Platform Independent PFAC Implementations using OpenCL on Heterogeneous Parallel Computing

Rishi Acharya
NRI Institute of Research & Technology
Bhopal, India

ABSTRACT
The basic and standard multiple patterns string matching algorithm is Aho-Corasick invented by Alfred V. Aho and Margaret J. Corasick. The algorithm of Aho-Corasick can match multiple patterns simultaneously and affirmed deterministic performance under all circumstances. Various real world applications were provided by this algorithm like computational biology, intrusion detection systems, multimedia, search engine and text mining. Performances parallelization of Aho-Corasick is crucial in order to improve performance of these applications and meet with real time environments. Parallelization of Aho-Corasick can provide drastic performance under various circumstances. The parallel version of Aho-Corasick is PFAC (Parallel Failure Less Aho-Corasick). PFAC algorithm provides high degrees of parallelization and various improvements in Aho-Corasick algorithm. We are going to implement PFAC. Along with PFAC various memory access techniques will be used to implement the algorithm. Also to bring to notice that previously PFAC library is built in CUDA. These libraries are platform dependent and run on only NVIDIA GPUs. General-purpose computing on graphics processing units (GPGPU) is the utilization of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). In this paper we have various versions of PFAC using OpenCL which is platform independent and able to run on any GPU. Here we will also discuss the performance details of various PFAC versions.

Keywords
CUDA, GPGPU, OPENCL, PARALLEL COMPUTING, PFAC

1. INTRODUCTION
String matching or string searching [1] algorithms check and detect the presence of one or more known pattern sequences inside a dataset. These are the standards of many critical real world applications for their well-known applications to databases and text processing. Pattern matching algorithms are fundamental components of applications like DNA and protein sequence analysis, data mining, antivirus softwares [53], machine learning problems and security systems, such as Intrusion Detection Systems (IDS) [2] for Networks (NIDS), Applications (APIDS), Protocols (PIDS), or Systems (Host-based IDS—HIDS) [3]. All these applications perform processes on large volume of textual data and require extremely high performance to produce meaningful results in an pre-determined time. Aho-Corasick (AC) [4] algorithm is one of the most efficient and most studied, especially for text processing and security applications among all string matching algorithms. This is because of its exact, multi-pattern approach and ability to perform the search in time linearly proportional to the length of the input stream.

The essential part of today's mainstream computing systems is the graphics processing unit (GPU) [5,6,7]. There has been a remarkable improvement in the performance and capabilities of GPGPUs since last few years. The present GPGPU is not only a powerful graphics engine but also a highly parallel programmable processor, computing upon tremendous arithmetic and memory bandwidth. For the reason that GPU's rapid growth in both programmability and capability has created an interest of research community mapped with a wide range of computationally demanding, complex problems to the GPGPU. This capability of general purpose computing on the GPU, also known as GPU computing or GPGPU has established the GPU as a outstanding alternative to traditional processing units such as microprocessors in high-performance computer systems of the future.

Initially the functions performed by Graphics chips were fixed for graphic processing only. But in due course of time it became much programmable and computationally powerful. This led NVIDIA to introduce the first GPU during late 1990s. In the 1999-2000 timeframe, computer scientists and domain scientists from various fields started using GPUs to accelerate a range of various computing and scientific applications. This was the era the architecture of GPU was enhanced to GPGPU, or General-Purpose computation on GPU [7].

During this era users achieved illegal performance (over 100 x compared to CPUs in some cases), the challenge was that to include in GPGPU the use of graphics programming APIs like OpenGL and Cg to program the GPU. Though GPGPU was having tremendous capability of programming GPUs for scientific processing it got limitation for accessibility. The reorganization by NVIDIA for the potential of bringing this computational performance of GPGPU for the larger scientific community, invested in making the GPU fully programmable processing, and offered flawless experience for programmers and developers with familiar languages like C, C++, and Fortran. GPGPU computing power is growing faster than ever before. Today, some of the fastest supercomputers in the world rely on GPGPUs for advance scientific discoveries; 600 universities around the world has implemented and involved in teaching parallel computing with NVIDIA GPGPUs; and hundreds of thousands of programmers and developers are actively using GPGPUs for enormous computing processing.

The NVIDIA products like GeForce, Quadro and Tesla support GPU computing and the CUDA parallel programming model. These GPGPUs are working only on NVIDIA platform dependent environment. Developers have access to only NVIDIA GPGPUs in any platform of their choice, including the latest Apple MacBook Pro.

Tesla GPGPUs is recommended for workloads where data reliability and overall performance are critical. Tesla GPUs are designed and architected from the core to accelerate scientific and technical processing workloads. Using innovative features of the "Kepler architecture," the latest Tesla GPUs generates 3x more performance compared to the previous architecture of GPUs, more than one teraflops of double-precision floating point while dramatically advancing programmability and efficiency. Kepler is the world's fastest and most efficient high performance computing (HPC) model.

Initially Apple Inc., developed OpenCL [8,9], which holds the trademark rights, and came up with the initial proposal in collaboration with technical teams of AMD, IBM and Intel. Apple submitted this initial proposal to the Khronos Group. On 16 June 2008, the Khronos Compute Working Group was formed with representatives from CPU, GPU, embedded-processor, and software companies. To finish the technical details of the specification for OpenCL 1.0 the group members worked for five months till 18 November 2008. This technical specification was reviewed by the Khronos technical committee members and approved for public release on December 2008. [10,11]

In our research paper we proposed the three enhanced versions of PFAC algorithms which are developed in OpenCL programming language and are platform independent libraries working on heterogeneous architectures like GPGPUs.

2. PFAC (PARALLEL FAILURE LESS AHO-CORASICK)

Parallel Failure less Aho-Corasick (PFAC) algorithm is the parallel and enhanced version of Aho-Corasick algorithm. The algorithm does not follow failure function. Large number of patterns in a given data set is focused while using PFAC implemented on GPGPU using OpenCL. PFAC eliminates the thread when there is no valid state. Each thread access Deterministic Finite State Automata (DFA) from global memory in GPGPU and an input string of data set is assigned to each thread. Failed transaction or process does not require backtracking. This is more efficient and faster version of PFAC taking into account the memory access technique. Most of the threads eradicate in earlier stages of text matching process [12].

In PFAC Algorithm machine is developed without any failure function and is supporting platform independent model as well. Total number of executed threads will be equal to the number of pattern length of given text in a data set. Each thread has its own variable scanning length. If it finds a valid state in previous thread then next state will be checked by next thread. Hence a single thread may scan more than one alphabet at a time.

Figure1 represents the example of DFA for multiple patterns {HE, HER, HIS, HIM}. In this example of DFA we have not enclosed failure function of each state. As aware failure function is not required in PFAC.

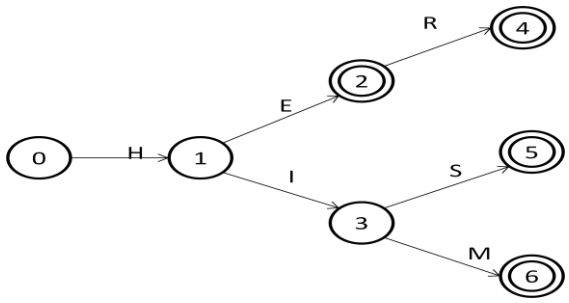

Fig 1: PFAC Machine and Thread Generation Example

3. PFAC Version1

This is our first implemented version of PFAC based on purely platform independent heterogeneous architecture. In this version global memory is accessed to contain state machine and counter for total count value. The logic of global memory access technique is described in figure 2. Global memory is common for all compute unified devices and every one can access same global memory for data set.

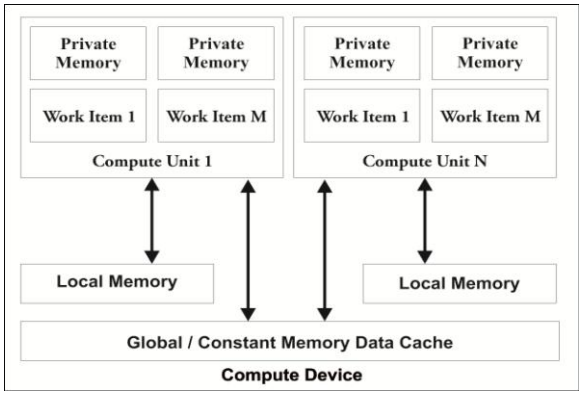

Fig 2: Different memory access technique

This version is using the basic enhancement of PFAC algorithm. This memory allocation technique is same as the malloc() function of previous versions. In this version the total number of variables used for pattern count is equal to text or pattern length. For each count a new variable is assigned. An individual count for storing number of occurrences of patterns we are searching and scanning for and then at last sum of all counts are done to calculate number of occurrences of patterns as described in figure 3.

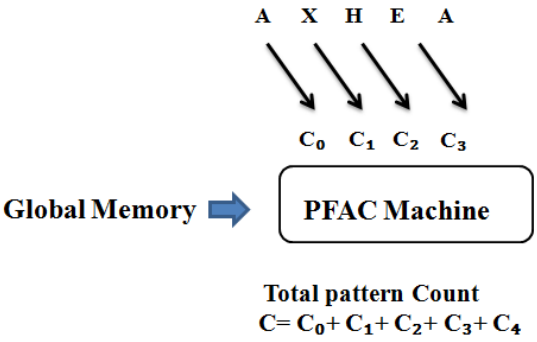

Fig 3: PFAC Version1 Concept

4. PFAC Version2

This is our second implemented version of PFAC based on purely platform independent heterogeneous architecture. It removes the limitations of PFAC version1 like the data size of version1 was limited to the size of 50 MB. In this version we have implemented an exclusive pattern counter instead of individual counter on the threads. This version ensures that only one variable will access the code at a time. This PFAC version provides synchronized access of counter. In Figure 4, counter declared in global memory is described. Here we use only single count hence synchronization is required between the threads to calculate the total number of occurrences of that patterns.

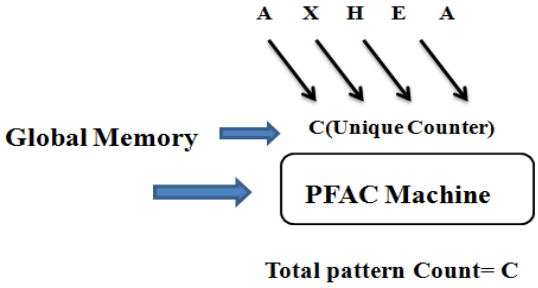

Fig 4: PFAC Version2 Concept

The main advantage of this version is to access less memory for storing count here, but as the size of file increases number of threads increases and all the threads will wait in queue and as the total number of threads will increase size of queue will increase and after a point queue will unable to handle large number of waiting threads simultaneously leading the system to crash.

This method is efficient in terms of data sets but for larger data sets this method remains inefficient.

This method is able to handle medium size data sets up to 100 MB but due to usage of synchronization variable execution time is increased and this method fails on large size data sets. To remove this limitation a new version of PFAC is developed by us.

5. PFAC Version3

This is our third implemented version of PFAC based on purely platform independent heterogeneous architecture. In PFAC version3 speedup and synchronization are the key factors included. Here we access local memory of work unit instead of global memory to keep counter and machine in local memory. Final count is calculated in global memory.

In this PFAC version where Local Count and vector is used to record the count the occurrences of patterns. Vector and DFA is placed in local memory of each thread. Vector is used to store pattern occurrences. A final counter is placed in global memory to make total count in process. Problem of system to handle large number of threads is conquered. This version of PFAC is faster than all previous versions of PFAC because vector is placed in the local memory instead of global memory and threads can access DFA from local memory. So access time of memory is decreased.

During execution after the processing of scanning and searching phase, all threads approach for global memory to make a final count.

Synchronization is also to be included so as to avoid overwriting of count variables in the array of pattern matching. And by using synchronization we safeguard our count from getting overwritten. But using synchronization some threads will be forced to halt, thus slowing the overall process. So this version of PFAC takes more time in scanning and searching the patterns than the first version of PFAC but is faster than the second version.

This version of PFAC can execute files of lager size than the first version. But again when we increase the size of the file, the number of threads also increases. And we hit our limit as number of threads is so high that the system is unable to handle so many threads at a time. Thus this version of PFAC can search patterns in larger files than the first version but still it is unable to search in files of very large sizes.

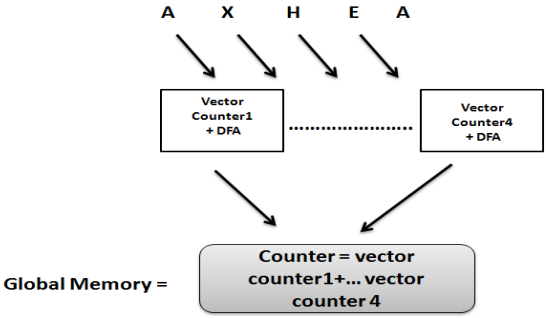

Fig 5 : PFAC Version3 Concept

6. EXPERIMENTAL RESULTS

Theses PFAC versions implemented on different Single Instruction Multiple Data (SIMD architecture and provide massive improvement in pattern matching efficiency.

6.1 Experimental Environment-

- Processor: Core i3
- RAM: 4 GB
- OS: Windows 7
- Language: Visual C++ runs on Visual Studios 2008
- GPGPU: AMD Radeon HD 6800 series (192 cores)
- Language (parallel implementation): OpenCL

6.2 Experimental Data for PFAC-Versions

6.2.1 Text File:
Text Size of 50 MB, 100 MB and 200 MB having large number of occurrences of patterns.

6.2.2 Pattern File: One File pattern length of 20 patterns.

Here we are taking total number of work items equal to text stream length of data set without setting any local workgroup size.

6.2.3 Experiment
We have implemented PFAC in three different ways:

- PFAC-Version1
- PFAC- Version2
- PFAC- Version3

6.2.4 Results- All times are execution time in milliseconds.

Table1. Execution time comparison of all PFAC Versions

Data Set(MB)	PFAC Version1	PFAC Version2	PFAC Version3
50MB	490 ms	500 ms	311 ms
100MB	980 ms	1010 ms	960 ms
200MB	1850 ms	2020 ms	1660 ms
300MB	2340 ms	2540 ms	2000 ms

7. CONCLUSION

We have implemented and analysis peculiar techniques of PFAC on GPGPU using OpenCL and discussed their issues. Here we have implemented the three versions of PFAC. These versions are platform independents and able to work on any GPGPUs. Execution time of PFAC Version3 count was fastest but unable to handle larger data sets.

8. FUTURE WORK

In near future will try to enhance the efficiency of Aho-Corasick parallel algorithm by removing the limitations of size of data set from basic versions of PFAC. We will try to perform some optimization techniques on PFAC to increase the overall throughput.

9. REFERENCES

[1] Sung-Il Oh, Inbok Lee, Min Sik Kim,"Fast filtering for intrusion detection systems with the shift-or algorithm", In the proc. of 18th Asia-Pacific Conference on Communications (APCC), pp. 869-870, Oct. 2012.

[2] Zhen LIIU, Su XU, Jue ZHANG,"Improved Algorithm of pattern matching for Intrusion Detection", 2009 International Conference on Multimedia Information Networking and Security, Wuhan, CHINA, 2009, pp.446-449.

[3] Yihui SHAN, Yuming JIANG, Shiyuan TIAN, "Improved Pattern Matching Algorithm of BMHS for Intrusion Detection", Computer Engineering, vol. 35, pp.170-173, 2009.

[4] Vidya Saikrishna, Akhtar Rasool, Nilay Khare, "String Matching and its Applications in Diversified Fields", IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, January 2012.

[5] Xinyan Zha, S. Sahni, **"**GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU", In the proc**.** of IEEE Transactions on Computers, Vol. 62, Issue 6, pp. 1156-1169, June 2013.

[6] Wu-chun Feng, Shucai Xiao," To GPU synchronize or not GPU synchronize?", In the proc. of 2010 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 3801-3804, June 2010.

[7] Yi Yang, Ping Xiang, Mantor, M., Huiyang, Zhou,"Fused GPU-GPGPU Architecture", In the proc. Of High Performance Computer Architecture (HPCA), pp. 1-12, Feb. 2012.

[8] cs.nyu.edu/courses/spring12/CSCI-GA.3033------012/lecture2.pdf

[9] http://en.wikipedia.org/wiki/OpenCL

[10] www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

[11] www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf.

[12] R. Takahashi, U. Inoue, "Parallel Text Matching Using GPGPU", in the proc. 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), pp. 242-246, Aug. 2012.