Thread divergence free and space efficient GPU implementation of NFA AC

Yogesh Charan

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

Department of Computer Science And Engineering

June, 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

Yogesh.

(Signature)

(YOGESH CHARAN)

CS13M1007

(Roll No.)

Approval Sheet

This Thesis entitled Thread divergence free and space efficient GPU implementation of NFA AC by Yogesh Charan is approved for the degree of Master of Technology from IIT Hyderabad

(M.V. Panduranga Rao) Examiner
M. V. PANDURANGA RAO
IITH

(Internal) Examiner
N.R. ARAVIND, Dept. of CSE.
IITH

(Dr. Subrahmanyam Kalyanasundaram) Adviser
Dept. of Computer Science and Engineering
IITH

(T. Bheemarjuna Reddy) Chairman
Dept. of CSE
IITH

Acknowledgements

I would like to express my sincere gratitude to my thesis adviser Dr.Subrahmanyam Kalyanasundaram for sharing expertise, constant encouragement, valuable guidance, patience. I would also like to thank my family and friends for their support and constant encouragement.

Dedication

Dedicated to my parents

Abstract

Multipattern String Matching problem reports all occurrences of a given set or dictionary of patterns in a document. Multipattern string matching problems are used in databases, data mining, DNA and protein sequence analysis, Intrusion detection systems (IDS) for applications (APIDS), networks (NIDS), protocols (PIDS), Host-based IDS, antivirus softwares, and machine learning problems. Parallel algorithm for multipattern string matching can be useful for above mentioned application because by using parallel plateforms large number of threads can be executed parallelly and these thread can search for patterns in parallel. One of the multipattern search algorithm is a Aho-Corasick (AC) is a multipattern search algorithm. AC algorithm has two versions : NFA AC and DFA AC. DFA AC and NFA AC has a matching automata to perform multipattern searching. NFA AC automata takes less memory then DFA AC automata. Many parallel implementation for AC algorithm are available. Thread divergence free GPU implementation for DFA AC algorithm is available but GPU implementation for NFA AC algorithm is not available. We have developed thread divergence free implemetation of NFA AC algorithm and we have given a space efficient version of NFA AC automata. Space requirement for our NFA AC algoritm is $log(N)$ times less then DFA AC implementation where N is number of nodes in automata. Our NFA AC implementation can store upto 2K nodes in shared memory of size 64KB on GPU and that can be very fast when compared to the DFA AC implemetation.

Contents

Declaration ii

Approval Sheet iii

Acknowledgements iv

Abstract vi

Nomenclature **viii**

1 Inroduction **1**

1.1 Background and Motivation 1

1.1.1 Multipattern String Matching Problem 1

1.1.2 Document Retrieval Problem 3

1.2 Overview of the Thesis Work 4

1.3 Thesis Overview 4

2 Related Work **5**

2.1 Multipattern String Search 5

2.2 Document Retrivel Problem 8

3 Graphics Processing Units (GPUs) Architecture and CUDA Programming Model **13**

3.1 Graphics Processing Units (GPUs) Architecture 13

3.2 CUDA Programming Model 14

3.2.1 Compiler Model 17

3.2.2 Execution Model 18

3.2.3 CUDA Memory Architecture 18

4 GPU to GPU implementation and optimization of Aho-Corasick algorithm **20**

4.1 GPU-to-GPU Implementation Strategy 21

4.2 Thread Divergence Free NFA AC algorithm for GPU 24
4.3 Space requirement optimization for AC algorithm 27
4.3.1 DFA AC space requirement 27
4.3.2 NFA AC space requirement 27
4.4 Comparision of NFA and DFA AC algorithm For GPU 29

5 Conclusion and Future work **30**

References **31**

Chapter 1

Inroduction

1.1 Background and Motivation

String matching is a problem of finding and reporting given pattern in a document. String matching problems are used in databases, DNA and protein sequence analysis, data mining, Intrusion Detection Systems (IDS) for protocols (PIDS), networks (NIDS), applications (APIDS), host-based IDS, antivirus softwares, and machine learning problems.

String matching problems can be divided in two important subproblems depending on the query definition:

1. Multipattern String Matching Problem: Multipattern String Matching problem finds all occurrences of a given set or dictionary of patterns in a text document.

2. Document Retrieval Problem: String database is a collection of text documents. Suppose we are given a string database D of text documents $d_1, ..., d_k$ with $\sum_{i=0}^{k} |d_i| = n$ and we are given an online query comprising of a pattern string P of length m. Document Retrivel Problem is to retrieve all the documents d_i in which the query pattern P occurs.

1.1.1 Multipattern String Matching Problem

Multipattern string matching is used in applications such as network intrusion detection, business analytics, digital forensics and natural language processing. For example, Snort is an open source network intrusion detection system (NIDS) and Scalpel is an open source file carver that uses multipattern string matching algorithms. Snort has a predefined collection of patterns to detect any intrusion attempt into the network. So, whenever a new packet comes into the network Snort

searches for set of patterns in the packet to detect and prevent network intrusion attempt. Snort uses Aho-Corasick (AC) [1] multipattern search algorithm to search for patterns in the incoming packets. Aho-Corasick (AC) et. al. [1] multipattern search algorithm first constructs finite state pattern matching machine from collection of predefined patterns and then uses this pattern matching machine for multipattern search in document and Runtime for this algorithm is $O(n)$ here n is size of the input text document. Snort uses AC algorithm because runtime for AC algorithm is independent of the number of patterns in the dictionary and linear in the length of the target string. Scalpel has collection of header/footer pairs as set of patterns and searches for all occurrences of these patterns in disk to extract data from a disk drive. Scalpel uses Boyer-Moore et. al. [2] algorithm to search header/footer pairs in disk. Boyer-Moore et. al. is a single pattern search algorithm. Boyer-Moore single pattern search algorithm searches for the first occurrence of a pattern string in target string and the runtime for algorithm is $O(length)$ here length is sum of target string length and pattern string length. Scalpel runs Boyer-Moore single pattern search algorithm for all header/footer pairs, so its runtime is $O(length*count)$ here $count$ is number of different patterns in the pattern dictionary and $length$ is sum of lengths of target string and pattern string. For text processing and security applications AC algorithm is most used algorithm because only this algorithm provides multipattern string matching in time linearly proportional to the length of the input document.

String matching can be done using parallel architectures like multicore, multithreaded and GPU (Graphics Processing Unit)s because input string can be divided into smaller substrings to perform pattern search in parallel. Pattern search in each such smaller substring can be done using a thread or thread block or core in parallel and that would result in high throughput. There are various types of parallel string matching algorithm are available, such as Scarpazza et. al. [3] [4]. These algorithms describe parallel version of the Aho-Corasick (AC) algorithm using deterministic finite automata for IBM Cell Broadband Engine (CBE). Zha-Sahni et. al. [5] describes parallel version of the Aho-Corasick algorithm using compressed form of the non-deterministic finite automata for IBM Cell Broadband Engine (CBE). Jacob et al. [6] searches for 16 patterns in parallel with the use of 16 GPU core and for searching these patterns within a packet and it uses Knuth-Morris-Pratt (KMP) [7] single pattern matching algorithm. This version of multipattern algorithm was developed for Snort. Huang et al. [8] and Smith et al. [9] uses GPU for network intrusion detection. Huang et. al. uses Wu-Manber et al. [10]'s multipattern search algorithm and Smith uses deterministic finite automata (DFA) and extended DFA to perform regular expression matching.

GPU architecture has two components viz. device and host. Device represents the GPU part and host represents CPU part. In GPU architecture, host or CPU acts as a master processor

and GPU or device acts as a slave processor. Device and Host have seperate memory spaces. Algorithm development for GPU can be defined based on location of the input data and location of the result/output. For example, in the GPU-to-GPU implementation of string matching problem the input document resides on the device memory and result of pattern matching on document also resides on device memory. In the Host-to-Host implementation of string matching problem the input document resides on the CPU memory or main memory and result of pattern matching on document also resides on CPU memory or main memory. There are different version of GPU implementations for AC algorithm are available. Lin et al. [11] and Tumeo et al. [12]'s GPU implementation of Aho-Corasick (AC) algorithm is a Host-to-Host GPU implementation. Lin et. al. uses one thread for each position in the input document and this thread determines whether assigned position to thread is the starting position for any pattern. Zha-Sahni et. al. [13] has developed GPU adaptations of the Aho-Corasick string matching algorithm for two cases GPU-to-GPU and Host-to-Host. Zha-Sahni et. al. [13] and Tumeo et. al. defines a thread for a specific portion of input document/string and that thread determines all possible patterns that are available in the assigned portion of input document/string. Zha-sahni et. al. and Tumeo et. al. defines portions such that they have sufficient overlap among each other so that patterns that are crossing a portion boundries can not be missed. AC algorithm first generates multipattern matching machine/automata for dictionary of patterns and then uses this precomputed multipattern matching automata to perform multipattern string matching in time linearly proportional to the length of the input data or document. Based on the definiton of multipattern matching automata, AC algorithm has two version Deterministic AC and Non-Deterministic AC. Deterministic AC has a deterministic finite matching automata and Non-Deterministic AC has a Non-Deterministic finite matching automata. Zha et. al. [13] has provided GPU implementation of Deterministic AC or DFA AC. NFA AC takes less space then DFA AC because NFA AC can be compacted. For NFA AC there is no thread divergnece free algorithm for GPU is present, we are going develop thread divergnece algorithm for NFA AC and reduce space requirement for NFA AC.

1.1.2 Document Retrieval Problem

In string matching problems target is to locate all occurrences of the pattern within the text/document here text/document is a relatively long character string then pattern. In some applications, the text is given in advance, and we may preprocess it and create an auxiliary data structure called an index for the text and then we can use this index to answer any pattern matching query more efficiently.

For example, if we use suffix tree et al. McCreight et. al. [14] a linear-space index for the text locating all occurrences of a pattern P of length $|P|$ can be done in runtime $O(|P| + occ)$ here occ is number of occurrences of pattern P in text and runtime is independent of the length of the text.

String database is a collection of text documents. Suppose we are given a string database D of text documents $d_1, ..., d_k$ with $\sum_{i=0}^{k} |d_i| = n$ and we are given an online query comprising of a pattern string P of length m. Document Retrivel Problem is to retrieve all the documents d_i in which the query pattern P occurs. Document Retrivel Problem has been studied by Muthukrishnan et al. [15]. The main issue here is that there may be many occurrences of the pattern in collection D, but the overall number of documents d_i in which the pattern occurs might be much smaller then number of documents in collection D. So, method of finding all the occurrences first and then reporting unique documents can not be efficient because of run time $O(|P| + count)$ where count is total number on document. Muthukrishnan gave an optimal $O(n)$ - space data structure which answers the document retrieval query in $O(|P| + occ)$, where occ is the number of documents which contain the pattern P. This has been a popular approach of many subsequent papers on document retrieval query eg. sadakane et al. [16] and Valimaki et al. [17] which attempted to derive compressed data structures for this problem.

1.2 Overview of the Thesis Work

There are two version of AC algorithm are available NFA AC and DFA AC. NFA AC takes less space then DFA AC because NFA AC can be compacted. Zha et. al . [13] has provided GPU implementation of Deterministic AC or DFA AC. For NFA AC there is no thread divergnece free algorithm for GPU is present. As a part of theis work we have developed GPU-to-GPU version of NFA AC algorithm which has no thread divergence. We also developed a space efficient version of NFA AC algorithm and then compared this space efficient version of NFA AC with DFA AC with respect to GPU.

1.3 Thesis Overview

Thesis chapter 2 is dedicated for related work to understand basic information about multipattern string search and document retrievel. Thesis chapter 3 gives overview about gpu and cuda architecture and cuda programming interface. Chapter 4 is about detailed analysis and solution of GPU-to-GPU based AC algorithm solution. At last in chapter we conclude our thesis work.

Chapter 2

Related Work

2.1 Multipattern String Search

Multipattern String Matching problem reports all occurrences of a given set or dictionary of patterns in a document. String matching problems are used in databases, data mining, DNA and protein sequence analysis, Intrusion detection systems (IDS) for applications (APIDS), networks (NIDS), protocols (PIDS), Host-based IDS , antivirus softwares, and machine learning problems.

First linear time algorithm for string matching was developed by Knuth-Morris-Pratt (KMP) and first multipattern search algorithm was developed by Aho-Corasick (AC). Multipattern search applications generally use two basic algorithm to preform multipattern search. These two algorithms are Boyer-Moore pattern search algorithm and Aho-Corasick (AC) multipattern search algorithm.

Suppose, there is a text T and a pattern P. Boyer-Moore searches for the first occurrence of a pattern string in target string and runtime for algorithm is $O(length)$ there *length* is sum of length of T and P. Boyer-Moore uses bad character function for pattern P, bad character function specifies how many characters to shift pattern P when current character from P and T does not match. Galil et al. [18] and Horspool et al. [19] pattern matching algorithms also use bad functions for pattern searching. There are many algorithms that use or extends Boyer-Moore pattern search algorithm for multipattern string search. For example, Baeza-Ricardo et al. [20], Commentz-Beate et al. [21]. These multipattern search algorithms extend the bad character function for P and define the bad character function for dictionary patterns.

Aho Corasick (AC) Multipattern Search Algorithm: AC algorithm uses pattern matching machine for multipattern searching in a document. Here we will see what is a pattern matching

machine for AC algorithm and how this machine can be used for multipattern matching. Suppose, P is a finite set of patterns such that $P = p_1, ..., p_k$ and T is a input string. Our goal is to report all the patterns of P which are available in T as a substring of T.

Pattern Matching Machine: A pattern matching machine is a program which takes input a string text T and return all patterns of P which are availabel in T as a substring. Pattern matching machine is a set of states and each state has a state id which is a number. Node 0 act as a root node. Pattern matching machine uses three functions: a goto function (g), a failure function (f), and an output function (out) process any node. Machine reades charater from T and makes state transitions from current state with respect to the read character. Three functions for a state are as follows :

L(q) is a function which returns string that is concatenation of characters of path from root node 0 to node q.

1. g(q,a) is a goto function which gives the state entered from state q by matching char a and there are three cases
    - if for state q edge (q, v) is labeled by a, then set g(q, a) = v;
    - g(0, a) = 0 for each character 'a' that does not label an edge out of the root node 0 So the matching machine stays at the initial state while scanning non-matching characters
    - Otherwise g(q, a) = ϕ
2. f(q) for node id q is a failure function which gives the state entered at a mismatch.
    - f(q) is the node, that is labeled by the longest proper suffix w of L(q) s.t. w is a prefix of some pattern p_i
    - a fail transition for a node does not miss any potential occurrences
    - f(q) is always defined for a node, since L(0) is a prefix of for every pattern
3. out(q) is a output function which gives the set of patterns recognized when machine is in state q

Example: suppose P = {he, hers, his, she} then matching machine is shown in Figure 2.1. These are the outputs for nodes

out(0) = ϕ, out(1) = ϕ, out(2) = ϕ,

out(3) = {he}, out(4) = ϕ, out(5) = ϕ

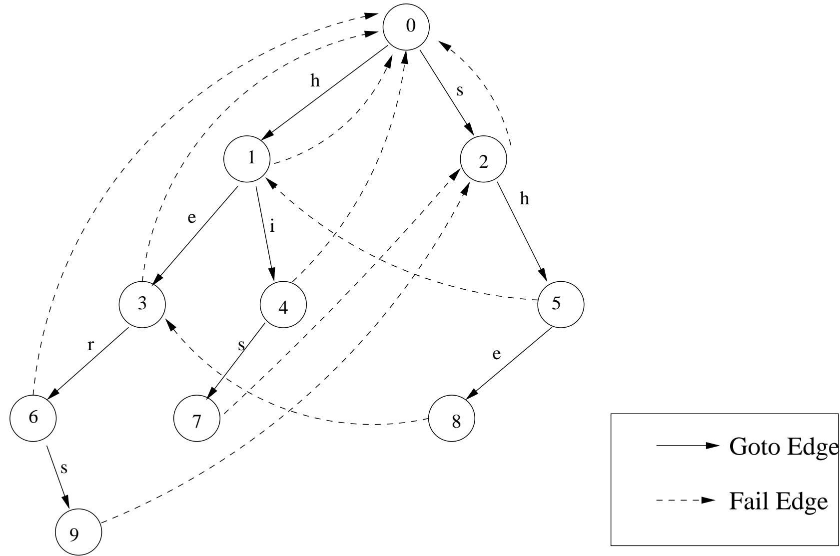


Figure 2.1: AC matching machine

out(6) = ϕ, out(7) = {his}, out(8) = {he, she} , out(9) = {hers}

Now lets see multipattern matching using these above functions. Pattern searching using pattern matching machine is as follows. Let *state* be the current state of the machine and t_i the current symbol of the input string T.

1. If g(s, t_i) = $s^{`}$ then there would be a goto transition in machine. Machine enters state $s^{`}$ and returns out($s^{`}$). Now next character of T becomes current input symbol for machine.

2. If g(s, t_i) = ϕ then the machine uses failure function f and makes failure transition. Now machine repeats this cycle with state f(s) and t_i as the current input symbol.

Algorithm 1 describes the exact procedure for multipattern search using matching machine. AC algorithm has defined two versions based on how pattern matching machine for the dictionary of input patterns is defined. These two versions are nondeterministic and deterministic multipattern matching algorithm. These versions use a finite state machine to represent the dictionary of input patterns. Deterministic version (DFA) of AC at each state has well-defined state transition function for every character in the alphabet and list of matched patterns. The multipattern search begin with defining the matching machine/automaton start state as the current state and first character in the text string T is assigned as the current character. Matching automata makes a state transition by examining the current character of T. At each step of a matching procedure a transition to the state corresponding to the current character is made and the next character of string T becomes the current character. After performing a state transition machine lists the matched patterns for the reached/next state as a output along with the position of the current character in the string T.

The number of state transitions made by the DFA while performing multipattern search in a string of length n is n. In the nondeterministic-NFA version matching automaton states have two kinds of transitions success and failure. Success transitions are defined for automata states for characters that match a pattern character and a failure transition is defined for the remaining characters. When the NFA version is used, The number of state transitions made while performing multipattern search are $2n$. The NFA version of matching automata uses less memory then DFA version of matching automata. In NFA version of matching automata states have few success transitions and can be compacted better than DFA states. AC has described how to compute the DFA and NFA for a set of patterns.

Input: A text string $T = t_1, t_2, ..., t_n$ where each t_i is an input character and M is a pattern matching machine with goto function g, failure function f, and output function out, as described above.
Output: List of patterns that is substring of T
Method.

```
0.  begin
1.      state ← 0
2.      for i ← 0 until n do
3.        begin
4.          while g(state, t_i) = fail do state ← f(state)
5.          state ← g(state, t_i)
6.          if out(state) != φ then
7.            begin
8.              print out(state)
9.            end
10.         end
11.   end
```

Algorithm 1: AC - Multipattern searching algorithm

2.2 Document Retrivel Problem

Basic Tools

1. Suffix Tree:

   Suppose suffix tree for string S is T(S) and is a compressed trie of all the suffixes of s. Each edge in suffix tree is labeled with a substring of S. For any node v in suffix tree T(S), suppose σ_v is the string obtained by concatenating the substrings labeling the edges on the path from the root node to v in the order they appear. Each leaf l represents one of suffix of string S and leaves of suffix tree has one-to-one relationship with the suffixes of S. So leaf l $\sigma_l = S[j...|S|]$ for a unique$j$. At each node in the trie, children are sorted based on the first symbol on the strings labeling the edges.

2. Suffix Array:

   The suffix array SA [1, n] of a string S [1, n] is a permutation of the string positions {1, . . . , n} such that the suffixes S [SA[i], n] of string S are listed in lexicographic order as value of i(index) increases. Every substring of S is a prefix of a suffix of S, and the suffixes prefixed by a pattern string P form a lexicographic index range in SA, and the starting positions of all the occurrences of a string P in S are found within an interval SA [lp, rp], which can be found by binary search on suffix array.

   For example suppose we have given a text T = happypuppy$ then Suffixes of T are: Here i: suffix means suffix which starts from index i in T.

   0 : happypuppy$ 1 : appypuppy$

   2 : ppypuppy$ 3 : pypuppy$

   4 : ypuppy$ 5 : puppy$

   6 : uppy$ 7 : ppy$

   8 : py$ 9 : y$

   10 : $

   Suffixes of T in lexicographical sorted order:

   1. $ 2. appypuppy$ 3. happypuppy$

   4. ppy$ 5. ppypuppy$ 6. puppy$

   7. py$ 8. pypuppy$ 9. uppy$

   10. y$ 11. ypuppy$

   Figure 2.2 describes the suffix tree and suffix array for above mentioned example.

3. Given text T and pattern P where $|T| = n$ and $|P| = m$.

   Query 1: Is pattern present in text ?

   This can be answered using Suffix Tree in time $O(m)$ and space $O(n^2)$, using KMP (Knuth-MorrisPratt string searching algorithm) in time $O(n)$ and space $O(m)$ and using Suffix Array in time $O(mlogn)$ and space $O(n)$.

   Query 2: Report all the indexes in T where pattern P is present ?

   This query can be answered by Suffix Tree in time $O(m+occ)$ where occ is number of occurrence of pattern P in Text T and space $O(n^2)$ with the help of longest common prefix information

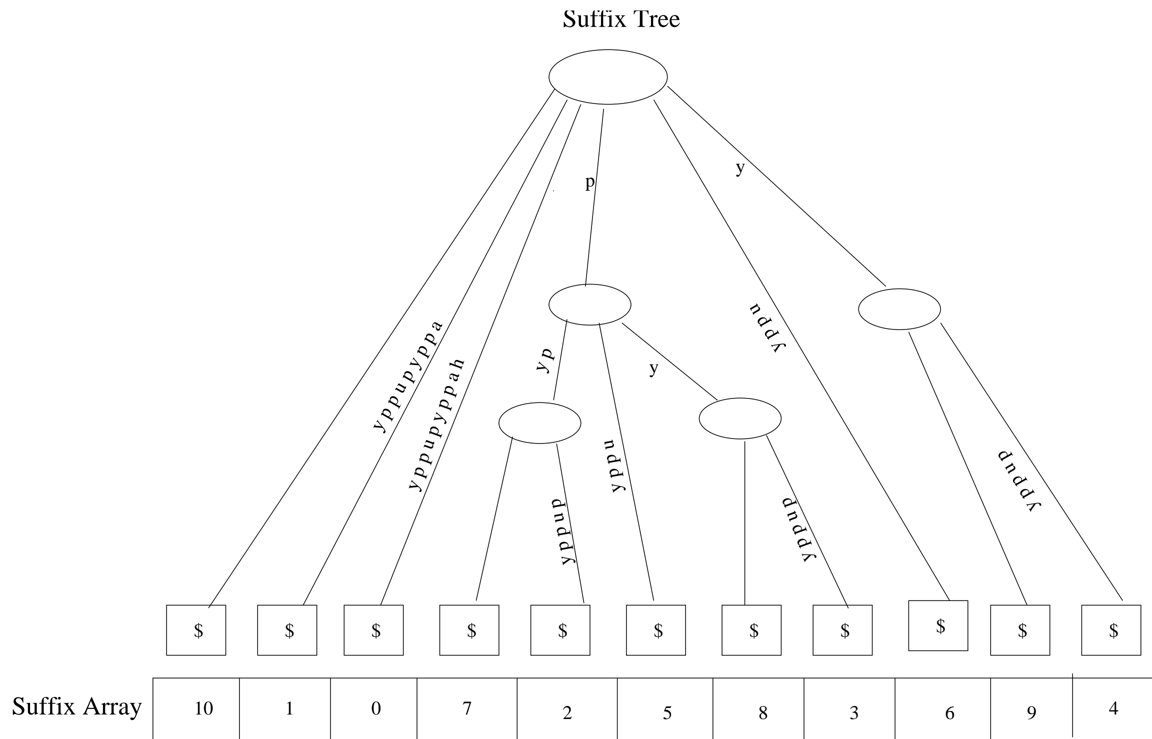


Figure 2.2: Suffix tree and Suffix Array

of suffixes, by suffix array in time $O(mlogn)$ and space $O(n)$ and by KMP in time $O(n)$ and space $O(m)$.

Optimal Algorithm for the Document Listing Problem: We are given a collection D of documents $d_1, ...d_k$ with $\sum_{i=0}^{k} |d_i| = n$ and $sum|d_i| <= l$. In the document listing problem, we are given a query comprising of a pattern string p of length m and our goal is to return the set of all documents that contains one or more copies of p. Muthurishnan gives the optimal version for document listing problem and current solutions for document listing problem are advanced version of this algorithm where space requirement are reduced for solution.

Now we will discuss the Muthurishnan's solution. First we define data structure primitives that this algorithm require and then define optimal solution.

Data Structure Primitives: Supoose document collection D has fixed size alphabet. The definition of a suffix tree for multiple string documents is called the generalized suffix tree. Suppose σ_v is the string obtained by concatenating the substrings labeling the edges on the path from the root node to v in the order they appear. In generalized suffix tree each leaf has dummy leaf children with respect to each document which has σ_v as a suffix. We use generalized suffix tree to index the

suffixes and search for patterns.

Locus μ_p for pattern p is the node in the generalized suffix tree such that σ_{μ_p} has the prefix p and $|\sigma_{\mu_p}|$ is the smallest of all such nodes which has p as prefix. The locus of p is not available is p is not present in any of the document. So if p has μ_p then p p occurs present in some documents. Generalized suffix tree of the library of documents can be built in $O(n)$ time and space as described by Weiner et al. [22]. So for any pattern p where $|p| = m$, locus can be determined in time $O(m)$. Suppose lowest common ancestor for any two nodes u and v in suffix tree T is represented by lca(u, v).

Muthukrishnan's Algorithm: Muthukrishnan' algorithm first preprocesses given document collection then this preprocessed form of document used to answer document listing queries. Muthurishnan's preproccesing has runtime $O(n)$ and after preprocessing document listing queries can be answered in time $O(m + occ)$ where occ is the number of documents where p is present.

Preprocessing is as follows:

Create generalized suffix tree for collection D of documents. Leaves of suffix tree represents n suffixes of the collection D. Label all these leaves of suffix tree $l_1, ..., l_n$ in the order they appear in the inorder traversal of the suffix tree. Suppose leaf labels i and j such that $i < j$ then σ_{l_i} would be lexicographically smaller than σ_{l_j}. Array D is an array of documents id's that correspond to leaves in order, and L is an array of the strating index of suffix for leaves in order. So if D[i] = j and L[i] = k then σ_{l_i} is the suffix $d_j(k...|d_j|)$.

Now algorithm for document listing is as follows :

Step 1: For p find locus μ_p.If μ_p is not present then p is not present in any document and stop.

Step 2: Find l_s and l_b using μ_p here l_s is leftmost descendant leaf of μ_p and l_s is rightmost descendant leaf of μ_p. So σ_{l_s} is lexicographically the smallest suffix that starts with σ_{μ_p} and σ_{l_b} is lexicographically the largest suffix that starts with σ_{μ_p}. All the leaves in range $l_s, ..., l_b$ has p as a prefix and these are the only leaves which has p as a prefix.

Now find unique document id in D[l_s,...l_b].

Suppose C is an array defined using array D. Values in C chains the occurrences of suffixes from a given document in the lexicographic order.So we set C[i] = j if j ¡ i, D[i] = D[j] = k, and j is the largest index with this property. If no such j exists, we set C[i] = - 1, a boundary value. Document i contains p if and only if there exists precisely one k in C[$l_s, ..., l_b$],such that D[k] = i and C[k] ¡ l_s.

Step 3: Given s and b, find all i [s, b] such that C[i] ¡ s and output D[i]. From the lemma above, it follows that the documents that contain p are all uniquely listed in the output. In order to find

all i [s, b] with C[i] ¡ s, we find j [s, b] such that C[j] is the smallest in [s, b] using the RMQ query with [s, b]. If C[j] ¿ s, then output is empty and we stop. Else, we output D[j] and solve the same problem on [s, j - 1] and [j +1, b] .This procedure clearly outputs each of the i [s, b] with C[i] ¡ s.

There are many solution for example sadakane et al. [16] and Valimaki et al. [17] etc. for document retrievel problem are availabel which are basically variant of Muthukrishnan's algorithm.

Chapter 3

Graphics Processing Units (GPUs) Architecture and CUDA Programming Model

Graphics Processing Units (GPUs) which were primarily designed for applications like medical imaging, computational fluid dynamics, 3D game rendering etc. Now GPUs advanced capabilities are being used broadly to accelerate computational workloads in areas such as cutting-edge scientific research, financial modeling. Intially GPU was hardwired to solve specific problems. Now GPUs have became programmable and there many programming interfaces for gpu are available. Using these programming interfaces gpu can be used to solve general purpose programming problems. Now GPU is a programmable graphics processor and a scalable parallel computing platform. There are several programming interfaces for gpu programming are available, for example CUDA and OpenCL. Now we are going to discuss about GPU architecture, CUDA programming model.

3.1 Graphics Processing Units (GPUs) Architecture

GPUs are specially designed hardware devices to cater the needs of highly parallel and compute intensive applications. CPU has some cores with good amount of cache memory and can handle a few threads at a time. GPU has hundreds of cores and can handle thousands of software threads parallely. Figure 3.1 compares the CPU, GPU architectures. In CPU, control unit has large number of transistors and arithmetic logic units (ALUs) has limited number of transistors. CPU is good for

execution of sequential codes because control unit has good number of transistors and provides different types of optimation like out of order instruction execution, branch prediction etc for sequetial execution of programs. GPU is well suited for compute intensive tasks because it can execute and handle large number of threads in parallel.

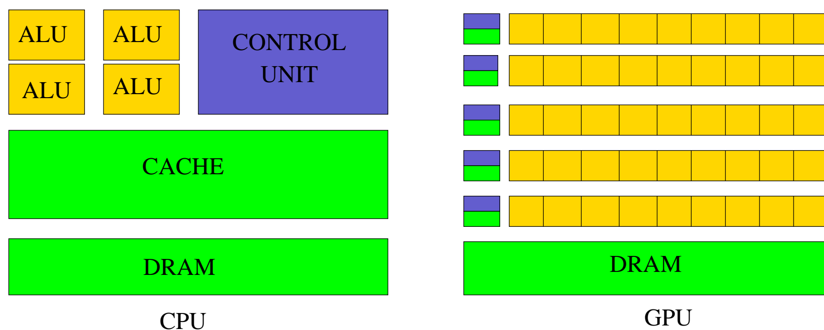


Figure 3.1: CPU and GPU architectures

Now lets understand how GPU runs large number of threads in parallel.There are different models of data parallel execution are available. A brief overview about data parallel execution models is given in Figure 3.2. GPU uses an architecture called SIMT (Single Instruction Multiple Threads). It means that group of threads called warp runs in parallel should run same instruction at any instance otherwise performance will drop drastically.

When Cuda is used as a programming interface for GPU then GPU is divides in four parts : Cuda Processor, Cuda core or streaming processor, Streaming multiprocessor, GPU device. Cuda processor represent a singal thread and streaming processor represents or runs warp of thread in parallel, streaming multiprocessor runs bunch of warps in parallel using streaming processors. GPU device is a collection of streaming multiprocessors (SM). Each SM has number of Streaming Processors or cuda core (SPs or simply GPU cores). Each SM has resposibility of creating, managing and executing threads in group (typically of size 32) called warps and these warps of threads run on Streaming Processors (SP). Brief overview is given in Figure 3.3.

3.2 CUDA Programming Model

Different type of programming interface are developed to enables programmer to utilize the massive parallel computing capability provided by the GPU for general purpose computing. Brief overview about these interfaces is given in Figure 3.4.

We are going to discuss about CUDA programming model in the following sections:

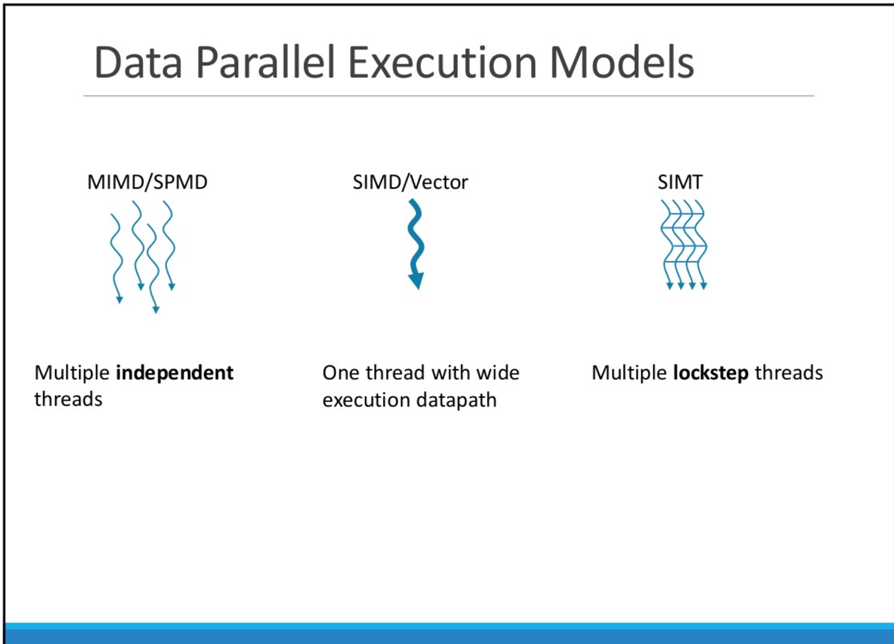


Figure 3.2: parallel data execution models

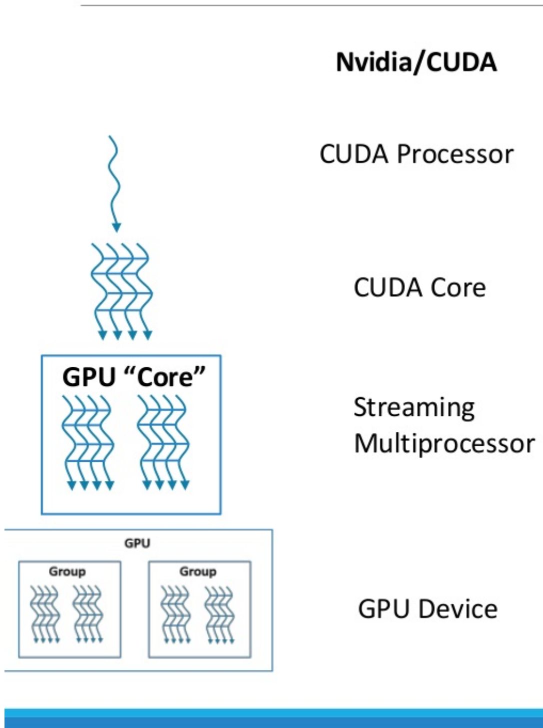


Figure 3.3: Cuda Architecture overview

GPU Programming Models

CUDA – **C**ompute **U**nified **D**evice **A**rchitecture

- Developed by Nvidia -- proprietary
- First serious GPGPU language/environment

OpenCL – **Open** **C**omputing **L**anguage

- From makers of OpenGL
- Wide industry support: AMD, Apple, Qualcomm, Nvidia (begrudgingly), etc.

C++ AMP – **C++** **A**ccelerated **M**assive **P**arallelism

- Microsoft
- Much higher abstraction that CUDA/OpenCL

OpenACC – **Open** **Acc**elerator

- Like OpenMP for GPUs (semi-auto-parallelize serial code)
- Much higher abstraction than CUDA/OpenCL

Figure 3.4: GPU Programming Models

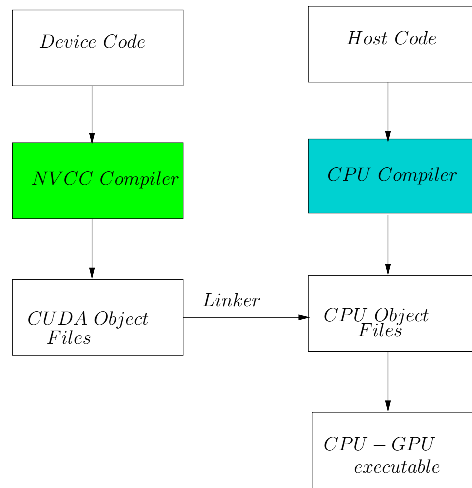


Figure 3.5: CUDA Compilation Model

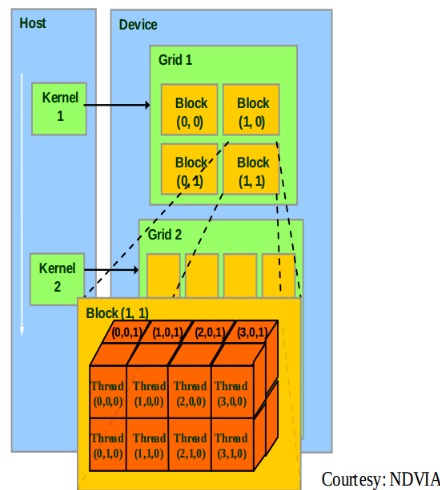


Figure 3.6: Thread Organization in CUDA

3.2.1 Compiler Model

CUDA programming has two part 1. A code which going to run on a CPU that is called host code and 2. A code which is going to run on GPU that is called device code. So every cuda program after compilation segregates the code into host and device code. Cuda uses compiler name nvcc for device code compilation. *nvcc* translates the device code into Parallel Thread Execution (PTX) code which is a pseudo-assembly code. The host code is compiled using a CPU compiler (C or C++). Brief overview about generation of CPU-GPU executable file and CUDA program compilation is given in Figure 3.5.

3.2.2 Execution Model

Thread management with respect to programer's view has four components:

- Kernel: kernel specifies what operation an individual thread is going to perform
- Block:Block represents group of threads. Block is a group of threads which is going to be execute on same SM
- Grid: Grid represents group of blocks. Grid can be specified in 2-3 dimention
- Threads : Thread performs the operation that is specified by kernel. Combination of thread id , block id can be used for 2-D or 3-D mapping

CUDA programming model has two level hierarchy for threads namely blocks and grids as shown in Figure 3.6 and described above. Each thread block runs independent of each other. Thread block can be scheduled on any SM. The block size is a multiple of thread warp size. CUDA launches large number of concurrent threads on GPU to solve problems parallelly. Each thread within a warp executes same set of instructions in same order but on different data. Thread responsibility is defined by users using kernel. CUDA maps thread block to a specific SM. More then one thread blocks can be mapped to same SM. Threads within a block can communicate using shared memory and threads across the blocks can communicate only using global memory.

3.2.3 CUDA Memory Architecture

Brief overview about these memories is given in Figure 3.7 and 3.8. GPU has six types of memories and each of them are useful for specific purposes. These six types are like this :

1. Registers: Register can be accesed by a thread
2. Shared Memory: Shared memory can be accesed by all the thread within a block but amount of this memory is very less 16KB-64KB .
3. Constant Memory: This is also a read only type meory and has a constant cache on GPU.
4. Texture Memory: is read only type memory and is very useful when coalescing is problem. Texture memory has cache memory which is optimized for 2D access pattern.
5. Global Memory: This is CPU main memory
6. Local Memory: Used for whatever doesnt fit into registers and is a part of global memory

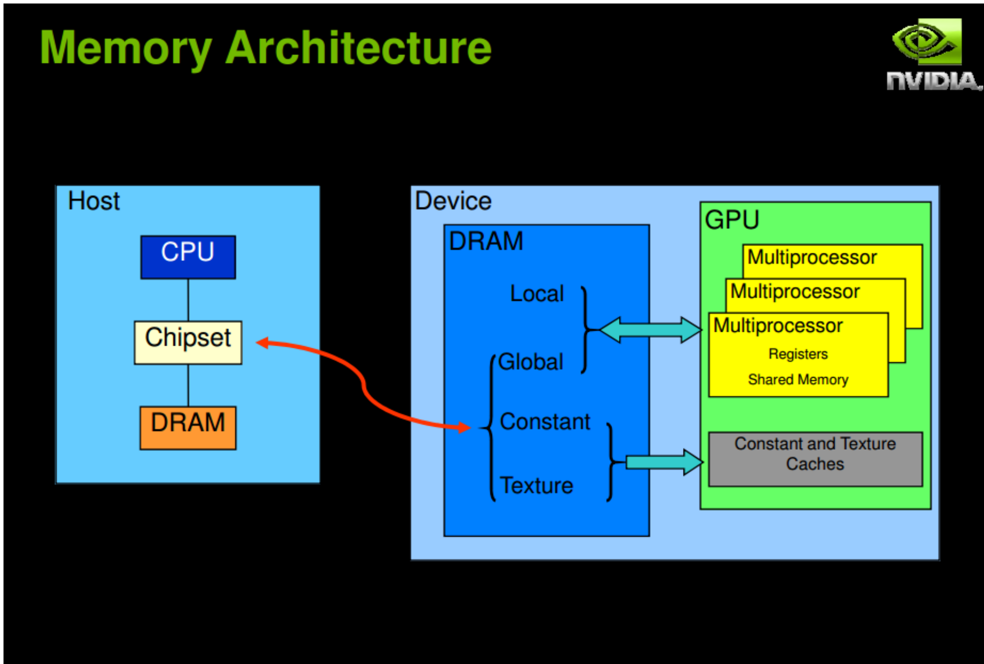
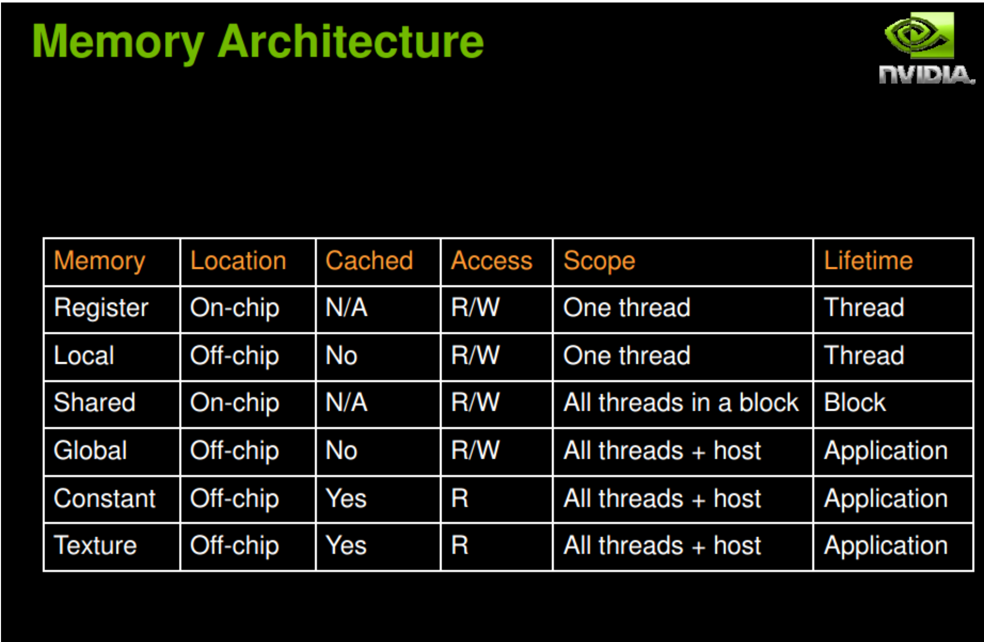


Figure 3.7: CUDA Memory Architecture



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Figure 3.8: Memory Type summary

Chapter 4

GPU to GPU implementation and optimization of Aho-Corasick algorithm

Aho-Corasick (AC)[1] has defined a multipattern search algorithm, this algorithm has a pattern matching machine/automata to perform multipattern search in linear time. Suppose, S is a string in which set of patterns need to be searched. The multipattern search begins with defining the matching machine/automaton start state as the current state and first character in the text string T is assigned as the current character. Matching automata makes a state transition by examining the current character of T. At each step of a matching procedure a transition to the state corresponding to the current character is made and the next character of string T becomes the current character. After performing a state transition, machine lists the matched patterns for the reached/next state as an output along with the position of the current character in the string T. AC multipattern matching algorithm has two versions - nondeterministic and deterministic version. Both versions use a finite state machine to represent the dictionary of input patterns. Deterministic finite automata (DFA) version of AC at each state has well-defined state transition function for every character in the alphabet and list of matched patterns. The number of state transitions made by the DFA while performing multipattern search in a string of length n is n. In the nondeterministic finite automata (NFA) AC version matching automaton states have two kinds of transitions success and failure. Success transitions are defined for automata states for characters that match a pattern character

and a failure transition is defined for the remaining characters. When the NFA version is used, The number of state transitions made while performing multipattern search are $2n$. The NFA version of matching automata uses less memory then DFA version of matching automata. In NFA version of matching automata states have few success transitions and can be compacted better than DFA states. AC has described how to compute the DFA and NFA for a set of patterns.

In next sections we will see how multipattern search algorithm given by [1] can be implemented using GPU.

4.1 GPU-to-GPU Implementation Strategy

Now we are going to discuss basic GPU implementation of AC algorithm. This implementation is defined by Zha-Sahni et. al. [13].

Character array *input* is input to the multipattern matching machine and *output* is an array of output states. Both arrays *input* and *output* reside in device memory. *output*[*i*] returns the state of the AC DFA following the processing of *input*[*i*]. Every state of the AC DFA contains a list of patterns that are matched when this state reached, so by using *output*[*i*] we can determine the matching patterns that end at input character $'i'$. These are some notations that we are going to use

n is number of characters in string to be searched

$maxL$ is the length of longest pattern

S_{block} is number of input charaters for which a thread block computes output

B is number of blocks which is equal to n/S_{block}

T is number of threads in a thread block

S_{thread} is number of input characters for which a thread computes output which is equal to S_{block}/T

TW is total work which means effective string length processed by GPU

Now we are going to discuss computational strategy given by Zha-Sahni et. al.

Partition the output array into blocks of size S_{block}. The blocks are numbered from 0 to n/S_{block}. The i_{th} output block comprises output values from $output[i * S_{block} : (i+1) * S_{block} - 1]$. To compute the i_{th} output block, it is sufficient to use AC on $input[i * S_{block} - maxL + 1 : (i + 1) * S_{block} - 1]$. For simplicity assume that there is special character that is not the first character of any pattern and set $input[-maxL + 1 : -1]$ equal to this special character. So overall a block processes a string of length $S_{block} + maxL - 1$ and produces S_{block} elements of the output and total number of blocks

B are going to be n/S_{block}.

Suppose that T number of thread are used for computation of an output block. Then, each thread is going to compute $S_{thread} = S_{block}/T$ of the output values to be computed by the block. So thread id t (thread indexes begin at 0) of block id b is going to compute

$output[b * S_{block} + t * S_{thread} : b * S_{block} + t * S_{thread} + S_{thread} - 1]$

and for computing these output values of thread t of block b we need to process the substring

$input[b * S_{block} + t * S_{thread}] - maxL + 1 : b * S_{block} + t * S_{thread} + S_{thread} - 1$

Algorithm 2 is psuedocode for a $T - thread$ computation of block i of output.

begin
// compute block b of the output array using T threads and AC
// following is the psuedocode for a single thread, thread t, $0 <= t < T$
t = thread index;
b = block index;
$state = 0$; //intial DFA state

$outputStartIndex = b * S_{block} + t * S_{thread}$;
$inputStartIndex = outputStartIndex - maxL + 1$;

//process $input[inputStartIndex : outputStartIndex - 1]$
for $(i = inputStartIndex; i < outputStartIndex + S_{thread}; i++)$
  state = nextState($state$, $input[i]$);

//compute output
for $(i = outputStartIndex; i < outputStartIndex + S_thread; i++)$
  $output[i] = state$ = nextState($state$,$input[i]$);
end

Algorithm 2: GPU to GPU pseudocode for AC algorithm

In Algorithm 2, *nextState* function used for finding the next state. Based on the precomputed matching machine types - NFA and DFA, nextState function can have two kind of definitons.

Now we are going to discuss the definion of *nextState* function for NFA and DFA version.

1. **DFA version *nextState* definiton:** In deterministic version-DFA each state of the matching automaton has a well-defined state transition for every character in the alphabet. so to find next state get next value for current character we can get value directly. Algorithm 3 describes *nextState* definition for DFA version.

// **Input:** state ID *state*and a current character *ch*
begin
  1. return nextstate[ch] for state ID *state*
end

Algorithm 3: nextState function definiton for DFA

2. **NFA version *nextState* definiton:** In the nondeterministic-NFA version matching automa-

ton states have two kinds of transitions success/goto and failure/fail. Success transitions are defined for automata states for characters that match a pattern character and a failure transition is defined for the remaining characters. Now we are going to discuss about goto and fail transitions of NFA AC. A automata for AC is a program which takes input a string text T and return all the patterns of P which are available in T as a substring. Automata is a set of states and each state has a state id which is a number. Node 0 act as a root node. pattern matching automata uses three functions: a goto function(g), a failure function(f), and an output function *out* process any node. Matching automata reades charater from T and makes state transitions from current state with respect to the read character. Three functions for a matching automata state are as follows :

L(q) is a function which returns string that is concatenation of characters of path from root node 0 to node q.

(a) g(q,a) is a goto function which gives the state entered from state q by matching char a and there are three cases

- if for state q edge (q, v) is labeled by a, then set g(q, a) = v.
- g(0, a) = 0 for each character 'a' that does not label an edge out of the root node 0 So the matching machine stays at the initial state while scanning non-matching characters.
- Otherwise g(q, a) = ϕ.

(b) f(q) for node id q is a failure function which gives the state entered at a mismatch.

- f(q) is the node, that is labeled by the longest proper suffix w of L(q) s.t. w is a prefix of some pattern p_i.
- a fail transition for a node does not miss any potential occurrences.
- f(q) is always defined for a node, since L(0) is a prefix of for every pattern.

(c) out(q) is a output function which gives the set of patterns recognized when machine is in state/node id q.

Algorithm 4 describes *nextState* definition for NFA version using values of goto, fail and output function for states.

DFA version definition of *nextState* can be used in Algorithm 2 but NFA version definition of *nextState* cann't be used in Algorithm 2 because this definition would generate thread divergence.

// This is a difinition of next State function for NFA : nextStateNFA
// *state* is current state of automata and i is a current character
begin
1. **while** g($state, i$) $= fail$ **do** $state \leftarrow$ f($state$)
2. $state \leftarrow$ g($state, i$)
end

Algorithm 4: Finding next for NFA version of AC algorithm

Thread divergence: Threads from a block are bundled into fixed-size warps for execution on a CUDA core, and threads within a warp must follow the same execution trajectory. All threads must execute the same instruction at the same time. In other words, threads cannot diverge. But if threads within a warp are not following the execution trajectory then this type of execution would create thread divergence within a warp. Thread divergence is not permitted because GPU uses single instruction multiple threads (SIMT) model for parallel execution of threads within a warp.

Now we are going to discuss see how thread divergence is happening because of $nextState$ definiton of NFA version AC algorithm (Algorithm 4). Thread divergence occurs because of line number 1 in Algorithm 4. Because of line number 1 threads within a warp may run loop different times and generate different instruction trajectory for threads.

In next section we will see how to remove thread divergence for NFA AC algorithm by redefining $nextState$.

4.2 Thread Divergence Free NFA AC algorithm for GPU

Lets see an example of matching automata for NFA AC algorithm. Suppose P is a collection of pattern strings.

Suppose P = {he, hers, his, she} then matching machine/automata is shown in Figure 4.1. These are the outputs for nodes

out(0) = ϕ, out(1) = ϕ, out(2) = ϕ,

out(3) = {he}, out(4) = ϕ, out(5) = ϕ

out(6) = ϕ, out(7) = {his}, out(8) = {he, she} , out(9) = {hers}

There are two parts in NFA AC automata in Figure 4.1. One part that contains dark edges is goto trie for NFA AC and other part that contains dotted edges is fail tree for NFA AC.

This a properties related to NFA AC automata which is useful to understand:

- Fail function property : As we discussed in previous section. For state q f(q) is the node labeled by the longest proper suffix w of L(q) s.t. w is a prefix of some pattern p_i. Fail function will

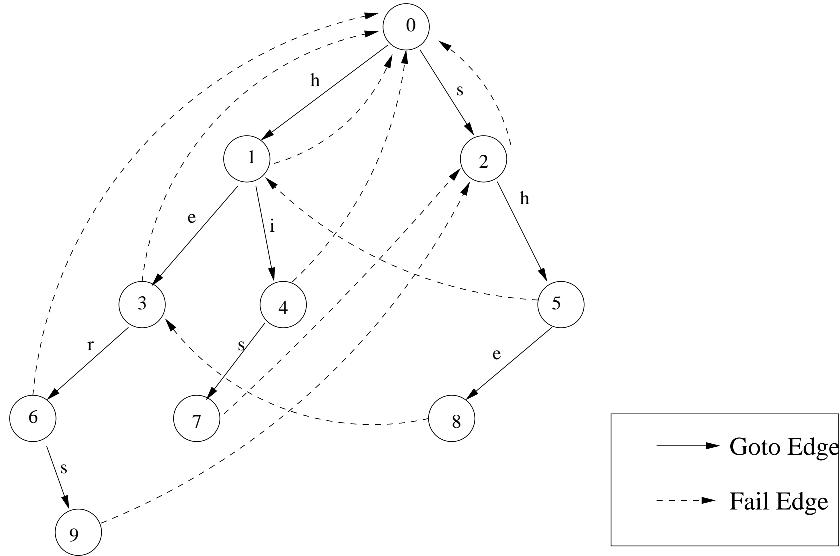


Figure 4.1: NFA AC matching automata

return f(q). So for any node q fail transition would at least takes one level up in goto tree.

Suppose *level* represents depth of goto tree of NFA automata. So for any node to find next goto state in Algotithm 4 would take at max *level* number of fail transition. So while loop in Algorithm 4 can be run for *level* number of time for each node to find next goto node.

NFA version AC algorithm has thread divergence because of *nextState* definiton of NFA version AC algorithm (Algorithm 4). Thread divergence occurs because of line number 1 in Algorithm 4. Because of line number 1 threads within a warp may run loop different times and generate different instruction trajectory for threads.

We can use above mentioned Fail function property to redefine *nextState* function such that thread divergence can be removed. Algorithm 5 defines exact definiton of *nextState* function such that it has no thread divergence. In Algorithm 5 we replaces while loop of Algorithm 4 by new while loop such that it will run for *level* number of times for each thread. This new definition of while loop generates same instruction trajectory for threads. So Algorithm 5 is thread divergence free because all threads are going to have same execution trajectory.

Algorithm 6 is thread divergence free NFA AC algorithm for GPU and nextStateNFA function in Algorithm 6 refers to Algorithm 5.

So overall in this section we developed a thread divergence free NFA AC algorithm for GPU.

In next section we will see how NFA AC version matching automata can be represented in less space then DFA AC automata.

begin
$level$ = max level or depth of NFA goto trie
$nodeID$ = int [level];
// $state$ is current state and ch is current character and $num = 0$ has level number of bits
// Now this is new definiton
1. i=0;
2. **while** i != level **do**
3.   nodeID[i] = $goto(state, ch)$;
4.   state = fail(state);
//num's ith will represent wheather ith value of nodeID is non zero or not
//bitmask num to represent nodeID's values
5. **for**(i=0;i¡level;i++)
6. $num = num|((nodeID[i]\&\&1) << i)$;
// now find first set bit in num and this will be a next goto node
7. $nonZeroBit$ = log(num & (-num))
8. return nodeID[$nonZeroBit$]
end

Algorithm 5: Thread divergence free version of nextStateNFA function

// compute block b of the output array using T threads and AC
// following is the psuedocode for a single thread, thread t,$0 <= t < T$
begin
t = thread index;
b = block index;
$state = 0$; //intial DFA state

$outputStartIndex = b * S_{block} + t * S_{thread}$;
$inputStartIndex = outputStartIndex - maxL + 1$;

//process input[inputStartIndex : outputStartIndex - 1]
for $(i = inputStartIndex; i < outputStartIndex + S_thread; i++)$
  state = nextStateNFA(state, input[i]);

//compute output
for $(i = outputStartIndex; i < outputStartIndex + S_thread; i++)$
  output[i] = state = nextStateNFA(state,input[i]);
end;

Algorithm 6: Thread divergence free GPU algorithm for NFA AC algorithm

4.3 Space requirement optimization for AC algorithm

When we use AC algorithm for multipattern searching we hace precomputed matching automata and this is important to reduce space requirement of automata in GPU implementation. Now we are going to discuss space requirement for DFA and NFA version of AC algorithm.

4.3.1 DFA AC space requirement

Suppose DFA AC automata has N states and alphabet size is 256. In DFA AC each automata node/state of the matching automaton has a well defined state transition for every character in the alphabet. So each node requires to store next node id corresponding to each alphabet ASCII value. There are N nodes so $log(N)$ bits would be required to represent a node id. So each node would need 256*$log(N)$ bits or $32 * log(N)$ Bytes. So total space required to store DFA would be $N * 32 * log(N)$ Bytes or N*256*$log(N)$ bits.

4.3.2 NFA AC space requirement

As we described previously that matching automata of NFA AC can be represented in two parts refer Figure 4.1:

1. Goto Trie: Represented by dark edges in Figure 4.1. Each node has next node id with respect to a character if there is a output edge for that character.

2. Fail Tree: Represented by dotted edges in Figure 4.1. Each node just has one output edge to fail node.

Space requirement for goto trie would be N*256*$log(N)$ bits because alphabet size is 256 and fail tree would be $N * log(N)$ bits. Now we are going to reduce space requirement of goto trie of NFA AC.

Goto trie part of matching automata of NFA AC is going to be look like trie/automata Figure 4.2. In Figure 4.2 output edges of a node are corresponding to some character. Nodes in matching auomata goto part of NFA AC would need to hold next node id with respect to a character if there is a output edge for that character from that node. This is how we can represent goto trie of matching automata:

1. Representation 1: Node has 256 size array $next[256]$ where $next[i]$ will hold next node id if there is output edge for character i here $'i'$ is ascii value for character.

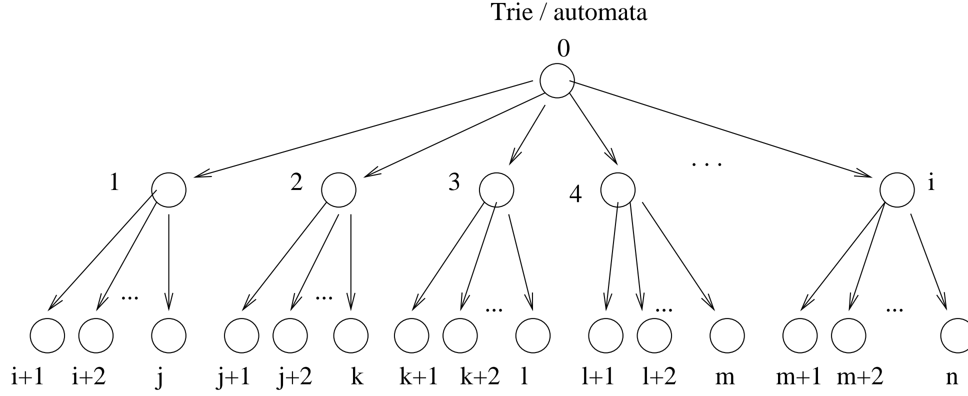


Figure 4.2: Trie/Automata

2. Representation 2: Suppose node id are generated as defined in Figure 4.2, level wise increasing and within a level node id increasing from left to right. Suppose each node stores two values: a number *num* of length 256 bits and left most child node id. Here ith bit value in *num* represents that wheather there is a output edge or next node for character $'i'$ is available or not. This how we can get next node id for character i for any node with the help of *num* and left most child node id: next node id would be sum of leftmost child node id for current node and number of set bits before ith bit in 256 bit number *num*. So overall

$$nnid = leftNN + count$$

Here *nnid* is next node id for character i and *leftNN* is node id for left most child node id and *count* is number of set bits before i_{th} bit in 256 bit number *num* stored in node.

Representation 1 would take space $N * 256 * log(N)$ bits and representation 2 would take space $N * (256 + log(N))$ bits. Representaion 2 has one problem that everytime next node id for character i is require to be found, it performs 256 sum operations and 256 shift operations in *num* to find number of set bits before i_{th} bit in 256 bit number *num* stored in node.

Solution for problem of representation 2 is like this: Use *offset* array of length 32 where each $offset[i]$ value is a 1 Byte number and $offset[i]$ stores number of 1's available in previous $8 * i$ bits of *num*. Now number of set bits before ith bit in *num* would be $numSetbits = offset[floor(i/8)] + count$. Here count is number of 1 bits in from $floor(i/8) * 8$ bit to i_th bit in 256 bit number *num*. Now next node id for character i would be

$$nextNodeId = leftmostchildID + numSetbits$$

So Node description for representation 3 would be like this :

1. Left Most Next Node id is a $log(N)$ bit number

2. 256 bit number

3. Offset array of length 32 where each value of $offset[i]$ is 1 byte number so total 256 bits required

So space requirement for this representation would be $N * (log(N) + 256 + 256)$ bits. Above two representations 2 and 3 of goto trie does not have much space requirement difference but later one is efficient for computing next node id.

4.4 Comparision of NFA and DFA AC algorithm For GPU

Zha and Sahni have given solution for DFA AC only. Now since we have given NFA AC solution with no thread divergence, we can use NFA AC for multipattern searching. Space requirement for DFA AC is $N * 256 * log(N)$ bits and for NFA using representation described in section 4.3.2 is $N * (log(N) + 256 + 64) + N * log(N)$ bits. Here N is total number of nodes in automata. NFA AC takes log(N) or 256 times less memory then DFA implementation, depends on value of N.This is how $log(N)$ or 256 times less memory can make differnce in GPU.

GPU has various types of memories and each of these memories have very significant speed difference. So automata storing location matters in GPU. In GPU like NVIDIA Tesla K20 we have 48KB of shared memory and 64KB constant memory. Zha and Sahni is storing there automata into texture memory, because when node id takes 1 byte then shared memory and constant memory can store at max 128 nodes which is very less number of nodes for most of multipattern search applications. If we use our NFA AC solution with node id of 2 byte (Number of nodes are 65,536) shared and constant memory can store at maximum 2K nodes. Automata with 2K nodes can be useful for many multipattern search applications. Storing in shared memory and constant memory important because shared memory is as fast as register when accessed within a warp of threads and constant memory is also very fast then texture memory.

Chapter 5

Conclusion and Future work

Zha-Sahni et. al. has given GPU implementation for DFA AC algorithm. Now we have provided a thread divergence free GPU implementation of NFA AC algorithm. Space requirement for DFA AC is $N*256*log(N)$ bits and for our NFA AC space requirement is $N*(log(N)+256)+N*log(N)$ bits Here N is total number of nodes in automata. So our compacted version of NFA AC automata takes $log(N)$ times less space than DFA AC automata. With 64 KB shared memory our NFA AC automata can store about 2K nodes, which can be useful for many applications. More optimized version NFA AC algorithm can be given and space requirement can be reduced for NFA AC automata.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM* 18, (1975) 333–340.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM* 20, (1977) 762–772.

[3] D. P. Scarpazza, O. Villa, and F. Petrini. Peak-performance DFA-based string matching on the Cell processor. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007 1–8.

[4] O. Villa, D. P. Scarpazza, and F. Petrini. Accelerating real-time string searching with multicore processors. *Computer* 42–50.

[5] X. Zha, D. P. Scarpazza, and S. Sahni. Highly compressed multi-pattern string matching on the Cell Broadband Engine. In Computers and Communications (ISCC), 2011 IEEE Symposium on. IEEE, 2011 257–264.

[6] N. Jacob and C. Brodley. Offloading IDS Computation to the GPU. In Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual. IEEE, 2006 371–380.

[7] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing* 6, (1977) 323–350.

[8] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai. A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. In Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on. IEEE, 2008 62–67.

[9] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. IEEE, 2009 175–184.

[10] S. Wu, U. Manber et al. A fast algorithm for multi-pattern searching .

[11] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu. Accelerating string matching using multi-threaded algorithm on gpu. In Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE. IEEE, 2010 1–5.

[12] A. Tumeo, S. Secchi, and O. Villa. Experiences with string matching on the fermi architecture. In Architecture of Computing Systems-ARCS 2011, 26–37. Springer, 2011.

[13] X. Zha and S. Sahni. Multipattern string matching on a GPU. In Computers and Communications (ISCC), 2011 IEEE Symposium on. IEEE, 2011 277–282.

[14] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)* 23, (1976) 262–272.

[15] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2002 657–666.

[16] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2002 225–232.

[17] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In Combinatorial Pattern Matching. Springer, 2007 205–215.

[18] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *Communications of the ACM* 22, (1979) 505–508.

[19] R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience* 10, (1980) 501–506.

[20] R. A. Baeza-Yates. Improved string searching. *Software: Practice and Experience* 19, (1989) 257–271.

[21] B. Commentz-Walter. A string matching algorithm fast on the average. Springer, 1979.

[22] P. Weiner. Linear pattern matching algorithms. In Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on. IEEE, 1973 1–11.