Cacey Ostic and Nick Petrilli
Parallel Processing
Project 4

Project Overview

For Project 4 we were tasked with recreating our Project 3 (MPI) but now implementing the solution with CUDA.

readFile Function

Our readFile function is where our input and pattern files get read. The function has four parameters: the filename, a pointer to a flattened array to hold the character contents of the file, and two integer pointers to store the number of rows and columns in the file. The file is read into the array to be used for pattern matching. If either of the files can't be opened, the function returns false.

__device__ checkForPattern Function

Our checkForPattern function is where the contents of both files are traversed to find the pattern in the input file. The function takes eight parameters: a pointer to the flattened character array representing the input matrix, a pointer to the flattened character array representing the pattern matrix, the numbers of rows and column for each matrix, and i and j variables for the row and column index where the pattern matching starts. Within a nested for loop, looping through the rows and columns of both arrays, we check that the characters in the input file match all of the characters in the pattern. If the pattern doesn't match, or we go out of bounds, our flag variable will be set false indicating the pattern was not found at that position. During this pattern matching process, we are ignoring any wildcard characters, indicated by '*', since they can match any character and be out of bounds. If a pattern is found, the function returns true for the main function to handle.

__global__ patternMatchingKernel Function

Our patternMatchingKernel function is the main kernel function used by the GPU. It is called by the CPU main to start the process of pattern matching. The function first calculates the stride, the number of total threads being used by multiplying the grid size by the block size. It also calculates k, the position in the input array to check for the pattern. The remainder of the method is inside a loop that calculates the row and column values and calls the checkForPattern function at that position. If a pattern is found, an entry is added to the resultCoords array. Lastly, k is increased by the stride. The loop continues until k is larger than the size of the input. The addition of the stride variable allows for the use of interleaved partitioning instead of sectioned partitioning, increasing the memory access time of the program.

Main Function

The start of the main function takes in 4 input parameters: the input file name, pattern file name, number of blocks, and number of threads for the program to run. After this, both the input file and pattern file are read into one-dimensional arrays for the pattern matching kernel to use.

Next, we create variables to store these arrays and the resultCoords array on the GPU and allocate the necessary space for them using cudaMalloc. After allocating the memory space, we use the cudaMemcpy function to copy the contents of the variables from the host to the device (CPU to GPU).

After allocating the space and copying the variables to the device, we call the patternMatchingKernel GPU function to perform the pattern matching and writing to the resultCoords array.

Once the patternMatchingKernel stops executing, we copy the contents of the resultCoords array from the device back to the host. At this point we use the cudaFree function to free the memory being used by the device variables.

Lastly, we write the coordinates from the resultCoords array to our different output files.

GPU Model

NVidia GeForce RTX 2060

Test Results

NOTE: This project uses CUDA 12.4. If you try to run the visual studio project on another computer, you may need to edit the Build Customizations to use the installed version of CUDA. (For the lab computers in Hancock, this meant switching from CUDA 12.4 to CUDA 12.2)

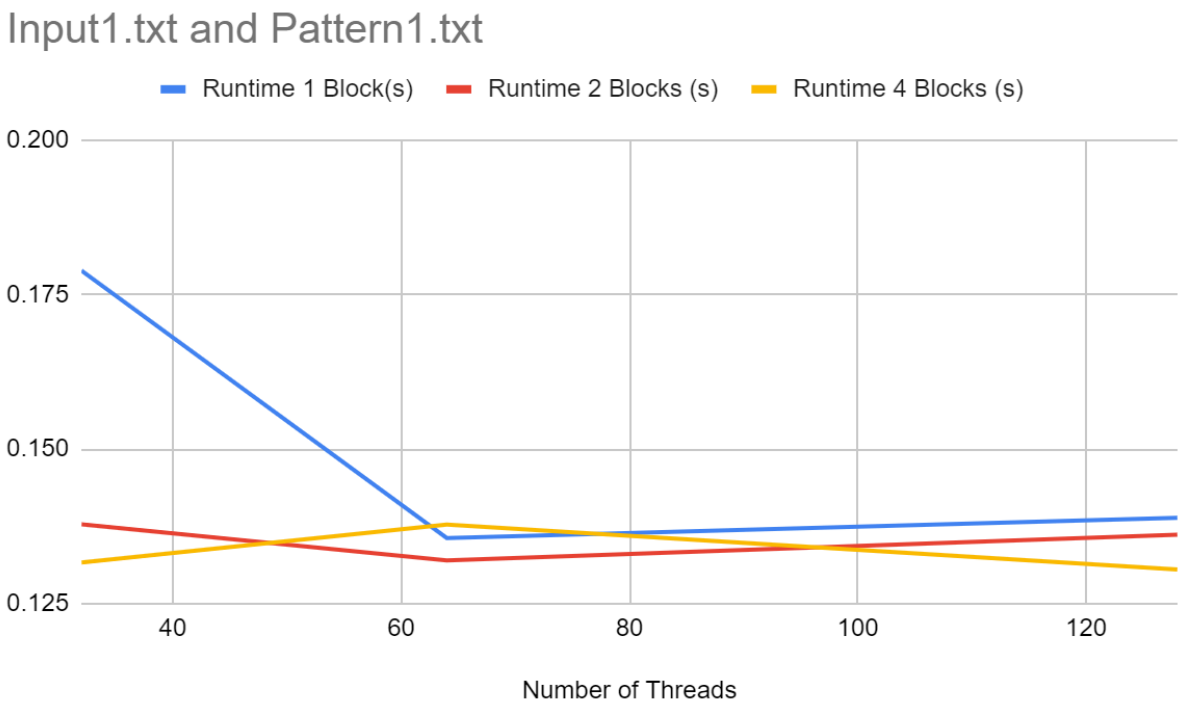
Input1.txt and Pattern1.txt

Input1 is one of the smallest test cases. It contains a small block of random characters. Eight occurrences of the pattern were inserted into the random block.

Pattern:

Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)
1	32	input1.txt	pattern1.txt	1	8	2097.152	0.178847
	64					2097.152	0.135623

	128					2097.152	0.138877
2	32					2097.152	0.137837
	64					2097.152	0.131986
	128					2097.152	0.136159
4	32					2097.152	0.131671
	64					2097.152	0.13779
	128					2097.152	0.130548



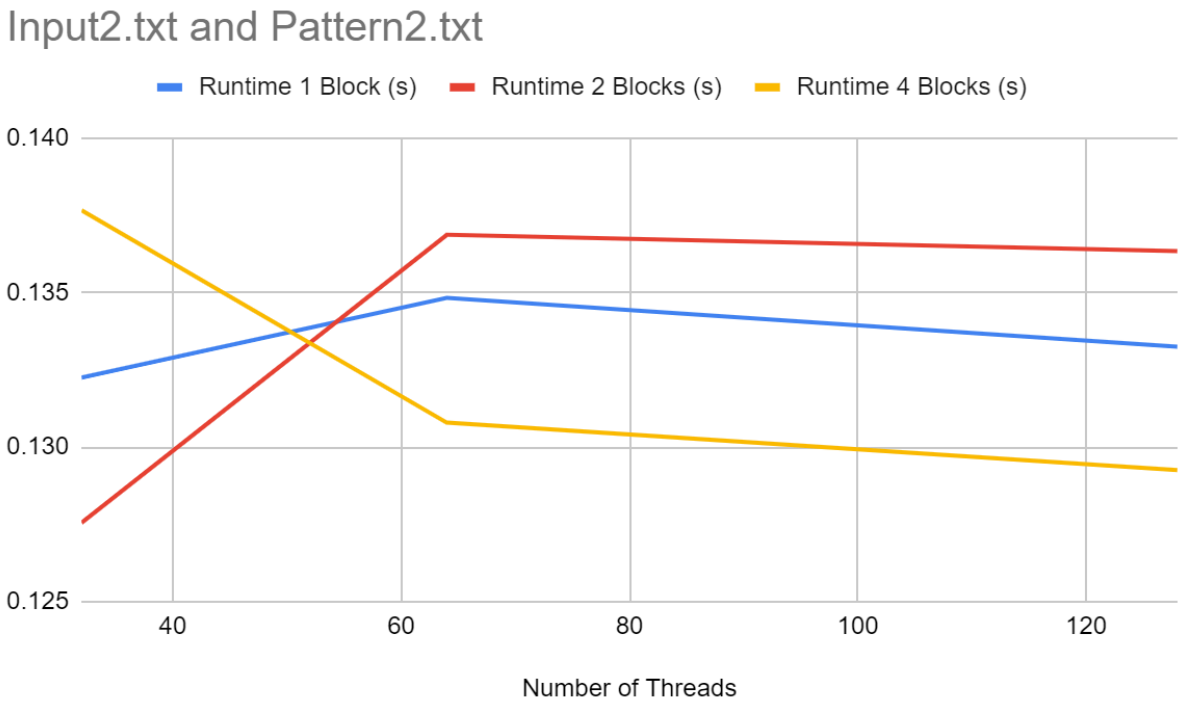
Input2.txt and Pattern2.txt

This test case is similar to input1 with random characters, but it is slightly larger. It also contains 0 occurrences of the pattern to test how the program responds to finding no matches.

Pattern:

Number of Blocks	Number of Threads	Input File	Pattern File	File Size	Occurrences	Memory (KB)	Runtime (s)

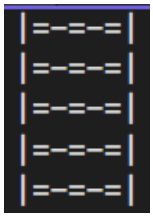
				(KB)			
1	32	input2.txt	pattern2.txt	6	0	2097.152	0.132248
	64					2097.152	0.134828
	128					2097.152	0.133244
2	32					2097.152	0.127556
	64					2097.152	0.136866
	128					2097.152	0.136342
4	32					2097.152	0.137661
	64					2097.152	0.13079
	128					2097.152	0.129254



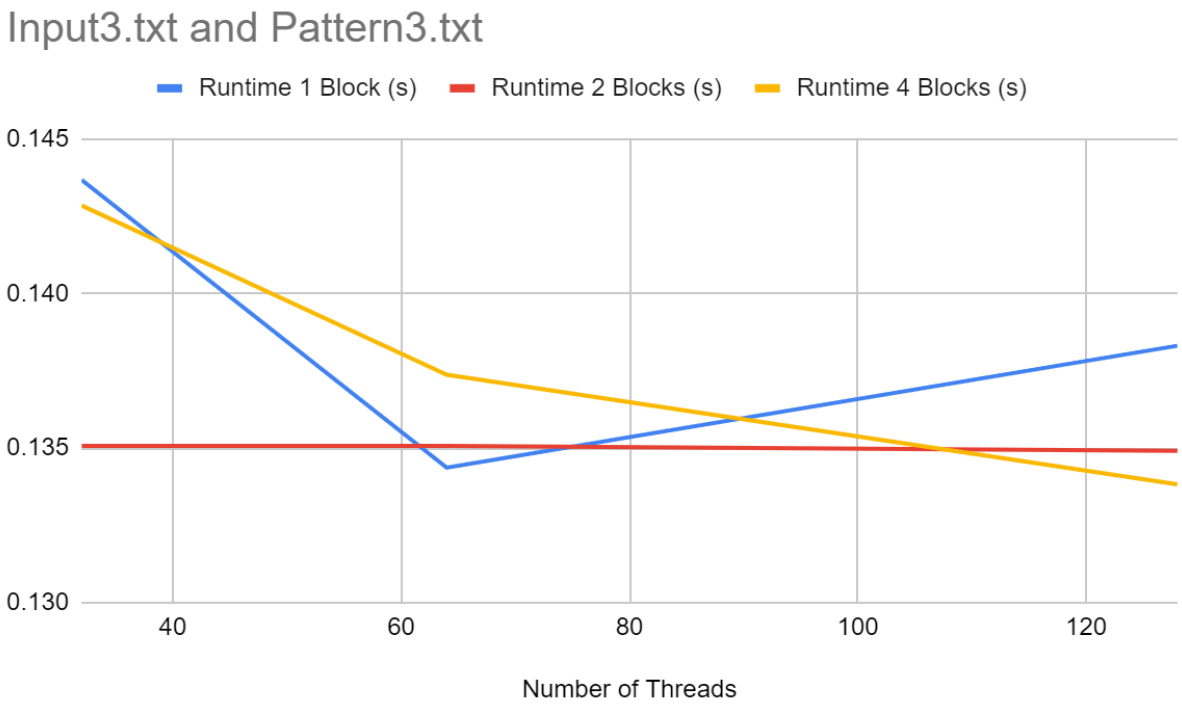
Input3.txt and Pattern3.txt

Input3 is the medium sized test case for the program. It is significantly larger than both Input1 and Input2 and contains many more occurrences of the pattern. The pattern itself is also larger than previous patterns.

Pattern:



Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)
1	32	input3.txt	pattern3.txt	76	295	2097.152	0.143674
	64					2097.152	0.134366
	128					2097.152	0.13831
2	32					2097.152	0.135071
	64					2097.152	0.135072
	128					2097.152	0.134909
4	32					2097.152	0.142846
	64					2097.152	0.13737
	128					2097.152	0.133825



Input4.txt and Pattern4.txt

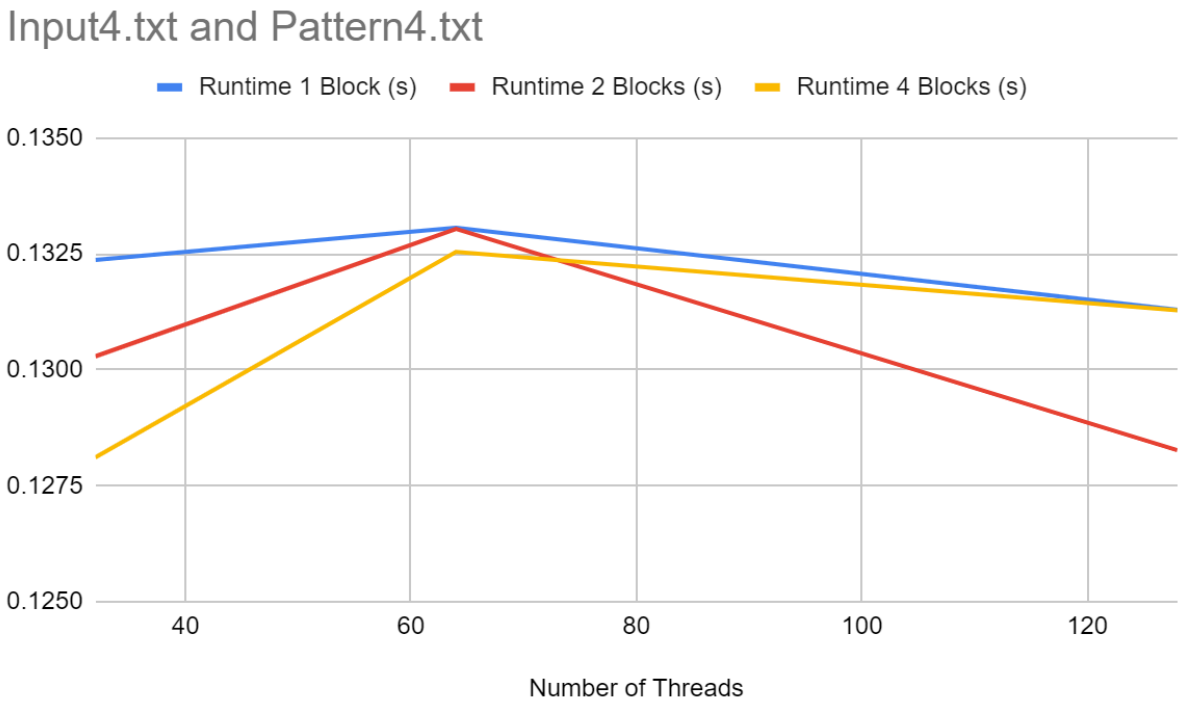
Input4 is a special test case with a small block of ascii characters. The pattern is the same size as and contains the same characters as the input.

Pattern:

```
hsgrrrflgrmljqtycrdtagfoosjsekptyqpkshumwjwckvuxevgrrbufygwnjbsz
warmgatrsxsrnngxhxjbmmxbubltwihbbchvqpmccxbnvzidllizedeatrsvyzsc
crpgjfnbrpuvyhkvvsepkovpuqkkkcbkqvnrndqrdwcbrnyqumswtayoollhunqg
ilyfrbgrfcmzhphxzvyyzgxrrbalgpmkepahjwiwapqscpojecmtijyvqfqfwyec
rfxnygclacjmxjnzxclddvpckktztzughxnpwkiqcdnqiuissnvwtaetyjbpckyr
mkwfeqwxopimykzablmeacumhknogxchabrdzwhvoexpsrpvplznpwqspkclneuw
iktszsmdzdhivclolmrmdzjcqhlpvkfkisanmslkwqzwzlqvgipaqgtzzrcfsjwr
kkuaokztormrsduzeeacmxgidtpmxwnahvviyatckqbzqspepfcnxnieadequvap
mnqsizicvodfwnnjcsqemvrpavmjqyflqjrltcidcsbufcreiulrkayvpchakmvr
ztmquvyuiihijbjamdrzoylherrkbemxyhmrrfwnicgfyqasjdcdhtlbkuehimkx
traqwxdmgwdlriscsedgzrimbbmtuxyxgghbwgumceqhszbqtytvkttztoysjpbw
qwwzltmupwrdjhcfcerlpnjrhcpjevqjagidosooiijnyzuhuwlwysphhfkyttpj
```

Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)

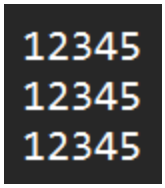
1	32	input4.txt	pattern4.txt	1	1	2097.152	0.132369
	64					2097.152	0.133061
	128					2097.152	0.131296
2	32					2097.152	0.130291
	64					2097.152	0.133038
	128					2097.152	0.128266
4	32					2097.152	0.128111
	64					2097.152	0.132543
	128					2097.152	0.131283



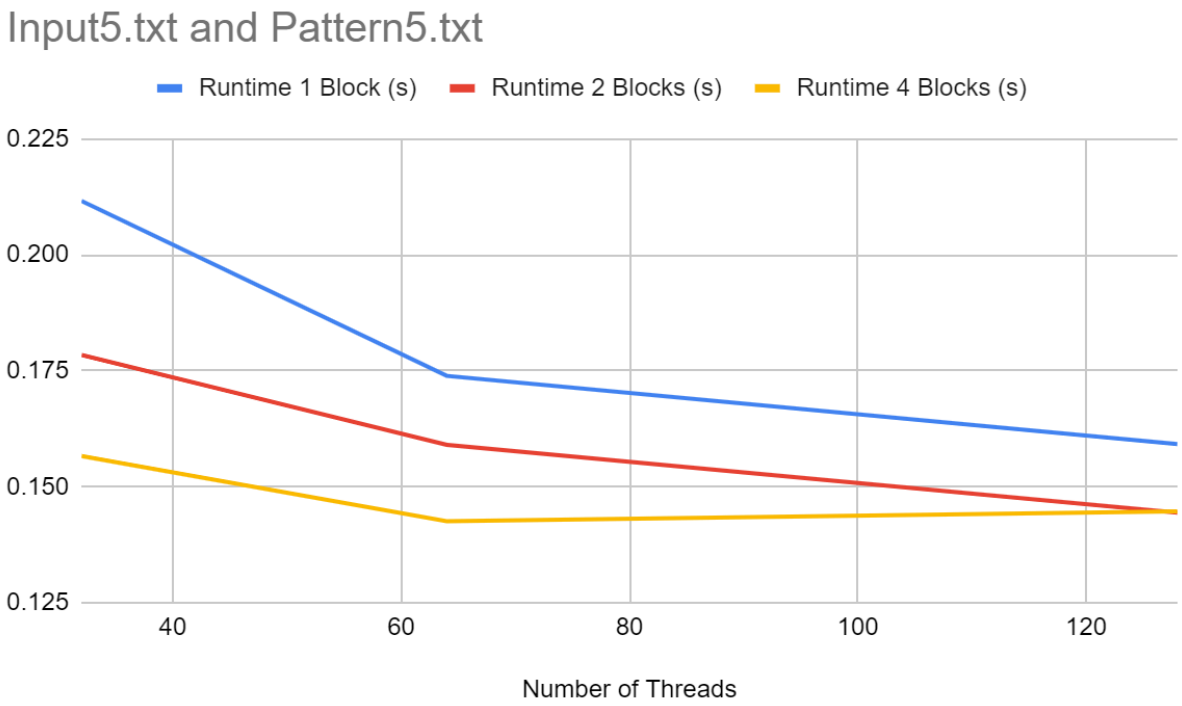
Input5.txt and Pattern5.txt

Input5 is the first of three large files tested with the program. It is over 1MB with six thousand occurrences. This is the first test case that show significant improvements in runtime.

Pattern:



Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)
1	32	input5.txt	pattern5.txt	1,503	6,000	14680.064	0.21163
	64					14680.064	0.173903
	128					14680.064	0.159202
2	32					14680.064	0.178417
	64					14680.064	0.159041
	128					14680.064	0.144402
4	32					14680.064	0.156609
	64					14680.064	0.142541
	128					14680.064	0.144703



Input6.txt and Pattern6.txt

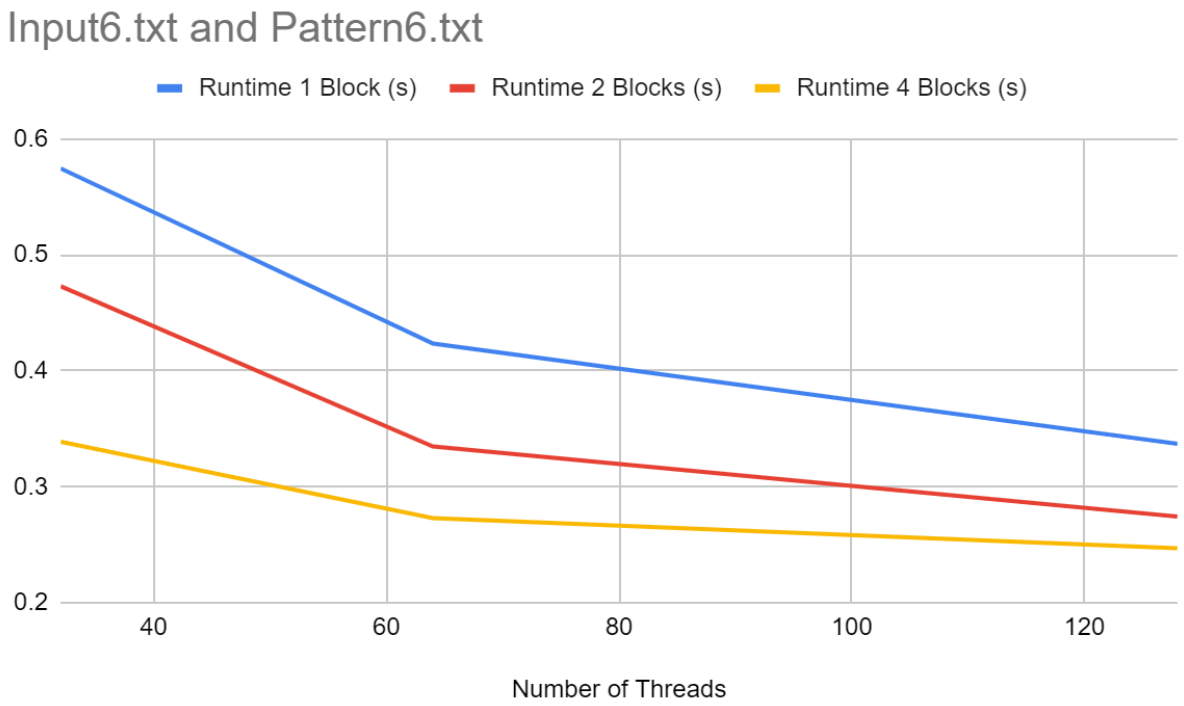
Input6 is the next large file tested. The file is 18MB and contains 131,000 occurrences. This is also a test for a single letter pattern.

Pattern:

```
a
```

Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)
1	32	input6.txt	pattern6.txt	18,762	131,000	178257.92	0.57454
	64					178257.92	0.423506
	128					178257.92	0.337026
2	32					178257.92	0.472968
	64					178257.92	0.334687
	128					178257.92	0.274227

4	32					178257.92	0.338777
	64					178257.92	0.272857
	128					178257.92	0.247009



Input7.txt and Pattern7.txt

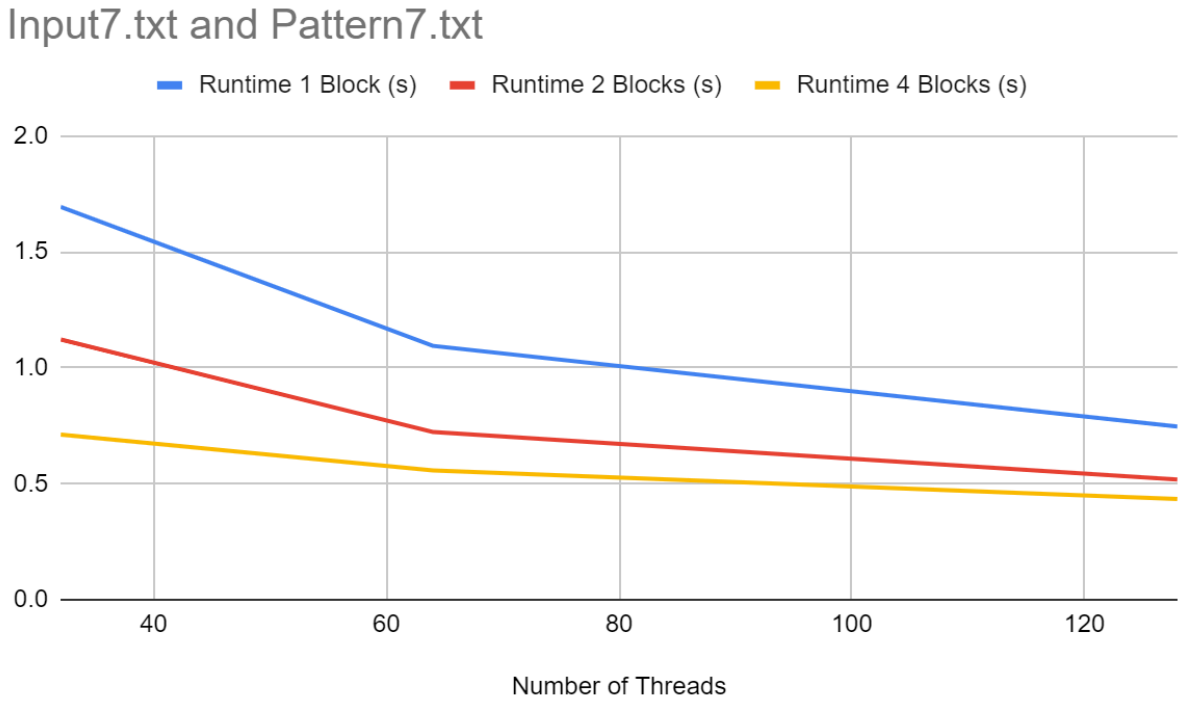
Input7 is the largest text file that was tested. It is nearly 50MB with close to 900,000 occurrences.

Pattern:

```
[pattern]
[second ]
[third  ]
[fourth ]
```

Number of Blocks	Number of Threads	Input File	Pattern File	File Size (KB)	Occurrences	Memory (KB)	Runtime (s)

1	32	input7.txt	pattern7.txt	49,176	875,000	455081.984	1.69367
	64					455081.984	1.09445
	128					455081.984	0.747193
2	32					455081.984	1.12194
	64					455081.984	0.722901
	128					455081.984	0.517855
4	32					455081.984	0.711638
	64					455081.984	0.556989
	128					455081.984	0.433732



Conclusion

The test cases above show that the results are much more consistent for larger files. In the first four test cases, using files that are smaller than 100KB, the runtime graphs were very inconsistent. All values are within a very small range, usually differing by only tens of milliseconds. The graphs exaggerate the values due to the small intervals on the y-axis. However,

for the three larger files, there is much more consistent data. For an increase of threads with consistent blocks, represented by each individual line, there is a decrease in runtime as thread count increases. Each line represents a different number of blocks, and as they increase, the runtime also decreases. This shows that using the GPU to decrease runtime only works well for larger files.

Looking at the memory increases across the test cases, they are roughly correlated with the file sizes for the larger files. For all four of the smaller files, there is a consistent amount of memory used. This may be a static amount of memory used by CUDA and there wasn't enough data allocated to overreach this limit. This likely means there is unused processing potential. However, for the three larger files, the amount of memory used by CUDA is roughly ten times the file size. For each of the trials using different numbers of threads and blocks, the amount of memory used is the same since the same data is allocated and copied.

Compared to MPI, CUDA processes large files much faster. Testing inputs 5 and 6 using MPI resulted in runtimes that were orders of magnitude larger than CUDA runtimes. Trying to test input 7 resulted in the computer nearly freezing. However, the runtimes of smaller files were much better with MPI than CUDA. In conclusion all three of these points show that CUDA is only good for processing sufficiently large files.