Multi-stream Parallel String Matching on Kepler Architecture

Nhat-Phuong Tran[1], Myungho Lee[1,*], Sugwon Hong[1], and Dong Hoon Choi[2]

[1] Department of Computer Science and Engineering, Myongji University,
38-2 San Namdong, Cheo-In GuYong In, Kyung Ki Do, Korea 449-728
[2] Korea Institute of Science and Technology Information (KISTI),
245 Dae Hak Ro, Yu Seong Gu, Daejeon, Korea 305-806

Abstract. Aho-Corasick (AC) algorithm is a commonly used string matching algorithm. It performs multiple patterns matching for computer and network security, bioinformatics, among many other applications. These applications impose high computational requirements, thus efficient parallelization of the AC algorithm is crucial. In this paper, we present a multi-stream based parallelization approach for the string matching using the AC algorithm on the latest Nvidia Kepler architecture. Our approach efficiently utilizes the HyperQ feature of the Kepler GPU so that multiple streams generated from a number of OpenMP threads running on the host multicore processor can be efficiently executed on a large number of fine-grain processing cores. Experimental results show that our approach delivers up to 420Gbps throughput performance on Nvidia Tesla K20 GPU.

Keywords: string matching, Kepler GPU, multi-stream, HyperQ, multithreading.

1 Introduction

Aho-Corasick (AC) algorithm [1] is a multiple patterns matching algorithm which can simultaneously match a number of patterns for a given finite set of strings (or dictionary) against a given input data. The AC algorithm is commonly used in various applications such as network intrusion detection [16], [17], genome/protein matching for bio-sequence analysis [12], [15], among many others. In order to speed up the string matching operations and meet the real-time performance requirement imposed on these applications, achieving high performance for the AC algorithm is crucial.

Recently, the Graphic Processing Unit (GPU) is becoming increasingly popular for various applications. The architecture of the GPU has gone through a number of innovative design changes in the last decade which have drastically increased the peak floating-point throughput performance (flops) [5], [6]. Although the AC algorithm is not floating-point intensive, it can be benefitted by many promising architectural characteristics of the GPU. A GPU has a large number of (fine-grain) cores where massive parallel pattern matching operations for the AC algorithm can be performed in parallel. A GPU also provides high memory bandwidths. Thus it can

* Corresponding author.

James J. (Jong Hyuk) Park et al. (eds.), *Mobile, Ubiquitous, and Intelligent Computing*, Lecture Notes in Electrical Engineering 274,
DOI: 10.1007/978-3-642-40675-1_47, © Springer-Verlag Berlin Heidelberg 2014

feed the input data and the reference pattern data at a high rate for possible matches. On the other hand, a GPU has a complicated memory hierarchy whose efficient use has a major effect on the application's performance and is mostly under the programmer's control. Therefore, we need sophisticated parallelization techniques to achieve high performance for the AC algorithm.

In this paper, we present a multi-stream based parallelization approach for the AC algorithm on the latest Nvidia Kepler GPU. Our approach efficiently utilizes the HyperQ feature of the Kepler GPU so that multiple streams generated from a number of OpenMP threads running on the host multicore processor can be efficiently distributed and executed on a large number of fine-grain processing cores. furthermore, it also exploits the high degree of the on-chip parallelism and the complicated memory hierarchy of the Kepler GPU in order to maximize the throughput performance. Experimental results on Nvidia Tesla K20 GPU based on Kepler GK110 architecture along with multicore host processor (Intel Xeon E5-2650) show that our approach delivers up to 420Gbps throughput. Comparing with a single stream parallelization approach, it leads to 1.45-times higher throughput performance.

The rest of the paper is organized as follows: Sections 2 introduces the AC algorithm. Section 3 describes the architecture of the latest GPU including the Nvidia Kepler and its execution model. Section 4 explains our multi-stream based parallelization approach of the AC algorithm on the Kepler architecture. Section 5 shows the experimental results on Nvidia Tesla K20 GPU employing the Kepler GK110 architecture. Section 6 wraps up the paper with conclusions.

2 Aho-Corasick (AC) Algorithm

The Aho-Corasick (AC) algorithm is a multiple patterns matching algorithm which can match multiple patterns simultaneously for a given finite set of strings (or dictionary). The AC algorithm can be implemented as a Non-deterministic Finite Automata (NFA) or a Deterministic Finite Automata (DFA). The AC consists of two phases: 1) first, a pattern matching machine called the AC automaton (machine) is constructed from a finite set of patterns; 2) second, the input text data is applied to the constructed AC machine in order to find the locations that the patterns appear [1].

The AC automaton invokes three functions: a goto function g, a failure function f, and an output function *output*:

- The goto function g function maps a pair consisting of a state and an input symbol into a state or a message *fail*. The AC machine has the property that $g(0, \sigma) \neq fail$ for all input symbol σ.
- The failure function f maps a state into another state. It is consulted whenever the goto function reports a "fail".
- The output function *output* maps a set of keywords to output at the designated states.

We implement the AC algorithm as a DFA. The DFA consists of a finite set of states S and a next move function δ such that for each state s and an input symbol a, $\delta(s,a)$ is a state in S [4]. Thus, the next move function δ is used in place of both the goto function and the failure function. The output function is also incorporated in the DFA. The DFA processes the input text with length n in $O(n)$.

3 Overview of Nvidia Kepler GPU Architecture

In this paper, we use Nvidia's latest GPU based on Kepler architecture (Tesla K20) consisting of 15 Streaming Multiprocessors (SMXs) or thread blocks with the Compute Capability 3.5. Compared with the previous Fermi architecture, it has a larger number of threads and registers available on each SMX. Furthermore, it provides new architectural features such as Hyper-Q, Dynamic Parallelism, and GPUDirect [19]. The Dynamic Parallelism adds the capability for the GPU to generate new work for itself, synchronize the results, and control the scheduling of the work without the involvement of the CPU. Thus it provides the flexibility to adapt to the amount and the form of parallelism through the program execution. The GPUDirect enables GPUs within and outside a single computer to directly exchange data without going through the CPU and system memory. Thus can significantly reduce the data transfer overheads. The Hyper-Q allows multiple CPU cores to launch work on a single Kepler GPU simultaneously. This increases the utilization of the GPU, thus improve the throughput performance with the involvement of multiple CPU cores. In this paper, we utilize the Hyper-Q feature to maximize the throughput performance of the AC algorithm.

For executing programs on the Nvidia GPU, we use CUDA. CUDA programs use a hierarchy of memories of the Nvidia's GPU. They are registers and local memories belonging to each thread, a shared memory and the level-1 data cache used in a thread block (SMX in Kepler architecture) and shared by threads belonging to the block, and the global memory accessed from all the thread blocks [5, 6]. In CUDA programs, data needed for computations on the GPU is transferred from the host memory to the global memory, optionally placed in the shared memory by the programmer or automatically loaded in the L1 cache or read-only data cache by the cache controller, and used by thread blocks and thread processors through the registers. The multiple threads assigned to each thread block executes in the SIMD mode by having the same instruction managed by the Instruction Unit on different portions of data. When a running thread encounters a cache miss, for example, the context is switched to a new thread while the cache miss is serviced for the next 400 cycles or more. Thus the GPU is executing in a multithreaded fashion.

4 Multi-stream Parallelization Approach

The AC algorithm introduced in Section 2 proceeds in two steps: 1) construction of the AC pattern matching machine; 2) conducting the pattern matching operations using the machine. In typical pattern matching applications using the AC, the first phase is performed once and the second phase is repeated multiple times. In this paper, we perform the first phase of the AC sequentially using single CPU core. Then we perform the second phase on the GPU in parallel with the involvement of multiple CPU cores where multiple streams are generated. Thus the multi-stream parallelization approach is focused on the second phase.

5 Experimental Results

We implemented the parallel multi-stream AC algorithm. Our experiments are conducted on a system including the Intel multicore processor (2.0Ghz Intel Xeon E5-2650) with 20MB level-3 cache, Nvidia Tesla K20 GPU with 5GB device memory. We also used Nvidia GeForce GTX 285 GPU for performance comparison with the K20 GPU. The OS is Centos 5.5. We used input data sizes in the range of 50KB - 500MB and the numbers of patterns in the range of 100 – 20,000. In order to generate the random input data sets and the reference pattern data sets, we first collected 50GB of data from a variety of magazines such as TIME, BBC, among many others. Then we extracted the input data and the pattern data from the collected data. In all experiments conducted, we ignored the time spent in the construction phase of STT which run on single CPU core and the time to copy the input text data and the STT to the GPU device memory.

Figure 3 show the throughput performance of different input sizes when the number of patterns is fixed at 20000. The throughput increases as the data size increases, in general. This is especially true for the Kepler GPU where the number of cores has drastically increased compared with the previous generation GPUs. Our approach delivers up to 420Gbps throughput. Comparing with a single stream parallelization approach, it leads to 1.45-times higher throughput performance. The throughput on the GTX285, however, is rather flat as the data size increases: the throughput saturates around 100Gbps.

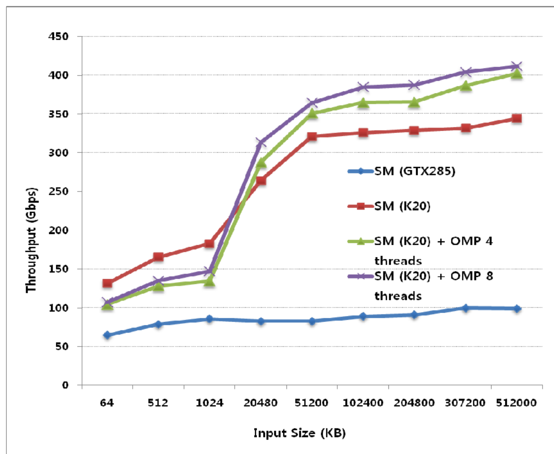

Fig. 3. Throughput (Gbps) for different input data sizes when the number of patterns is fixed at 20000

6 Conclusions

In this paper, we proposed a multi-stream parallelization approach for the AC algorithm on a GPU. The proposed approach efficiently utilizes the HyperQ feature of the Kepler GPU so that multiple streams generated from a number of OpenMP

In AC algorithm, the input text length is usually very long. Thus we partition the input text into many parts and apply the multiple patterns matching procedures in each part in parallel. In order to process the pattern matching in each part, we generate a CUDA stream. Thus multiple CUDA streams are mapped onto single Kepler GPU using the new Hyper-Q feature. In the earlier Fermi GPU, up to 16 streams are mapped to a GPU. However, all the streams are multiplexed into the same hardware work queue. Thus they are executed serially in the same queue. In the Kepler architecture, there are up to 32 hardware work queues between the host and the CUDA Work Distributor (CWD) logic in the GPU. Thus we can generate up to 32 streams and map the streams onto the same GPU to run them concurrently (see Figure 1).

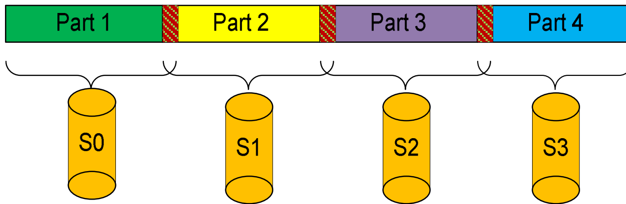

Fig. 1. Partitioning of input data into multiple parts to create multiple CUDA streams for parallel execution on the Kepler GPU using the Hyper-Q

Each stream calls the kernel function independently for matching patterns on each part of input data corresponding to each stream. Each kernel function creates a number of blocks and a number of threads per block for applying the pattern matching. Compared with the Fermi architecture where the multiple streams are time-multiplexed to share the same GPU, the Kepler architecture allows the sharing of the GPU simultaneously [19]. Thus, resources of the Kepler GPU are utilized more efficiently. In order to implement the parallel multi-stream pattern matching, we create a number of OpenMP threads on the host multicore processor each of which create a stream individually. Each thread copy parts of the input data asynchronously to the global memory while the pattern matching is performed on the GPU. Thus, the kernel execution and the data transfer can be overlapped (see Figure 2). This improves the application's performance. The pinned memory on the host memory is used for the asynchronous copy. Thus, the data in host memory must be page-locked memory.

```
#pragma omp parallel for …
for(int i = 0; i < N ; i++)
{
    cudaMemcpyAsync( dev1, host1, size1,
                     cudaMemcpyHostToDevice, stream[i] );
    kernel<<<gdim, bdim, smem, stream[i]>>>( <parameters> );
    cudaMemcpyAsync( host2, dev2, size2,
                     cudaMemcpyDeviceToHost, stream[i] );
}
```

Fig. 2. Code snippets for creating and running parallel multi-streams

threads running on the host multicore processor can be efficiently distributed and executed on a large number of fine-grain processing cores. Experimental results on Nvidia Tesla K20 GPU based on Kepler GK110 architecture along with multicore host processor (Intel Xeon E5-2650) show that our approach delivers up to 420Gbps throughput. Comparing with a single stream parallelization approach, it leads to 1.45-times higher throughput performance.

Acknowledgements. The authors would like extend their thanks to the Center for Computing and This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (Grant No: 2012-042269).

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the ACM 20(Session 10), 761–772 (1977)
2. Jacob, N., Brodley, C.: Offloading IDS Computation to the GPU. In: The 22nd Annual Computer Security Applications Conference (2006)
3. Lin, C.-H., Tsai, S.-Y., Liu, C.-H., Chang, S.-C., Shyu, J.-M.: Accelerating String Matching Using Multi-Threaded Algorithm on GPU. In: 2010 IEEE Global Telecommunications Conference, GLOBECOM 2010, December 6-10, pp. 1–5 (2010)
4. Norton, M.: Optimizing Pattern Matching for Intrusion Detection (July 2004), `http://docs.idsresearch.org/OptimizingPatternMatchingForIDS.pdf`
5. NVIDIA, CUDA Best Practices Guide: NVIDIA CUDA C Programming Best Practices Guide – CUDA Toolkit 4.0 (May 2011)
6. NVIDIA, NVidia gtx280, `http://kr.nvidia.com/object/geforce_family_kr.html`
7. OpenACC (March 2012), `http://www.openacc-standard.org`
8. OpenCL, `http://www.khronos.org/opencl/`
9. Saavedra-Barrera, R.H., Culler, D.E., von Eicken, T.: Analysis of multithreaded architectures for parallel computing. In: ACM Symposium on Parallel Algorithms and Architectures - SPAA, pp. 169–178 (1990)
10. Scarpazza, D., Villa, O., Petrini, F.: Peak-Performance DFA-based String Matching on the Cell Processor. In: International Workshop on System Management Techniques, Processes, and Services (2007)
11. Scarpazza, D., Villa, O., Petrini, F.: Accelerating Real-Time String Searching with Multicore Processors. IEEE Computer Society (2008)
12. Schatz, M.C., Trapnell, C.: Fast Exact String Matching on the GPU. Center for Bioinformatics and Computational Biology (2007)
13. Sen, S.: Performance Charaterization and Improvement of Snort as an IDS (August 2006), `http://www.princeton.edu/~soumyas/bell_labs_report_snort.pdf`
14. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating GPUs for Network Packet Signature Matching. In: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, pp. 175–184 (2009)
15. Tumeo, A., Villa, O.: Accelerating DNA analysis applications on GPU clusters. In: 2010 IEEE 8th Symposium on Application Specific Processors (SASP), June 13-14, pp. 71–76 (2010)

16. Tumeo, A., Villa, O.: Efficient Pattern Matching on GPUs for Intrusion Detection Systems. In: Proceedings of the 7th ACM International Conference on Computing Frontiers (2010)
17. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 116–134. Springer, Heidelberg (2008)
18. Volkov, V., Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebra. In: SC 2008, pp. Art.31:1–31:11 (November 2008)
19. White paper, NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK 110 The Fastest, Most Efficient HPC Architecture Ever Built, Nvidia (2012)
20. Zha, X., Sahni, S.: Multipattern string matching on a GPU. In: IEEE Symposium on Computers and Communications (ISCC), June 28-July 1, pp. 277–282 (2011)
21. Zha, X., Scarpazza, D., Sahni, S.: Highly Compressed Multi-pattern String Matching on the Cell Broadband Engine. In: IEEE Symposium on Computers and Communications (ISCC), June 28-July 1, pp. 257–264 (2011)