictionary*.

- The *pre-analyzed dictionary* contains the results that correspond to the morpheme analysis with high-frequency used tokens to avoid computational overheads.
- The *run-time morphological dictionary* keeps the candidate values of morpheme analysis and segmentation of morphemes.
- The *run-time cache dictionary* and *unknown pattern dictionary* are temporary dictionaries that store the results of morpheme analysis and segmentation of morphemes in run time. If an analysis task (morpheme analysis and segmentation of morphemes) returns success, results are stored in the *run-time cache dictionary*; otherwise, results are deposited in the *unknown pattern dictionary*.

If the memory of the GPU is large enough, all the dictionaries can be loaded into the GPU memory for better performance.

Fig. 5 shows the overview of the Korean morphological analyzer which uses the GPU.

The algorithm of the proposed Korean morphological analyzer is shown in Fig. 6. Merge function stores results of the reduce function at the cache dictionary, where data are sorted by frequency count. The pre-analyzed dictionary can reduce the computational overhead such as segmentation of morphemes and the number of run-time morphological dictionary lookups [7]. If the pre-analyzed dictionary has no matching token, the morphological analyzer processes a token based on the run-time morphological dictionary. The result from the morphological analyzer is subsequently stored in the run-time cache dictionary. The morphological analyzer can skip repetitive tasks such as pre-analyzed dictionary in run time because the results of previous steps are stored in the run-time cache dictionary. If the result of the morphological analyzer is an unknown pattern, it is stored in the unknown pattern dictionary. This will allow the morphological analyzer to skip the stored token.

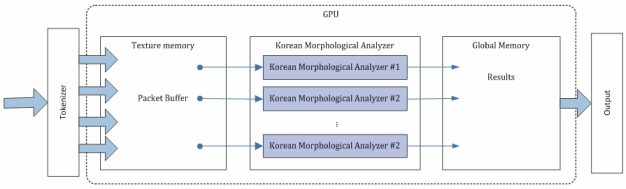

Fig. 5. Overview of the Korean morphological analyzer which uses the GPU

After the completion of the analysis task, analyzing various patterns such as new words and errors that can be collected from the cache dictionary and the unknown dictionary is the nest important step. The cache dictionary is used to update the pre-analyzed dictionary and the unknown dictionary is used to update the morphological dictionary.

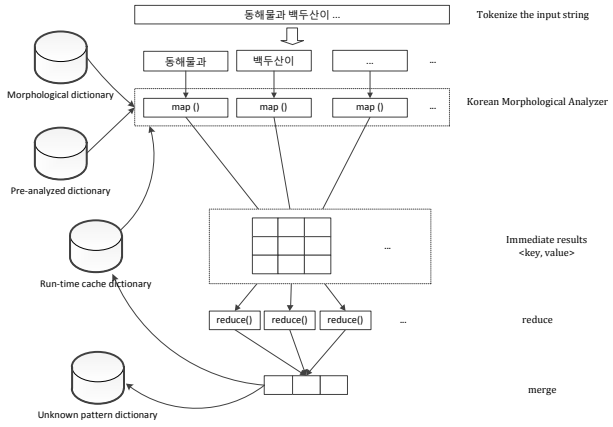

Fig. 6. Parallelized Korean morphological analyzer using Hyper Stream

4. Evaluation

This section presents the evaluation of the proposed Korean morphological analyzer. Evaluation is conducted by measuring the elapsed time for analysis speed under the configurations set.

4.1 Experimental Configuration

For performance evaluation, we prepare two configurations: CPU+MPI and GPU+MPI models. Text documents from blog posts are collected for testing. Table 1 shows the information from the test set. Documents from blogs and news are collected using the Web crawler. Some abusing documents such as SPAM are excluded from the collected document set [Table 2]. The speedup is defined as the ratio of the elapsed time with CPU+MPI to the elapsed time with GPU+MPI.

Our configurations are tested using a cluster with four nodes; each node is a PC server with two Intel Xeon CPU E5520@2.26GHz, nVIDIA NVS 295, and 16 GB physical memory. The operating system is Ubuntu Linux 9.10 (64 bit). The distributed file system server using Linux and PVFS is employed. All the nodes are connected through 1 GB/s networks. The nVIDIA NVS 295 Graphic card has 256 MB memory. Thus, we configure empirically the size of both cache dictionary and unknown pattern dictionary to 4 MB. The pre-analyzed dictionary and pre-analyzed dictionary are loaded into the GPU memory. These configuration conditions are adjusted, as needed, by the computing environment.

Table 1. Document set

Number of documents	Size	Source
8,878,128	14,633,952 (14 GB)	Blog Posts, News

Table 2. Information of dictionary

Dictionary	Number of patterns	Size
Pre-analyzed dictionary	534,513	23,692 KB
Morphological dictionary	321,810	19,390 KB

4.2 Results

We have tested the Korean morphological analyzer which uses the Hyper Stream, a MapReduce framework on the GPU. The number of nodes was scaled. The Korean morphological analyzer showed significant speedups on the GPU+MPI model compared with the CPU+MPI model. Significant performance improvement due to caching effects was obtained. Table 3 shows the elapsed time of the proposed Korean morphological analyzer while Fig. 7 shows the relative speedup performance of a parallelized Korean morphological analyzer. In the single node machine, the GPU+MPI model is approximately 40% faster than the CPU+MPI model. As the number of node increases, performance becomes significantly effective up to 60%. Since Hyper Stream provides significant speedups with MPI and multithreading, significant performance on a large number of nodes was expected.

The average accuracy of the proposed Korean morphological analyzer is 81%. The majority of error patterns are due to a new word, mistyping, non-Korean text, and unrecognized data in abusing documents such as SPAM. The unknown pattern dictionary can collect error patterns; thus, the accuracy of Korean morphological analyzer is expected to improve after the errors are fixed and new patterns are updated [Table 3].

Table 3. Elapsed time

Number of nodes	CPU+MPI	GPU+MPI	Speedup
1	54.35 min	31.52 min	42%
2	31.24 min	15.37 min	51%
3	22.34 min	10.15 min	55%
4	18.48 min	7.42 min	60%

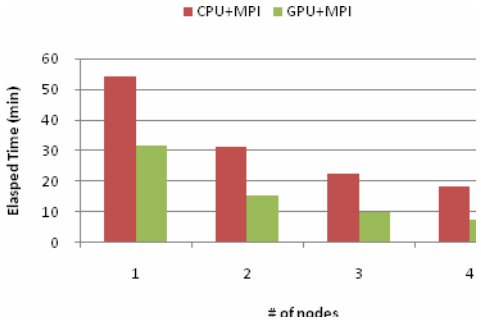

Fig. 7. Performance speedup of a parallelized Korean morphological analyzer

5. Conclusion

We have presented the design, implementation, and evaluation of a parallelized Korean morphological analyzer using GPU. GPUs are designed as a special-purpose co-processor and their programming interfaces are typically designed for graphics applications. Compared to CPUs, GPUs have greater computation power and memory bandwidth. Although GPU is more difficult to program because of the design of its architecture, it has emerged as a commodity platform for parallel computing. MapReduce, the other essential component of the proposed analyzer, is a programming framework distributed to aid the development of Web search applications on a large number of CPUs. MapReduce has been a successful solution in facilitating the development of complex and performance-centric analysis tasks. In our experiments, the performance of the Korean morphological analyzer is 60% faster than that of the CPU+MPI model. Based on this, future studies should be versatile, conducted to overcome the variety of linguistic expressions such as grammatical errors, new words, word spacing, and mistyping.

Although the speedup efficiency is partly limited by the hardware, especially the low-end GPUs, it is, to some extent, available and could be applied to more high-end GPU clusters more efficiently. Additionally, efficient and

faster GPUs mean more enormous data and better speedup efficiency. OpenCL would also be useful in parallel computing with GPU programming interfaces. The nVIDIA CUDA has shown better performance than OpenCL so far. Nevertheless, OpenCL will provide developers with a free, uniform, and effective platform which users can employ to develop reusable programs that are independent of hardware. We expect that it will be the industrial standard in the near future.

Acknowledgements

This work was supported by the research program of Dongguk University.

References

[1] MapReduce: Simplified Data Processing on Large Clusters, http://labs.google.com/papers/mapreduce.html
[2] GPGPU.org, http://gpgpu.org/
[3] OpenCL, http://www.khronos.org/opencl/
[4] nVIDIA CUDA, http://developer.nvidia.com/object/cuda.html
[5] CUDA, *Wikipedia*, http://en.wikipedia.org/wiki/CUDA
[6] Seung-Shick Kang and Yung Taek Kim, Syllable-based Model for the Korean Morphology. *The 15th International Conference on Computational Linguistics*, pp. 211–226, 1995
[7] Seung Hyun Yang and Young-Sum Kim, A High Speed Korean Morphological Analysis Method based on Pre-Analyzed Partial Words, *Journal of the Korean Information Science Society: Software and Application*, Vol. 27, No. 3., pp. 290–301, 2000
[8] Kwangseob Shim and Jaehyung Yang, MACH : A Supersonic Korean Morphological Analyzer, *Proceedings of the 19th International Conference on Computational Linguistics (COLING-2002)*, pp. 939–945, 2002
[9] OpenMPI, http://www.open-mpi.org/
[10] MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/
[11] Parallel Virtual File System (PVFS), http://pvfs.org/

Shiwon Cho is a Ph.D. candidate in the Division of Electronics and Electrical Engineering at Dongguk University. He received his B.S. and M.S. degrees from the Department of Electrical Engineering, Dongguk University. His research interests include GPGPU, general digital signal processing, parallel distributed processing, and natural language processing.

Dong-Wook Lee received his B.S. and M.S. degrees in Electrical Engineering from the Seoul National University, Seoul, Korea in 1983 and 1985, respectively. He obtained his Ph.D. degree in Electrical Engineering from the Georgia Institute of Technology, Atlanta, Georgia in 1992. From 1992 to 1993, he worked with Samsung Data Systems Co., Ltd. Currently, he is a professor in the Division of Electronics and Electrical Engineering at Dongguk University, Seoul, Korea. His research interests include general digital signal processing and digital communication.