cugrep: A grep implementation using CUDA

Tejaswi Tanikella
ttanikel@ucsd.edu

1 INTRODUCTION

For people working on UNIX platforms, using grep is an integral part of the experience. Among the many text processing command line utilities available on UNIX, grep might be the most commonly used and ubiquitous. The project described here is an attempt at building a tool which provides grep like search capabilities but uses CUDA to match lines on GPU cores.

The first motivation behind the project is grep's inherent parallelism. The input pattern can be used to compare each line without maintaining any data about the previous results. This inherent parallelism should theoretically allow us to search each line on an independent core in parallel, thus reducing the overall search time.

The second motivation behind the project is the promise of thousands of cores that modern GPUs provide. These cores have widely been used to crunch data in parallel. These processors can be used to run in-step and match the input pattern against each line.

The final motivation is to gain a better understanding of CUDA and gain experience using it. The project is the final part of PHY244, an opportunity to experiment and learn about parallel computing by getting my hands dirty.

The report explains the design of cugrep in the next section. It then explains NFA matching and matching that has been implemented in cugrep. Finally it describes the evaluation metrics, presents data comparing cugrep's results against grep and ripgrep and finally provides observations to explain the results.

2 DESIGN

2.1 Overall design

At a very high level, cugrep has five parts. They have been enumerated below and will be discussed one after the other in detail.

(1) Process input pattern
(2) read the input file and memcpy the file from host to device
(3) Start kernels to find line endings
(4) Start kernels to match each line against the input pattern
(5) Copy the matched results from the device back to the host and print the results.

The first part is to process the input pattern and convert it into a Non-Deterministic Finite Automaton (NFA). The details of processing the input pattern into a NFA have been explained in a later subsection.

The second part is to read the input file into host memory and then copy it to the device. A simple approach could be call the `read()` system call multiple times pulling the whole file into host memory. But this could reduce performance as the number of read calls increase with increasing file size. Instead cugrep uses the `mmap()` call to map the file into application memory. This has two benefits. Firstly, the OS will only allocate the space in the process's virtual memory and return a pointer but will not pull the whole file into memory until a memory access occurs. As we will see later, cugrep only tries to access parts of the file which has matches. Thus, only the parts of the file which has matches will be pulled by the OS into the application memory. The second benefit is that CUDA documentation advises developers to use pinned memory for better performance instead of using `malloc()` calls. Any memory allocated malloc() will be copied into pinned memory and then will pushed to the device. Thus, the use of mmap lets cugrep avoid issues with `malloc()` calls.

The third part is to identify the line endings. For this we have need to read each line and collect each point where a newline character is found. This part has been achieved by using two kernels and a reduction operation. This has been discussed in detail in the next subsection. After these kernels are done, a list of offsets are found, each of which point to the start of a line.

The fourth part is to match each line using the NFA. Pointers to the file, the offsets and the NFA are passed to the kernel. The kernel uses the offset to find the line it needs to process and then begins matching the line by running the NFA. In addition to the three input data pointers, the kernel is also passed a Boolean vector. If the kernel finds a match, it sets a value in the Boolean vector. This vector is used to print out the results.

The final part is to go through the Boolean vector and use the offset vector to print each matching line. Since this part is IO intensive, cugrep uses two techniques to reduce the overhead. First, instead of using `printf()`, we use `fputs()` which has does not have any formatting overhead and hence is faster than `printf()`. Second, `stdout` is line buffered by default, i.e. flushes the buffer on each newline character. cugrep makes changes so we use unbuffered `stdout`, flushing out data every 2KiB, which improves performance when a lot of lines need to be pushed out.

CUDA Kernel	Functionality
`find_line_num_offsets`	Find the number of line endings in the file. Based on the sum of these numbers, an array to store the offsets is declared.
`fill_offsets`	Populate the offset array with the line ending offsets
`match_lines`	Match each line against the input pattern

Figure 1: Each of the CUDA kernels and their functionality. All of these kernels are defined in `src/cugrep.cu`.

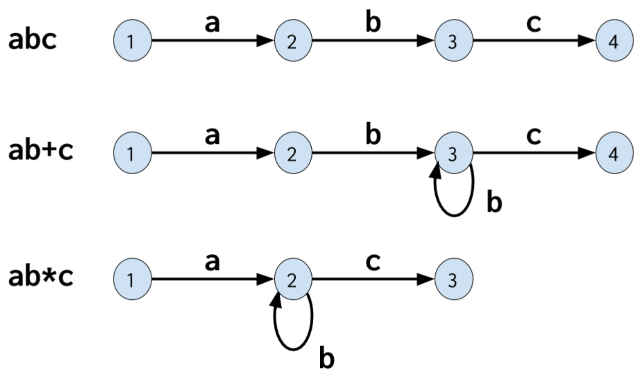

Figure 2: NFA built for regex patterns.

2.2 Input pattern processing and NFA

The input pattern contains characters and regex special operators. The current project does not implement the full set of regex operators but supports the following limited set of operators:

(1) `e*` : matches 0 or more of the preceding character e
(2) `e+` : matches 1 or more of the preceding character e
(3) `e|f` : matches expression e or expression f

Each input expression is translated into a Non-Deterministic Finite Automaton (NFA), is a state machine, such that at each state, any input value points to a single destination state. In our case, from each state an input character moves us to exactly one next state. A few sample NFAs have been provided in figure **??**. If along our process we reach the final state, we can declare that the line has matched with the input pattern.

Currently we do not support the `(`, `)` and `.` operators. These are very powerful and can allow real world and useful regex patterns. But these operators need recursive calls which CUDA kernels do not support. Their support can be added by hacking arrays in the kernel as stacks, but this has not been explored for two reasons. Firstly, this work is not exactly aligned with the objective of the project to gain experience with using CUDA. Secondly, the building and testing it may not be feasible within the time frame of a quarter.

2.3 Finding line endings

We now have an input file and an NFA to match each line, but we need find the start of each line, so we can start the a match kernel at each of these offsets. Even before we can start storing the offsets, we need a count of the total number of line endings so we can allocate space to store the offsets. Thus, we first find the total number of line endings. The `find_line_num_offsets()` kernel checks small equal sized sections of the file for line endings. Each thread counts the number found in it's section and updates it in an large array. We then run a reduction operation on the array to sum each element in the array. The result is the total number of offsets.

Next we start the `fill_offsets()` kernel, which is given a pointer to the offset array and a shared index value. The kernel searches in it's section for newline characters and stores line start offsets locally. Next, it increments the shared index value atomically with the number of offsets it found in it's part of the file. The `atomicAdd()` operation adds the count to the shared index value and returns it's old value. The kernel then copies the local values it collected into the offset array. The atomic operation lets the threads add values to the offset array without overwriting each other.

Once all the kernels are done, each line ending in the file is stored in the offset array. But, we since CUDA does not guarantee the order in which the threads run, the offsets are out of order. The thrust library, part of the CUDA libraries, was used to sort the array. Once all this done, we have the offsets at which each line begins, which can then be used to run the match kernel.

2.4 Match lines

The `match_lines()` kernel uses the NFA to match the line. The match kernel first copies the NFA into a shared memory since it will be accessed heavily. For each file input, it reads each character from the line to traverse along the state machine. If for any character a destination state cannot be found, we reset and move to the first state of the NFA. Along this process if we reach the final state, we can declare the line to have matched the input pattern and can stop processing.

3 EVALUATION

In each of the subsections below cugrep has been compared to two tools:

(1) `GNU grep v2.20` : This is the grep tool that is installed by default on the comet machines. The battle tested grep tool which we would like to compare cugrep against.

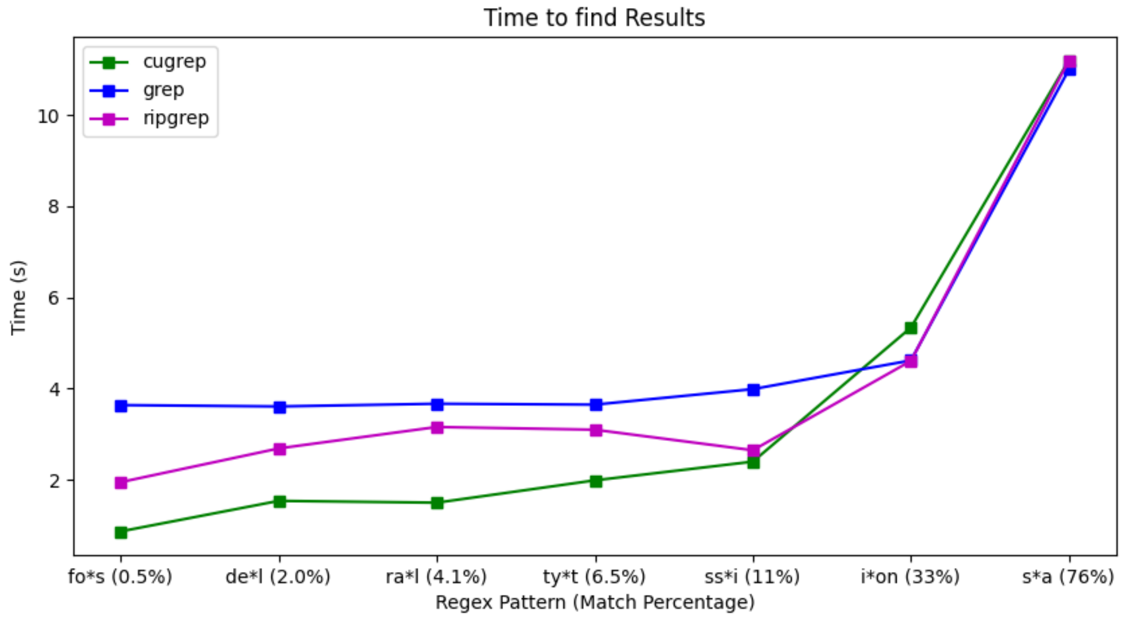

Figure 3: Time to find results by each of the three tools. The regex patterns are in increasing order of percentage lines that match the regex.

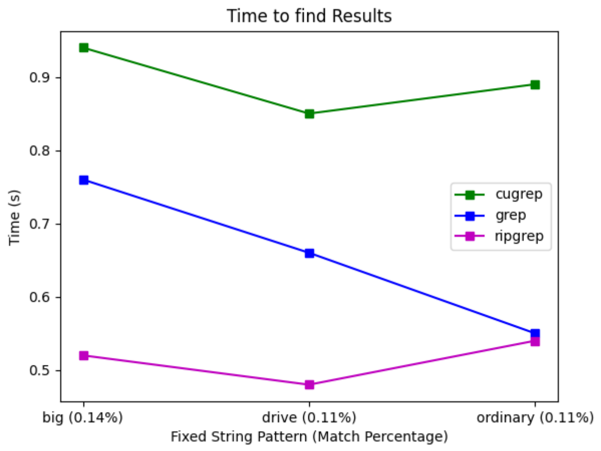

Figure 4: Regex Search: Time to find results of three fixed string patterns, with similar match percentages. The size of the fixed string pattern increases along the x axis.

(2) `ripgrep v11.0.2`: This is a modern take on grep, that uses the thread and memory safety guarantees provided by the rust language to process files using multiple threads in parallel. The tool also uses AVX vectorization instructions provided by Intel to vectorize processing.

The `time` unix tool is used to time each of the these tools. A sample sbatch file has been provided in `testing/cuda.sb`. The output of cugrep is compared with that of the other two to ensure correctness. Since for very small files the time difference between the tools is negligible, a large file with 19058252 lines, 933MiB in size has been used as the base file to run tests. The file was created by merging free ebooks available from Project Gutenberg. A smaller file (238MiB) has been provided among the artifacts `testing/file.7z`. A link to the complete file has been provide in the references.

3.1 Regex Search

The three tools were used to search with regex patterns. The results are shown in Figure 3.

We can make four observations, firstly cugrep performs really well when the pattern match percentage is less than 10%. This is promising, since this implies that cugrep can provide searches faster than grep and ripgrep in certain conditions.

The second observation is that as the match percentage is more than 10%, all the tools perform very similarly. As the match percentage increases, the IO to print the results to stdout increases, becoming the main bottleneck. Since all the tools print the same number of lines, all their runtimes converge.

The third observation is that for regex patterns with less than 10% matches cugrep's time increases monotonically. This might be becuase for low match percentages, a greater number of CUDA blocks work in step. When a thread finds a matched line, it stop searching. So as match percentage increases, divergence in a block increases, and the number of instep threads will decrease, decreasing the block's parallelism.

With a single thread, the time to match should theoretically be equal, since we match each line one after the other. This can be seen observed in grep's runtime, which stays around 3.6 seconds, for all sub 10% patterns, due to the aforementioned reason.

3.2 Fixed String Search

The three tools were used to search for fixed string patterns. A fixed string pattern is simply a regex pattern without any regex operators. Three fixed string patterns with sub 1% match percentage were used to ensure that IO is not a bottleneck, and the runtime refects the time it takes to match the pattern. The results are shown in Figure 4.

The plot shows cugrep performing very poorly compared to grep and ripgrep. Infact ripgrep produces the results in half the time for all three patterns. We also see grep's time reducing linearly as the pattern size increases.

The reason behind cugrep's poor performance is that while cugrep uses NFA based matching, grep and ripgrep switch the search algorithm to search fixed strings. Grep for example uses NFA matching for regex patterns, but uses the Boyer-Moore string matching algorithm to match fixed strings. The Boyer-Moore algorithm allows skipping parts of the string and this, in real world scenarios, causes it to run faster for longer input patterns, since longer sections can be skipped. This can be seen in the decreasing search time for grep as the fixed string length increases.

The same applies to ripgrep, but ripgrep in addition to Boyer-Moore, also employs Intel's vector instructions to search strings, thus letting it search faster than grep in certain cases.

Changing the search logic in cugrep was not persued with the limited time available, but should be a future direction of development.

4 CONCLUSION AND FUTURE WORK

cugrep is an effective substitute to grep in scenarios that requires searching simple Regex Patterns with low match percentages. In case of fixed strings, grep or ripgrep, are much more effective since they employ a different algorithm.

The current implementation has certain issues, which need to be pursued as future directions of development. Firstly the tool pushes the whole tool into the device memory. But if the file size is greater than the device memory, this operation would fail. This logic needs to be changed. Secondly, support for a additional regex operators needs to be added, which would increase the scenarios where the tool can be used. Finally, it needs to be explored if support for recursively searching lots of files can be added by using the async functionality provided by CUDA.

REFERENCES

[1] CUDA Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[2] Reduction Analysis http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf
[3] Implementing Regular Expressions https://swtch.com/~rsc/regexp/
[4] The complete 933Mib file: https://drive.google.com/file/d/16enNlN1MMblibZdfwxE6ahjLvQeE92cP/view?usp=sharing