Pattern Matching Benchmarks on CPU, GPU, and FPGA

Hanuman Chu
Colorado School of Mines
Golden, CO, USA

Joey Vongphasouk
Colorado School of Mines
Golden, CO, USA

1 INTRODUCTION

Pattern matching is a widely used computational task seen in many different domains such as network security, bio-informatics, and machine learning [1]. As these domains continue to grow in scale, the demand for faster and more scalable pattern matching also grows. A potential way of speeding up pattern matching is through parallelizing its execution onto hardware accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs).

This project benchmarks the performance of parallel pattern matching implementations on GPU and FPGA and compares against traditional CPU-based solutions.

2 TARGETED DOMAIN [2 PTS]

Pattern matching is the process of checking whether a specific sequence of characters exists in some given input such as large text files or continuous data streams. These patterns are typically implemented using regular expressions, which define search patterns and behaviors through basic characters and regex operators such as concatenation, alternation, and the Kleene operator. Recognizing these patterns has applications in areas like network security where incoming data streams are checked for malware. Our project targets the domain of automata-based pattern matching by benchmarking various implementations across different hardware platforms.

3 MOTIVATION: THE NEED FOR ACCELERATION [2 PTS]

On a large scale, pattern matching is slow with even files as small as a gigabyte taking several seconds to match a small ruleset. This means any speedup is appreciated even constant ones. Even better, patterns can be converted to NFAs which can be parallelized making them take less steps to process and while NFAs can be converted to DFAs thus taking the same amount of steps to process serially, the conversion is exponential in the worst case in terms of time and space.

4 RELATED WORK [8 PTS]

A paper published in May 2020 defines Grapefruit, a design for processing homogeneous NFAs on FPGAs [9]. Unlike regular NFAs, homogeneous NFA transitions don't have a character associated with them. Instead, it has state transition elements or STES which have an associated character set and transitions can only be used if the character being read matches the state they're going to.

As an example, we can look on the left of Figure 1. If we were in state 0 and read AA, we would stay in state 0, while reading CC would send us to both state 1 and state 2 because the asterisk in the character description for state 2 is a wildcard.

To convert a homogeneous automaton into a circuit, the algorithm first creates a look up table for each state. Each of these look up tables is fed a character each cycle and outputs a signal if the character matches the character set of the corresponding state in the automaton. These signals go to an AND gate which allows the signal through if the current state is active. For the start state, they use an always on wire for this purpose because the start state is always active. This is done because we are not matching a single string but rather whether any substring in an input file matches our regular expression. To determine whether non starts states are active, we have flip flops which store whether a parent of the state outputted an on signal last cycle. As an example, we can see that the flip flop for state 1 has a wire coming from an OR gate which has wires coming from both state 0 and state 1 because state 1 in the automaton has incoming edges from state 0 and state 1.

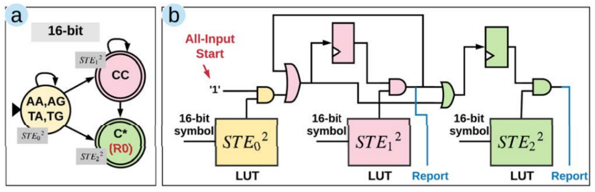


Figure 1: A 16-bit homogeneous automaton and the corresponding circuit diagram [9].

After wiring up the LUTs and FFs for each state, we add wires from the output signal of any states that correspond to an accepting state in the automaton and combine them into an output signal for our automaton representing whether the automaton had a match that cycle.

In addition to Grapefruit, we found several other papers dealing with accelerating the processing of regular expressions. One which released in March 2020 covers 3 optimizations to GPU based NFA processors [1]. Another which released in December 2007 details another design for an FPGA based NFA processing which has been obsoleted by Grapefruit [2]. Finally, we found a paper released in 2019 which covers Hyperscan, a CPU based NFA processor [3].

5 PROJECT DESCRIPTION [2 PTS]

5.1 CPU

5.1.1 Targeted platform. The project uses an 11th Gen Intel i9-11900K CPU for its baseline hardware implementation. This CPU has 8 cores and has a processor base frequency of 3.5 GHz speed.

5.1.2 Implementation Details. To test pattern matching on the CPU, we use the grep command implementation by GNU. We specifically test using the egrep command variant, a version of grep that has extended capabilities such as the + operator.

GNU grep utilizes the Boyer Moore algorithm for pattern matching [4, 5], which is an optimized search algorithm that looks to pre-process the search pattern and skip iterations of the text based on a bad character and good suffix heuristic.

The bad character heuristic works by keeping a table of numbers per symbol in the pattern which is associated with the amount of characters that can be skipped if a character were to mismatch. By matching from the end of the pattern, iterations of a match can be skipped if a mismatch were to occur. Refer to Figure 2 for an example where a mismatch occurs on the fourth character of the pattern. In this example, the search can skip one iteration as the character that mismatched was an “A", which has its next appearance 2 iterations away.

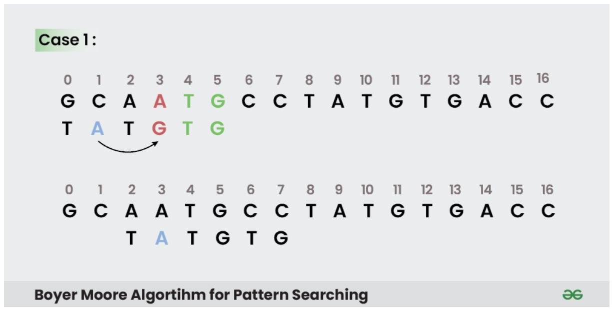
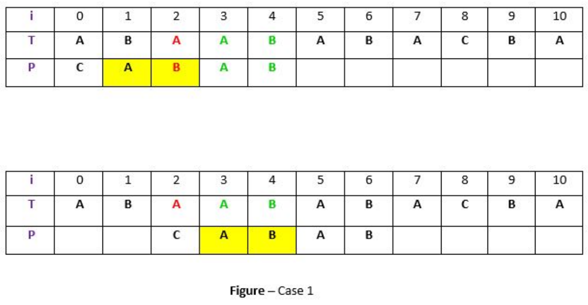

Case 1:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
G C A A T G C C T A T G T G A C C
T A T G T G
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
G C A A T G C C T A T G T G A C C
T A T G T G
Boyer Moore Algortihm for Pattern Searching

Figure 2: Boyer Algorithm for bad character heuristic [5].

Similarly, the good suffix heuristic keeps track of matching substrings within the pattern to skip iterations. Refer to Figure 3 for an example, where the “AB" suffix can match again 2 iterations ahead, thus 2 iterations can be skipped.



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

Figure 3: Boyer Algorithm for good suffix heuristic [6].

5.1.3 Challenges. The project ran into a few issues regarding benchmarking the CPU. For example, in order to get information on the CPU, we used the turbostat command-line utility discussed further in the report. This utility, however, reads system-level registers that require root access to the system which we did not have. We bypassed this issue by getting sudo access for turbostat from Justin Davis, a system administrator on the server with all computing devices we tested with.

5.2 GPU

5.2.1 Targeted platform. To test high-throughput pattern matching on a GPU, we used an NVIDIA RTX 3080Ti which has 10,240 cores, a boost clock operating at 1.67 GHz, and 12 GB of memory.

5.2.2 Implementation Details. We extend off the work done by Manish Burman and Brandon Kase [7]. Their project looked to implement a grep that used the CUDA API to parallelize pattern matching. At a high level, their project looked to load the dataset and regular expressions, loop through each regular expression and compute the NFA, then parallelize across each line in the searched input file and pattern match using the computed NFA. The pattern matching utilizes a naive string searching algorithm different from the one discussed for GNU’s grep implementation. This search looks to step and attempt a match on each character in the string. Refer to Figure 4 for an example NFA output produced by the project using the regular expression pattern “[0-3]a+b*".

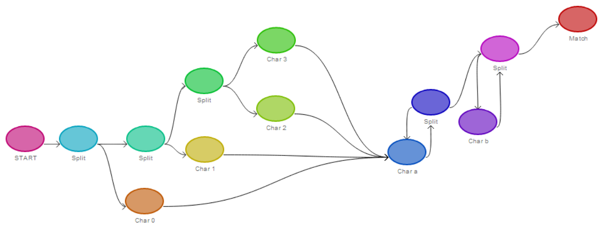


Figure 4: NFA produced for the regex pattern “[0-3]a+b*".

The project accelerates pattern matching by parallelizing on the searched input file, splitting by the newline character. We iterated on their project through the following changes:

- Changed whole line matching to now match substrings within lines.
- Changed iterative rule matching on the regex file to use a combined NFA of all the regexes in the ruleset rather than using individual NFAs for each rule.
- Optimized the kernel call by specifying the grid and block size specific to the NVIDIA RTX 3080Ti.
- Moved preprocessing of regex to NFA from the device (GPU) onto the host (CPU).

5.2.3 Challenges. In order to ensure a fair comparison between each device, we had to make changes to the GPU code so that similar inputs can be used and similar outputs are obtained through each method. The previous GPU pattern matching implementation had limited capabilities in comparison to the other pattern matching implementations on other devices. To solve this, we modified the existing code to perform similarly to the other devices in terms of expected input and output. Major issues include matching on whole lines and iterative rule matching on the ruleset. These changes along with other optimizations to the code were mentioned previously in the report.

We additionally looked to test other current GPU implementations of pattern matching as well as other personal GPU devices/ However multiple dependencies with CUDA versions and permission issues arose. Lastly, we encountered issues with using input files of 2 GB or more. This is due to how the current implementation loads all memory it needs at once and then performs its pattern match, thus being limited to the current memory of the device. Since the input file is already split up by newlines, it wouldn't affect correctness for us to split the file up and send it but we did not have enough time to implement this.

5.3 FPGA

5.3.1 Targeted platform. We are using an Xilinx Alveo U50 as our FPGA which has 872K LUTs, 1,743K registers, 28 MB of SRAM, and a PCIE bus which can be configured to act as 16 Gen 3 links which can do 8 GT/s or as 8 Gen 4 links which can do 16 GT/s. We are communicating with it on the CPU mentioned earlier via those buses.

5.3.2 Implementation Details. To make the FPGA process a regex, we first need to make a memory configuration file specific to the regex. To do this we follow the process in Figure 5.

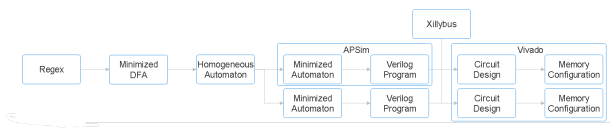


Figure 5: Pipeline for converting a regex into an FPGA memory configuration file

We start with a regex, turn it into a DFA, and minimize it. Then we convert it into a homogeneous automaton by taking every edge and converting it into an STE. As an example, we have in Figure 6 the DFA for the regex R(E|O)*D and the corresponding homogeneous automaton. Each node in the automaton is created from a node in the DFA and an incoming edge. If the node in the DFA is accepting, the node in the automaton is also accepting which can be seen in the node in the automaton corresponding to node 2 and character D in the DFA. After adding nodes, we add edges to the automaton based on the edge the node was created from. For example, the node 1:E was created from the E edge going to node 1 in the DFA whose source is node 1 so we add edges to 1:E from all nodes in our automaton that correspond to node 1 in the DFA: 1:R, 1:E, and 1:O. If the edge's source is from a start node, we instead mark the node in the automaton as a start which we denote in our diagram with an arrow coming from no node.

After we've finished creating our automaton, we need to minimize it and create a Verilog program representing it. We have 2 implementations for this: our own and one in the APSim repository which is listed in the resources section. To use the APSim implementation we convert the automaton to an anml file, a patented data format [10], and pass the result to APSim which outputs several Verilog files.

In our implementation, we minimize the automaton by doing 3 types of merges shown in Figure 7. The first is combining the character sets of nodes with the same parents and children. The second is combining nodes with the same parents and character sets because whenever one node is entered, the other node is necessarily entered as well. This shouldn't ever occur because we start with a DFA but we implemented it in case we figure out a minimization

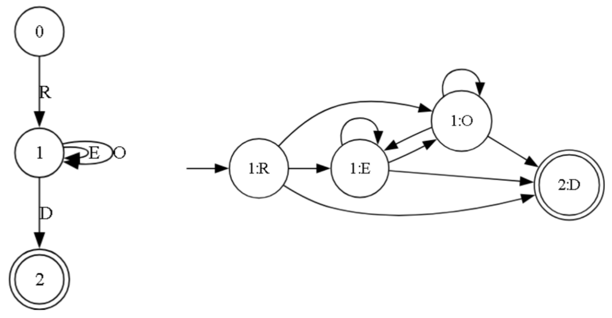


Figure 6: DFA (left) and homogeneous automaton (right) for the regex "R(E|O)*D".

that can create this situation as a byproduct. The third merge we do is combining nodes with the same children and character sets because it doesn't matter whether we are in one node or the other. In addition to merging nodes, we do other small optimizations like removing dead states and useless edges.

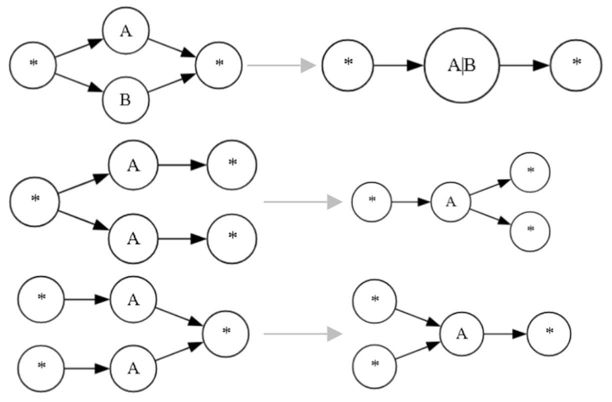


Figure 7: Three types of merges. From top to bottom these are: merging nodes with the same parents and children, merging nodes with the same parents and character sets, and merging nodes with the same character sets and children.

Once we've finished minimization, we turn our automaton into a Verilog program using the Grapefruit algorithm [9]. Like APSim, our implementation of the algorithm outputs a Verilog module which we attach to Xillybus, an IP Core that is free for research which allows us to communicate with the FPGA via file I/O. After putting them together, we use Vivado to design a circuit, optimize it, and create a memory configuration for the FPGA.

5.3.3 Challenges. The primary challenge we faced was being able to access things we needed to. Primarily for FPGAs, there aren't a lot of tools available to help us circumvent access permissions. Even when there were such tools, we had to ask they be installed and we be given sudo access to them. A workaround we had was sudo docker access which we can use to access any devices/files we needed by mounting them into a container. However, even with all the access we need, the device is still a machine other people are using. Out of consideration for them, we chose not to restart the Pikespeak device. As a result, we have been unable to fully test our program due to it requiring a system reboot to properly run the flashed FPGA configuration.

Another problem was with using the Vivado over SSH. The issue was with the Vivado UI which had a lag of anywhere from 2-20 seconds whenever we made any action such as a click or a scroll. We didn't count this as an access issue because Vivado was quite responsive in the command line, however this did significantly impact our workflow. The problem here was that we were unfamiliar with many aspects of FPGA development. The Vivado UI, which has things like block building and viewing circuits, would have been very helpful in building our program but because of this issue, our current product was built almost entirely via the command-line interface.

The third issue was with a lack of resources for development on the Alveo U50 specifically. We had a lot of trouble finding IP Cores that were able to do what we wanted. Additionally, the AMD downloads site was down for quite some time so we were unable to even see what the board and constraints looked like. Even Xillybus, which supports a plethora of FPGA devices, doesn't support the Alveo U50 board. As a result, we had to spend a ton of time modifying an IP Core for Virtex Ultrascale+ devices to get it to compile.

6 EXPERIMENT SETUP [2 PTS]

6.1 Methodology

We used the Pikespeak device for each of the experiments conducted. The input testbench for files to perform patterns on consisted of multiple free text files found online. Additionally, a script was used to shorten or extend a given text file to a specific length, where extending search files was done by appending multiple copies of that file onto each other. We created the regex rulesets for each file manually and aimed to have different rules to test each component for correctness and variance. To test for correctness, we created a script to check for similar outputs between each of the programs on the inputs given.

To test file sizes, we iterated on a “Romeo and Juliet" text file obtained through the Folger Shakespeare Library [8] and used two different rulesets. The first ruleset matched on a singular regular expression, "ROMEO", whilst the other tested ruleset tested on 26 regular expressions. We tested file sizes ranging from 100 kilobytes to 1 gigabyte. Bigger file sizes were considered, however the current GPU implementation prevented further testing as the implementation is limited by the device memory. To ensure consistency and reduce noise, we conducted each test 3 times and averaged the results. Additionally, we took into account states of the devices before and after executing the pattern match.

6.2 Metrics Measured and Comparisons Made

We measured the timing, power consumption, and memory usage of each computing device whilst varying the file size being parsed and the ruleset we used.

We used the turbostat command-line utility to obtain the timing of the program in seconds and the power usage in watts. The idle power usage of the CPU was also taken into account by polling the turbostat utility on the CPU whilst idle. Additionally, the maximum resident set size (RSS) was obtained through the time command with the verbose option.

The timing on the GPU was obtained by logging time stamps within the pattern matching program itself. We also used the NVIDIA System Management Interface (nvidia-smi) to obtain the total memory usage and total power consumption. The power consumption was obtained by polling the nvidia-smi command utility constantly over the program execution and obtaining the average power in watts and total energy consumed in joules.

As we weren't able to have the Alveo U50 run our program, we had to rely on estimations for the time and power. On the FPGA side, we used Vivado to tell us how much resources our program used, how much power was used, and if our program met timing constraints which we changed in increments of 0.05 ns. On the CPU side, we created a program which simply read in the input file, sent it to a file where it would go to the FPGA via the PCIE bus if our program was running, then read from that file which would be the output of our program and piped the output to the console.

We then measured this the same way we measured CPU pattern matching and combined this with the FPGA results. For timing, we took the maximum between the time it took for the CPU to read the input file and the time it takes for the FPGA to process one byte multiplied by the number of bytes in the input file. Simply taking the maximum doesn't account for file syncing delays but these should be negligible.

For FPGA power measurements, we added the total number of joules the CPU and the FPGA would take then divided by the total runtime calculated using the method mentioned previously. However, this does not account for file synchronization overhead, which could be significant. This is because the CPU needs to keep a lookout for when the file is ready to be written/read, amounting to some constant amount of power even if nothing is happening. As a result, our estimates likely underestimate the power by a small but not insignificant amount.

7 RESULTS [9 PTS]

7.1 Timing

7.1.1 Setup Time. We split up timing into setup time, the time it takes regardless of the size of the input file, and runtime, the time it takes to process the input file. We did this because for FPGA, the setup was a huge portion of the total time as you can see in Table 1 but was negligible for the rest. The reason for this is threefold. First, Vivado takes a while to design a circuit and create a memory configuration, taking a little over 5 minutes which increases depending on how complex the regex is. Second, flashing the memory configuration to the FPGA takes a constant 94 seconds each time. Third, we couldn't measure the time it takes to restart, so it is not included in the setup time. However, as an estimate, typical laptops take around 5-30 seconds to restart.

7.1.2 Runtime. As you can see in Figure 8, with just 1 regex the CPU is the most efficient which could be due to the fact that the regex is just a string so Boyer-Moore performs well. The FPGA implementations follow close behind but APSim's implementation departs when the input file is 10MB and our implementation departs

	Setup Time 1 Regex (s)	Setup Time 26 Regexes (s)
CPU	0	0
GPU	0	0
FPGA (Ours)	405	417
FPGA (APSim)	407	416

Table 1: Table with the setup time of each program rounded to the nearest second.

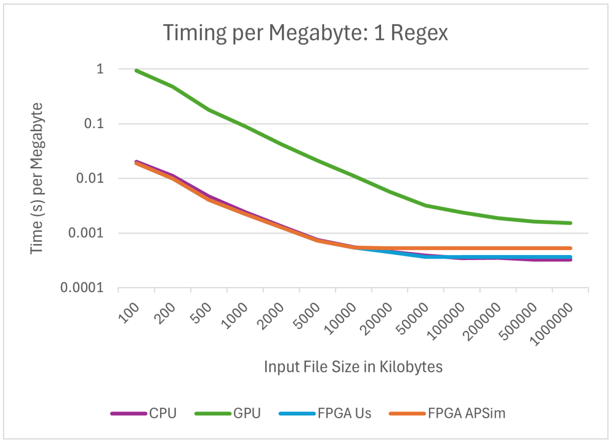


Figure 8: Runtime efficiency of pattern matching on a single regex.

at 50MB. This is due to file I/O being the bottleneck at small file sizes due to loading more memory from disk than the program needs, but as the files get larger, file I/O approaches its optimal efficiency and our FPGA's processing speed becomes the bottleneck. Meanwhile, the GPU implementation consistently takes more time per megabyte than all other implementations. This can be attributed to the time taken for copying memory from the host (CPU) onto the device (GPU).

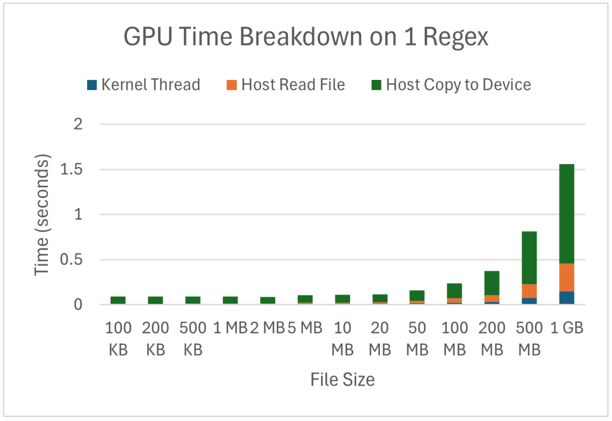


Figure 9: Time breakdown of GPU implementation on a single regex.

In Figure 9, we can see that the "Host Copy to Device" memory operation (HtoD) takes the majority of the time spent in the GPU implementation. This operation is needed when the CPU copies the input file and ruleset to the GPU. The "Host Read File" operation is another memory operation where the CPU reads the input file and ruleset from memory. The shortest operation comes from the parallelized kernel thread. This is within expectations, as memory operations such as HtoD are the usual bottlenecks in GPU applications.

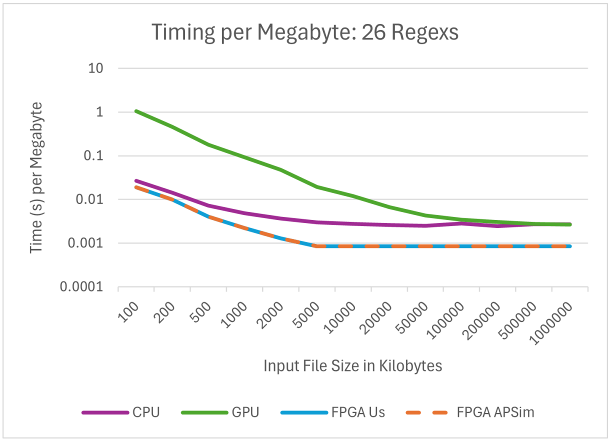


Figure 10: Runtime efficiency of pattern matching on 26 regexes.

Figure 10 shows the runtime of each implementation when performed using a ruleset of 26 regexes. In this case, we see that initially the GPU has higher time per megabyte of input than other implementations but converges to be slightly faster than the CPU with a file size of 1GB. The FPGA implementations are much more efficient than the CPU and GPU and interestingly take the same time (0.8 ns per byte) even though our implementation was faster for 1 regex taking just 0.35 ns per byte while APSim's took 0.5 ns. The difference there is due to APSim's implementation having two extra controls which allow for resetting the automaton and pausing the processing. In our implementation, to reset the automaton you would need to pass in a character not in the language and to pause you would need to stop the clock. Both of these are doable but we would consider adding these controls to our implementation because the cost they add is negligible with a complex ruleset as we can see here.

With either ruleset, the FPGA and CPU implementations are faster or operate at roughly the same speed as the naive search GPU implementation due to it spending a lot of time performing memory operations between the host and device. It is worth noting that the parallelized kernel thread performs similarly to both CPU and FPGA implementations showing signs of further improvement if direct memory access or streamed data were to be involved (such as data through NVIDIA GPUDirect).

We additionally profiled the GPU implementations using nvidia-smi whilst performing pattern matching on a 1 GB file using 1 and 26 regexes. Refer to Figures 11 and 12 for the timelines of each call where the red bars are memory operations and the green bars are CudaDeviceSynchronize calls (which in turn, describe the parallelized thread call).

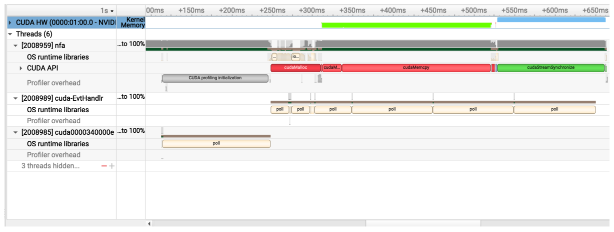


Figure 11: Timeline of GPU implementation on a single regex.

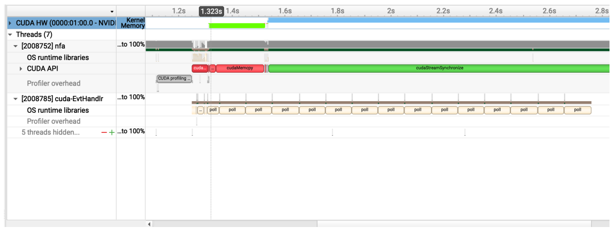


Figure 12: Timeline of GPU implementation on 26 regexes.

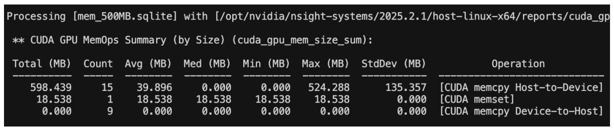
The timeline profiles show that increasing the size of the ruleset dramatically increases the time needed for the CudaDeviceSynchronize call to finish. Previous results in Figures 8 and 10 show that the GPU converges in efficiency and time to around the same point that the CPU implementation converges towards. However Figures 11 and 12 show that the timing the kernel thread shown by the CudaDeviceSynchronize call grows at a slower rate than the execution performed by CPU implementations.

7.2 Memory

7.2.1 CPU Memory. We looked to measure the maximum resident size of the CPU whilst executing each of inputs and rulesets. It was found that all inputs and rulesets had a 2560 KB maximum resident set size for its execution. This is to be expected as GNU grep performs its pattern match with a streamed input for its search file.

7.2.2 GPU Memory. The GPU pattern matching implementation showed a linearly scaling result in line with the size of the input file. This is also expected as the current GPU implementation looks to load both ruleset and input search file into the GPU before executing the pattern match. Figure 13 shows a breakdown of the highest memory operations used by the GPU from a pattern match of 26 regexes on a 500 MB input search file.

We see that the memory of the execution uses around 617 MB of memory total. Most memory usage comes from the CUDA memcpy HtoD, which is in line with how the current GPU implementation works where all of the input files are loaded into the GPU at the start. This shows that the current GPU implementation cannot handle files larger than its current memory and is not suited for such applications. We additionally tested this using 5 GB input files, which had confirmed our analysis as the program produced memory segmentation issues and halted personal devices.


```
Processing [mem_500MB.sqlite] with [/opt/nvidia/nsight-systems/2025.2.1/host-linux-x64/reports/cuda_gp

 ** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------------------
    598.439     15    39.896     0.000     0.000   524.288      135.357  [CUDA memcpy Host-to-Device]
     18.538      1    18.538    18.538    18.538    18.538        0.000  [CUDA memset]
      0.000      9     0.000     0.000     0.000     0.000        0.000  [CUDA memcpy Device-to-Host]
```

Figure 13: Memory breakdown of GPU implementation with 26 regexes on a 500 MB file input.

Implementation	STEs	LUTs	Registers	BRAMs
Ours w/ 1 regex	6	8974	18516	26.5
APSim w/ 1 regex	6	8976	18518	26.5
Ours w/ 26 regexes	56	9069	18544	26.5
APSim w/ 26 regexes	116	9107	18557	26.5

Table 2: Table with the number of state transition elements and FPGA resources used of each program.

7.2.3 FPGA Memory. Memory for the FPGA doesn't differ much between implementations because most of it is used by Xillybus but there are some differences. As you can see in Table 2, for the simple regex both of our implementations minimized to 6 STEs but for the ruleset we were able to minimize ours to be about half the size. The number of STEs used increases the number of LUTs and registers needed with our implementation using slightly less with the same number of STEs because of the previously mentioned lack of extra controls. Finally, the number of BRAMs is constant across implementations because Xillybus is the only thing that uses them.

7.3 Power

As you can see in Figure 14 and Figure 15, the CPU and GPU based pattern matchers have roughly the same amount of average power and increase slightly as the file size increases or when using the ruleset. The FPGA based programs follow the same pattern except that the average power goes down when using the ruleset. This is due to the FPGA itself just needing a constant 7.095 watts (7.096 for APSim's circuit of the ruleset) so most of the joules a run consumes are for loading the input file. This means that when using the ruleset which causes the FPGA to be what's taking most of the program time, the number of joules used isn't increasing as much as the amount of time the program takes resulting in a lower average power.

7.4 Key Takeaway

Overall, we see an increased performance boost in various areas on both parallel devices. This shows that pattern matching can be accelerated with such methods and can be beneficial for performance critical applications in future exploration. However, the setup times and practical use of using GPU or FPGA devices for pattern matching is still questionable. GPU is shown to be limited in some regards and can converge to slower times as seen in Figure 8. FPGA additionally requires setup prior to pattern matching which may not be beneficial in some use cases. However, hardware-accelerated pattern matching is viable in cases where a large streamed input needs to be processed such as in network security.

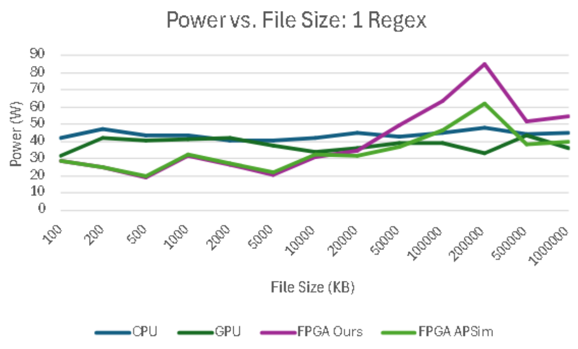


Figure 14: Average power of pattern matching on 1 regex

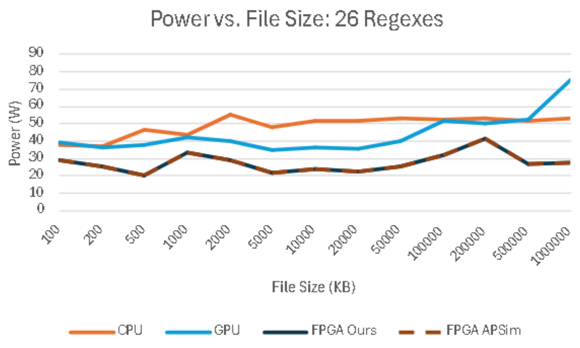


Figure 15: Average power of pattern matching on 26 regexes

8 LINK TO SOURCE AND OTHER RESOURCES [3 PTS]

See the following link for the public repository used for the project: https://github.com/joeyvongphasouk/582-course-project.

In addition we used several outside libraries. Refer to the following list for each library, its link, and its corresponding use case.

- grep(1) - https://man7.org/linux/man-pages/man1/grep.1.html : The grep command-line utility is used to search for patterns within files or input streams using regular expressions. Its extended variation egrep was used as a baseline implementation for traditional CPU-based pattern matching and was compared aginst other pattern matching implementations on other compute devices.
- CUDA-grep - https://github.com/bkase/CUDA-grep : CUDA-grep is a student project by Manish Burman and Brandon Kase that utilizes the CUDA API to accelerate matching on regular expressions. We iterate on their work to create fair comparisons and benchmarks against other pattern matching implementations.
- Nsight Systems - https://docs.nvidia.com/nsight-systems/index.html : Nsight Systems is a performance analysis tool developed by NVIDIA for profiling GPU applications. This tool was used to gather information on the GPU such as its power usage, overall memory utilization, and a general understanding through the GUI timeline of CPU/GPU activity.
- APSim - https://github.com/gr-rahimi/APSim : APSim is an automata processing simulator which we used as a point of comparison for our implementation of automata minimization and converting the automata to verilog code.
- Xillybus - https://xillybus.com : Xillybus is an FPGA IP Core that allows communication between the CPU and FPGA to be done via files on the CPU and a FIFO on the FPGA. They have IP Cores specific to devices and we were able to adapt an IP Core built for AMD (Xilinx) Ultrascale+ to the Alveo U50.

Many tutorials were used while working on the course project. Refer to the following list for each tutorial, its link, and its corresponding use case.

- Justin Davis' benchmarking introduction - https://edstem.org/us/courses/71791/discussion/6417495 : Justin Davis had shown the class how to use various benchmarking tools to measure specific metrics and can be used on the hpsslab devices. The Zoom recording and Google Collab resources provided insight into tools such as turbostat, NVIDIA System Management Interface (nvidia-smi) and NVIDIA Nsight Systems (nsys).
- GPU setup on WSL - https://peterchng.com/blog/2024/03/02/profiling-cuda-programs-on-wsl-2/ : The tutorial seen here provides a way to profile CUDA applications when running on a WSL2 Linux distribution. This was used to grant hardware permissions on personal devices to all users, which allowed for the additional recording of benchmark data.
- NVIDIA Nsight setup and tool explanation - https://www.youtube.com/watch?v=3DAYN-onSzY : This resource had shown how to run and profile arbitrary programs that used the GPU. Additionally, it showed insight into where to get additional information about the application ran and how to navigate the graphical user interface.
- Xillybus Setup - https://xillybus.com/downloads/doc/xillybus_getting_started_xilinx.pdf : This tutorial helped us set up Xillybus although it wasn't able to fix the incompatibility problems.

9 PROJECT/CLASS TAKEAWAY [2 PTS]

During this class, we learned about various types of compute acceleration and parallel methods of computation. In this project, we gained hands-on experience and learned more about how to use two specific types of compute acceleration devices (GPU and FPGA) as well as how to do performance analysis on these systems. This knowledge has improved our understanding of hardware-software interfaces which can be valuable in jobs that work with performance-critical systems and parallel computing. Additionally, this project has strengthened our ability to evaluate trade-offs between different acceleration platforms and has given us practical benchmarking experience that can be applied to future roles involving system optimization, embedded development, or high-performance computing.

10 FEEDBACK [+1 PTS (GOES TO PARTICIPATION)]

10.1 Hanuman's Class Feedback

The main thing I would change about this class is how our knowledge is tested. In the beginning of the class, I think we didn't have enough homeworks and I would have preferred assignments that went into more detail on how to use the things we were being taught like in HW4 and HW5 which were about CUDA/GPU. In the second half of the class, I felt like the project has too much emphasis on performance analysis instead of on understanding how to use the devices. It felt like you kept pushing for groups to use other people's code and use libraries that abstract controlling the accelerators like torch which results in students not getting any practice using the accelerators.

10.2 Joey's Class Feedback

The class covers a variety of accelerators, having dedicated lectures for each accelerator (with some having multiple like GPUs). One of the main assignments is the project given to students, where students explore an accelerator on a given application or benchmark. I believe this project is the most useful part of this class, however starting the project earlier and having more checkpoints to keep students on track will be very beneficial if implemented. We have seen many setup challenges in the project presentations that are based on project dependency. This is indeed one of the challenges of the project, but it would be more interesting to see other challenges that relate to how the application interfaces with the accelerator which I believe can be done through introducing the project earlier and having checkpoints for preliminary results.

Additionally, I think the course content is structured in a way that does introduce topics nicely, however I think FPGA being introduced later is a problem for those that want to get experience programming on an FPGA. Moving that up or introducing it at a broad level earlier could be helpful as we have seen many groups use FPGAs. Another more drastic approach is to de-emphasize FPGAs and ASICs in the beginning and have projects be focused on GPU acceleration (with special meetings for FPGA and ASIC projects). That way, students are able to have more similar projects for debugging and projects can be introduced earlier on.

REFERENCES

[1] H. Liu, S. Pai, and A. Jog, "Why GPUs are Slow at Executing NFAs and How to Make them Faster," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA: ACM, 2020, pp. 251–265. doi: 10.1145/3373376.3378471.

[2] Chiou, D., Sunwoo, D., Kim, J., Patil, N. A., Reinhart, W., Johnson, D. E., Keefe, J., & Angepat, H. (2007). FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate simulators. International Symposium on Microarchitecture, 249–261. https://doi.org/10.5555/1331699.1331723

[3] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19). USENIX Association, USA, 631–648.

[4] M. Haertel, "why GNU grep is fast," Freebsd.org, Aug. 21, 2010. https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html

[5] "Boyer Moore Algorithm for Pattern Searching," GeeksforGeeks, May 26, 2012. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

[6] "Boyer Moore Algorithm | Good Suffix heuristic," GeeksforGeeks, Jun. 21, 2017. https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/

[7] M. Burman, and B. Kase, "CUDA grep: A Hardware Accelerated Regular Expression Matcher," Cmu.edu, 2025. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s12/www/competition/bkase.github.com/CUDA-grep/finalreport.html (accessed Apr. 24, 2025).

[8] "Romeo and Juliet | Folger Shakespeare Library," www.folger.edu. https://www.folger.edu/explore/shakespeares-works/romeo-and-juliet/

[9] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs," in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), LOS ALAMITOS: IEEE, 2020, pp. 138–147. doi: 10.1109/FCCM48280.2020.00027.

[10] Glendenning, P., M. Tanner, J., C. Leventhal, M., & B Noyes, H. (2020). Methods and systems for representing processing resources (Patent No. 11816493). U.S. Patent and Trademark Office. https://patents.justia.com/patent/11816493