Parallel String Search Algorithm on GPU

Basar Kutukcu[1]

Abstract— **String searching is used in a wide area starting from trivial sentence search in personal computers to intrusion detection systems, search engines, plagiarism detection and more emerging research fields. String search is not a new thing in computer science and studied thoroughly in the past. Even though all the existing algorithms are optimized very well, they are bounded by the nature of CPU. In other words, all the existing algorithms are designed to work with one thread. Using parallelism in GPU, we can achieve a significant performance boost in string searching. In my parallel string search algorithm, every GPU thread are exactly mapped to one character in the whole text. All threads initially search for the first character of the key string. If a thread finds the first character of the key string, it checks if the found character is indeed the first character of the key string. While there are some limitations and drawbacks, the algorithm achieved approximately 18 times speed up compared to built-in 'findstr' command in Windows OS. The code of the algorithm can be found at: https://bitbucket.org/bkutukcu/parallelstringsearch**

I. INTRODUCTION

The main usage of string searching is to find particular strings in small files in personal computers. This type of usage directly serves its results to people. It is fast enough to be seen instant by people. Even though it would take twice as much time, people could not tell the difference in many cases. However, there are other usages of string searching and even though they are not common as much as the ones in personal computers, they are not less important. These other usages, such as intrusion detection systems, search engines and plagiarism detection, are open and required to improvements in performance. While the suggested algorithm in this paper does not directly address these problems, it could be adopted and improved to be used in such problems.

Ideally, a software solution is O(1). However, due to data of string searching, O(1) is not something that cannot be accomplished in CPU. There are, still, very efficient algorithms in CPU and they are fast enough for ordinary usage of string searching. On the other hand, there are emerging topics that require better performance than existing algorithms in string searching. This need can be filled using parallel algorithms in GPU.

II. PRIOR WORK

The prior work falls in two categories. The first category includes algorithms developed for CPU and the second category includes GPU algorithms. In the first (CPU) category, the fastest algorithm is used by 'findstr' command in windows (or GNU Grep). This command is quite fast for ordinary text searching. It is actually I/O bounded for small to moderate sized files. However, CPU's computing performance becomes improvable for large files. In the second (GPU) category, there are several people that implemented existing string searching algorithms on GPU and achieved significant speed-ups. Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis[1] showed that GPU implementations of 4 existing string searching algorithms (Naïve, Knuth-Morris-Pratt, The Boyer-Moore-Horspool and Quick Search) can be x24 faster than their serial implementations. Moreover, a group of students have implemented grep on CUDA and shared their source code while claiming x2-10 speed-up compared to original grep [2]. The drawback of these studies is that they try to implement existing serial algorithms on GPU. A new approach could be used to come up with simpler and better GPU specific algorithms to solve string searching problem.

III. ALGORITHM

This algorithm is tailored for GPU and therefore large number of threads. Unlike algorithms in CPU, this algorithm does not contain a major loop within kernel. In the algorithm, every single active thread is mapped exactly one character in the text. This might seem as an extreme limitation to the size of the searched text. It is theoretically true that this mapping limits the size. However, the limited size is so large for modern GPUs, the limit is practically too far to reach. To understand this limit in a clearer way, we can see the technical specifications [3] of the NVIDIA GPUs. After compute capability 3.0, the maximum x-dimensions of a grid of thread blocks is increased to 2^31 – 1. After compute capability 1.3, the maximum threads in a block is increased to 1024. Since text searching is a single dimensional problem, we could launch (2^31 -1) x 1024 threads in a single kernel call. To illustrate this number in a more understandable way, let's take a large known text file (a book). The famous book 'The Hobbit' by J.R.R. Tolkien is 513,717 characters long. To completely use the threads of the above-mentioned modern GPU, we need to search approximately 4,280,612 The Hobbit in a single kernel call. As it can be seen, this number practically is not reachable.

The block size is indeed chosen to be 1024 which is the maximum. Since kernel does not use any limiting factor such as shared memory or large amount of registers, choosing the block size as maximum would not generate any problem.

The algorithm is as follows. Every thread checks the character they are mapped to. If the checked character is not the first character of the key string, that thread's task is done for the current kernel call. If the checked character is

[1]Basar Kutukcu is Master's Student in Department of Electrical and Electronics Engineering, Middle East Technical University basar.kutukcu at metu.edu.tr

the first character of the key string, the thread starts checking the following characters to see if it is indeed mapped to the first character of the key string. If all the following characters are matched with the key string's characters, the thread saves the index of the first character.

Algorithm 1 Parallel String Search Kernel

```
1: procedure KERNEL(text, key, keySize, keyIndexes)
2:     if text[idx] = key[0] then
3:         save ← 1
4:         for i ← 1 to keySize do
5:             if text[idx + i] ≠ key[i] then
6:                 save ← 0
7:                 break                          ▷ from for loop
8:         if save = 1 then
9:             keyIndexes[++CurrInd] ← idx        ▷ AtomicAdd to increment CurrInd
```

The global memory is used for both the searched text and the key string. Saving the searched text in shared memory would not be convenient for two reasons. Firstly, an extra effort and performance cost would be required to arrange the data so that the local data in blocks would be overlapped by the size of key string. This arrangement would be required since otherwise the key strings placed at the boundaries of the blocks could not be detected. Secondly, saving the data in shared memory could not be effective since the majority of the data is only read once. The key string is also placed in global memory instead of registers. The reason for that is the benchmark results that are shared in the next section. However, a small modification, saving only the first letter of key string in registers, could achieved a small improvement compared to the case where all letters of key string are saved in global memory.

IV. EXPERIMENTS

A. Experiments to Compare Findstr and Parallel String Search Algorithm Variations

The experiments are done on the book 'The Hobbit'. Time is taken to be the only parameter to evaluate the algorithm. Time elapsed at transferring data to/from GPU is also taken into account since the comparison is done against 'findstr' and this command cannot be split into parts for benchmark purposes. First benchmark is done by searching the word 'Bilbo' in the book. 'Bilbo' appears 556 times in the book. Second benchmark is done by searching the word 'METU' which is inserted by editing the original text. 'METU' appears 10 times in the edited book. The same benchmarks are also done with a larger text which is 20 times copied version of the book. 3 different versions of parallel string search and 1 findstr are benchmarked with original book. 2 different versions of parallel string search and 1 findstr are benchmarked with edited (20x) book. The versions are as follows.

- Version 1: Findstr, built-in command of windows to search string in texts. The abbreviation, FS, is used in tables.
- Version 2: Parallel String Search with text and key string are in global memory. The abbreviation, GG, is used in tables.
- Version 3: Parallel String Search with text is in global memory and key string is in registers. The abbreviation, GR, is used in tables.
- Version 4: Parallel String Search with text is in global memory, key string's first letter is in register and rest of the letters are in global memory. The abbreviation, GM, is used in tables.

The results are obtained by running every case for 10 times and averaging the findings.

TABLE I
KEY IS 'METU', TEXT IS ORIGINAL BOOK

FS	GG	GR	GM
32,67ms	1,92ms	2,22ms	1,82ms

TABLE II
KEY IS 'BILBO', TEXT IS ORIGINAL BOOK

FS	GG	GR	GM
33,02ms	2,14ms	2,28ms	2,01ms

TABLE III
KEY IS 'METU', TEXT IS EDITED 20X BOOK

FS	GG	GM
38,16ms	13,84ms	13,53ms

TABLE IV
KEY IS 'BILBO', TEXT IS EDITED 20X BOOK

FS	GG	GM
34,33ms	17,09ms	16,67ms

It can be clearly seen that GM is the best implementation among parallel string search algorithms while every parallel implementation is better than FS. The other observations and comments are:

- While the best performance increase is approximately 18x with original sized book, the speedup gets low down to approximately 2x as the size of the text is increased by 20 times.
- While FS's performance does not change much by increasing size of text, parallel implementations' performances decrease drastically. The reason for that is memory copies between CPU and GPU.
- GM showed the best performance among parallel implementations because the first letter of the key string is guaranteed to be read by every thread. Therefore, putting it to a register increased the performance compared to GG. On the other hand, other letters of the key string are not guaranteed to be read and therefore

putting them in registers (as in GR) created an overhead that could not be compensated by occasional reading from registers.

B. Experiments to Compare The Time Elapsed on Kernel and Memory Copy Operations

Since the previous experiments proved that GM shows the best performance among all parallel string search algorithm variations, only GM is used to examine the elapsed time on the kernel and memory copy operations.

TABLE V
TIMINGS ON ORIGINAL BOOK, IMPLEMENTATION IS GM

x	METU	Bilbo
Kernel	0,33ms	0,49ms
Copy Opts	1,49ms	1,51ms
Total	1,82ms	2ms

TABLE VI
TIMINGS ON EDITED (20X) BOOK, IMPLEMENTATION IS GM

x	METU	Bilbo
Kernel	4,65ms	7,79ms
Copy Opts	8,88ms	8,87ms
Total	13,53ms	16,66ms

- What we can read from both of the tables above is that the kernel run for 'METU' takes less time than the one for 'Bilbo'. There are two reasons for that. Firstly, the 'METU' instances are significantly smaller than 'Bilbo' instances. Therefore, when 'METU' kernel runs, many lines, including a loop, memory write operation and an atomic operation, are not executed as much as in 'Bilbo' kernel. Second reason is 'false alarms' which are finding the first couple of letters of the key string but cannot find the rest. The kernel for 'METU' has less false alarms since finding 'ME' is more rare than finding 'Bi' in a book.
- When we increased the size of the text 20 times, both kernel executions and memory operations started to take more time to complete. It is straightforward to see why memory copy operations took more time than before since they are directly related with the size of the text. The time taken by kernel execution is also increased because when we increased the size of the text, we also increased the block number. Since the hardware of the GPU is limited and it is not guaranteed to execute all the blocks in parallel, some of the blocks are serialized and the elapsed time is increased.

C. Experiments to Compare Block Sizes

In this part, I examined the effect of block size on performance of kernel execution. I have used GM implementation, edited(20x) book and the key 'Bilbo'. When decreasing the block size, one important thing to consider is that the maximum size of the text that a kernel can search decreases. However, it is practically not important since the maximum amount is already impossible to reach as I have mentioned before.

TABLE VII
THE EFFECT OF BLOCK SIZE ON PERFORMANCE

Block Size	Kernel Time
1024	7,79ms
512	5,17ms
256	4,20ms
128	4,02ms
64	4,01ms

- From the table, we can see that decreasing block size increases the performance up to a point, then it starts to saturate. Using block size as 128-256 seem to be optimal in our case.

D. Experiments to Compare Release and Debug Builds

Up to this point, all the experiments are done on debug releases.In this section, the algorithm with GM version and block size of 256 is used to search the edited(20x) text to find 'Bilbo'. Two builds, debug and release, of the same case are compared. Only the kernel time is considered and memory copy operations are not measured.

TABLE VIII
PERFORMANCE COMPARISON BETWEEN DEBUG AND RELEASE BUILDS

Debug	Release
4,20ms	0,87ms

We see a huge performance increase when release build is used. While a performance increase is expected, having this much of performance increase proves that debug builds should not be used for benchmarking purposes when the platform is GPU using CUDA.

Experiments are done on NVIDIA GEFORCE GTX1050 and Intel i5-7300HQ.

V. DISCUSSION

The parallel string search algorithm has some limitations (or maybe flaws) as every other solution.

First, the length of the array, that holds the indexes of which found key string in the text, is unknown before running the kernel. If it is set too low, it may overflow with extra found key strings. If it is set too high, it decreases the performance significantly since memory copies have the largest portion of the elapsed time. This problem could be solved by using dynamic memory allocation in device and/or linked list data structure. However, these possible solutions are remained to be the subjects of future work since this paper focuses on searching algorithm.

Second problem is that first CUDA call takes huge time. It is so huge that it makes the performance improvements disappear. The reason for this huge time is that CUDA needs to execute some stand-up code and it executes them when

the first CUDA line is called. In a CUDA code, first call is generally cudaMalloc() and it takes an insignificant amount of time normally. However, when it is the first CUDA call, it can take more than a second to finish. It is not a big problem if parallel string search would be used for a while without stopping. However, it is a problem if the algorithm is to be used for one time to search strings in a text. In application level, this could be solved using a dummy CUDA call before starting everything and keeping CUDA alive even if there is no active searching.

It should be noted that while parallel string search algorithm shows better performance than CPU string search algorithm, it still cannot utilize the GPU completely. The very large part of elapsed time is from memory copy operations. Therefore, we could say that parallel string search algorithm is I/O bounded as well as CPU search algorithms but still GPU version is faster.

In overall, the parallel string search algorithm showed a significant performance increase over the fastest CPU string search algorithm. Even though it has some limitations and problems, it could lead to future research and optimization. It has a potential to be used on real world problems.

REFERENCES

[1] Kouzinopoulos, C. S., Margaritis, K. G. (2009). String Matching on a Multicore GPU Using CUDA. 2009 13th Panhellenic Conference on Informatics. doi:10.1109/pci.2009.47

[2] M.Burman,B.Kase, CUDA grep, (2012), GitHub repositary, https://github.com/bkase/CUDA-grep

[3] CUDA C Programming Guide. (n.d.). Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlfeatures-and-technical-specifications