View on GitHub

CUDA grep

Manish and Brandon's 15418 final project

Download this project as a .zip file Download this project as a tar.gz file

A Hardware Accelerated Regular Expression Matcher

Manish Burman

Brandon Kase

SUMMARY

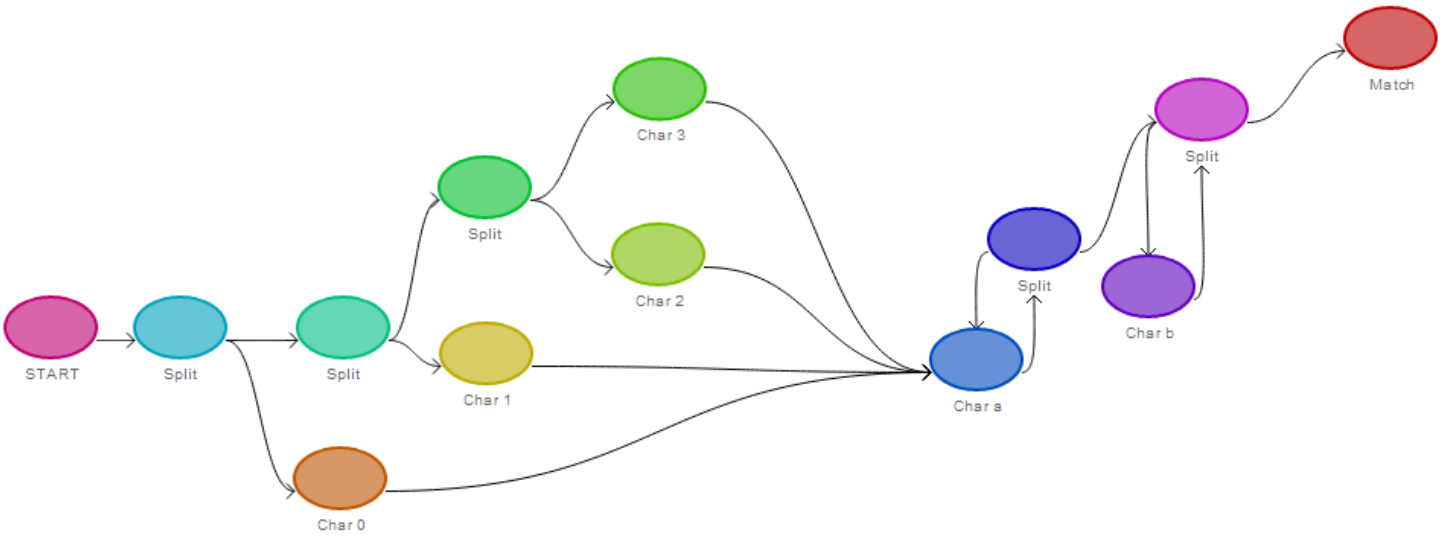
We successfully created a parallel regular expression matcher using CUDA for GPUs. Our implementation is anywhere from `2x-10x` faster than grep depending on the workload and about `68x` faster than the perl regex engine. We think that this makes it a viable candidate for use in the real world.

BACKGROUND

Regular expressions can be represented by finite automata - either NFAs or DFAs. We chose to use NFAs to implement our regular expression matcher simply because they use up a lot less memory than DFAs to represent the same regular expression. In the process, we created an NFA visualizer (shown below) to help us debug issues that we might come across.

We use the same representation of each NFA state as Russ Cox used in his starter code. Specifically each state contains an integer which represents the character being read -- the integer can also mean a Split or a Match. There is also two pointers back to the NFA state and some other fields that we use during the execution of our code. If the integer represents an ASCII character than we only look at the first pointer field of the state. If we did need to branch we put a Split state after the character (and perhaps more than one Split state). A Split state contains two non-null State pointers.

Here is a regular expression `[0-3]a+b*` visualized by our tool.



When we get a regular expression we initially parse it and take a complex regular expression (one with special syntax such as `[0-9]`) and simplify it (with at most `O(n^2)` in the length of the regular expression) to one using only `PLUS`, `ALTERNATE`, `CONCATENATE`, `STAR`, `QUESTION`, `ANY`, `PAREN_OPEN`, `PAREN_CLOSE` and from there convert it to postfix form. Then, we create an NFA (like shown above) from the postfix form. Once we have the NFA, we step through it based on the characters we need to match against a given string. If at any time a character we read does not have a transition in the NFA, checking stops. If checking is successful and we get to the match state, then we declare it to be a match.

The part that is computationally expensive and could benefit from parallelization is the large amount of pattern matching we do over a large body of text. We can break up the text into pieces and process these pieces in parallel. However, when running text through an NFA there is a dependency on the previous states -- you can not execute one NFA in parallel.

It is amenable to SIMD execution if each SIMD lane shares the same NFA. Since the work is the same at each state, different lines can run on different SIMD lanes with the same regular expression.

Basic high-level algorithm:

```
Load dataset and regular expressions
Compute NFA from regular expression
for all regular expressions
    for all lines in dataset
        use NFA to pattern match each line
```

APPROACH

Our matcher uses CUDA to run most of the computation on the GPU (no surprises there). We used C and tested on the GHC 5000 computers.

Initially our plan was to parallelize across the branches of the NFA. Once we had begun though, we realized that it was a much better idea to parallelize across the lines of the input file against which we had to match regexs. We opted to go this route because the time it took to match a single regex against a single line was basically negligible. Hence making matching faster by parallelizing across branches of the NFA would potentially end up wasting threads. Parallelizing across lines was a much better option.

We started off with Russ Cox's code initially and modified the sequential version to be able to match regexs against entire files. We also added a bunch of new features to it such as `ANY (.)`, ranges `[0-9][a-z]` and so on. We then worked on parallelizing the code. Initially we only parallelized the regular expression matching, but then realized that matching a single regex against every input file was actually slower than grep. This was because of the initial overhead of transferring the huge amount of data from the input file over to the GPU. As a result, we decided to change our workload to instead match multiple regular expressions against every input file transferred over to the GPU.

Also, we implemented and from here maintained a testbench to ensure that CUDA grep always adhered to the behavior of egrep.

Even still, we did as much as we could to minimize this overhead time. After trying to optimize the transfer of each line to the GPU, we instead combined all the lines into a big one-dimensional array and transferred it only once -- this led to a much lower initial transfer cost. We eventually realized that we could avoid even splitting the file input into lines in the first place and could read it once (marking newlines) into an array to transfer to the GPU.

We also changed the code and parallelized NFA creation for each regex. Each thread block effectively creates its own NFA from the regular expression. Then, each thread in that block uses the NFA to match a different line from the input file.

We chose to do the NFA creation on the GPU side since we realized that although it is faster to compute the NFA on the CPU, this time is negligible compared to the time we take using the NFA -- what wasn't negligible was the time it took to transfer the NFA over to the GPU since we essentially had to transfer each node of a graph one by one. If we computed it on the GPU we could avoid this transfer time -- and it did improve our results.

We found that mapping 5 warps per thread block gave us optimal speeds through latency hiding. Each warp contains threads that match on the same line due to the amenability to SIMD as described above. Each thread block has a specific regular expression (thus NFA) associated with it (there is no restriction on N thread blocks sharing the NFA since they will be working on separate lines no redundant work is done).

RESULTS

We used a 53MB javascript file as the data for CUDA grep to parse in our tests. Specifically, the lua interpreter compiled using emscripten to javascript and then concatenated with itself until the size grew to 53MB. Our regular expression file was 2000 lines of the same regular expression. We noticed that time doesn't actually vary much based on the regular expression being used as long as it's finding a needle in a haystack. For example, searching for a specific function name in a large code base or a macro definition. Searching for hay in the haystack is not a common use case for a regular expression matcher.

Sequential Matching

- For a `53 MB` file GNU grep takes anywhere from `0.025s to 0.113s`
- Our sequential matcher consistently takes about `0.29s` per regex
- Our sequential matcher is anywhere from `~3x to ~11x` slower than GNU grep

Note: we tried running multiple instances of grep concurrently (by daemon-izing), but observed that grep did not perform better than the average `0.025s to 0.113s`. We think that this might be because regular expression matching on the CPU is bandwidth bound as well.

Parallel Matching

- Our parallel matcher takes about `0.0122s` per regular expression
- That's anywhere from a `2x - 10x` speedup over grep!
- This is without taking into account the overhead of transferring data to the GPU.
- Here are the numbers related to the overhead, this is for a 53MB JavaScript file matching 2000 regular expressions

```
Parallel CopyStringsToDevice 0.0934 seconds
Parallel pMatch 10.1795 seconds
```

Note: that copying the strings to the device is extremely fast now (this also might be cache exploitation on the GPU)

Speedup Over PERL

For a 53 MB file

- Perl takes about `0.825s` per regular expression
- Our matcher takes about `0.012s` per regular expressions
- That's a `68.75x` speedup over sequential PERL matching

Above is a graph which records our speedup over the single-threaded grep which runs on a single core of a CPU (to be fair we did try to parallelize by spawning processes for each regular expression but this resulted in a slow-down for grep as explained above) for a 53 MB javascript file as we increase the number of regular expressions used per run time. We use wall clock time as our guide. As you can see it starts to level off at around `9x` when we begin matching a large number of regular expressions.

Speedup seemed to be bounded by data transfer. Specifically the bandwidth between the global memory when we transferred part of a line to

Our problem of regular expression matching has horrible locality since we really only look at a line once per regular expression. Perhaps if we had more time, we would attempt to test many NFAs per line in a thread block so that we have more temporal locality.

To test this theory out, we used a temporary kernel and just accessed memory and did NO regular expression matching and we got almost exactly the same time. This leads us to believe that our speedup is limited by the `170GB/s` bandwidth between global memory and the individual cores. This is pretty surprising because we thought we would only be experiencing this problem from transferring the data to the GPU not on the GPU.

We think that using GPUs vs CPUs was a good idea purely from an application standpoint. Regular expression matching is used in applications such as spam filters and anti virus software. This can slow a computer down. Not only is speeding up the computation useful to receive clean emails faster and for those who want a safe computer, but moving all this computation over to the GPU is likely to give the user a better experience since almost no computation is done via the CPU.

REFERENCES

http://swtch.com/~rsc/regexp/regexp1.html as a reference. We used http://swtch.com/~rsc/regexp/nfa.c.txt as starter code.

LIST OF WORK BY EACH STUDENT

Brandon

- Fixed issues with Russ Cox's initial code
- NFA Visualizer for debugging.
- Regular expression builder - added a lot of new features that weren't supported (for a more in depth description see above)
- Talked with Manish about parallelization strategies
- Implemented the system to transfer the entire body of text over with only one cudaMemcpy -- before we had been transferring the text line by line (we ended up removing this after Manish implemented the file reader that partitioned the text into lines while reading)
- Created Demo for the final presentation

Manish

- Ported over Postfix & NFA creation over to the GPU
- Ported over code for matching regular expressions over to the GPU
- Worked on Parallelizing code
- Implemented a custom stack to allow for divergence in recursion - The GPUs don't seem to be able to handle divergence in recursion - as in two recursive calls at every step - sort of like a tree. One recursive call works fine.
- Implemented an efficient initial file reader - drastically improved file read time
- Created a testbench to ensure we always remained compatible with grep
- Tried multiple approaches to get better speedup - e.g. have nfa creation for all regular expressions happen first and then match regular expressions against the nfas. Turned out to not help as the speedup was exactly the same - leads us to believe that its limited by the GPU global memory bandwidth.

Discuss on hackernews

CUDA grep maintained by bkase and mburman