Politecnico di Torino

GPU Programming
01URVOV

Master's Degree in Computer Engineering

String Matching Algorithms
Project Report

Referents:
Prof. Luca Sterpone
Prof. Bartolomeo Montrucchio
Dr. Josie Esteban Rodriguez Condia
Dr. Corrado De Sio

Author:
Matteo Isoldi

Contents

CHAPTER 1

Introduction

String matching is a term used to refer to the problem of finding the locations where one or more strings, called pattern, are positioned within a larger text. It represents an essential problem in the computer science field, given its numerous applications in text elaboration software, text research tools, online search engines, online dictionaries and many other contexts. Another relevant field, greatly influenced by the string matching problem, is the bioinformatics domain, where molecular biology problems on DNA sequences can be formulated as string matching problems. Given the amount of data to be searched, various algorithms have been designed to reduce latency, by applying different strategies. However, with the continuous increase of the amount of data available, the time required to completely scan texts for searching patterns still represents a critical bottleneck. One solution could be the parallelization of such algorithms, which can be easily achieved with GPUs.

The goal of this project is to parallelize three, state-of-the-art, string matching algorithms using the CUDA platform and to compare the measured performance with their serial counterpart. An additional comparison is then performed to evaluate the impact of the realized algorithms' optimizations with regards to the underlying GPU architecture. The hardware platforms chosen to host the program's execution and where the profiling operations are carried out are the NVIDIA Jetson Nano and NVIDIA RTX 3070. In the following chapters, a detailed report of the algorithms and implementations is provided: in Chapter 2, information regarding GPUs, the CUDA platform, the NVIDIA hardware platforms and the chosen algorithms are reported, while Chapter 3 and Chapter 4 contains information regarding their actual implementation and the obtained results' analysis. Chapter 5 contains the conclusions of this project's work.

CHAPTER 2

Background

In this chapter, a brief description of a GPU and the CUDA library, used to interact with it, is provided. In the following section, a summary of the characteristics of the architectures employed for the project can also be found. After that, a simple explanation of the three selected algorithms is given, to better understand the parallelization process described in the next chapter.

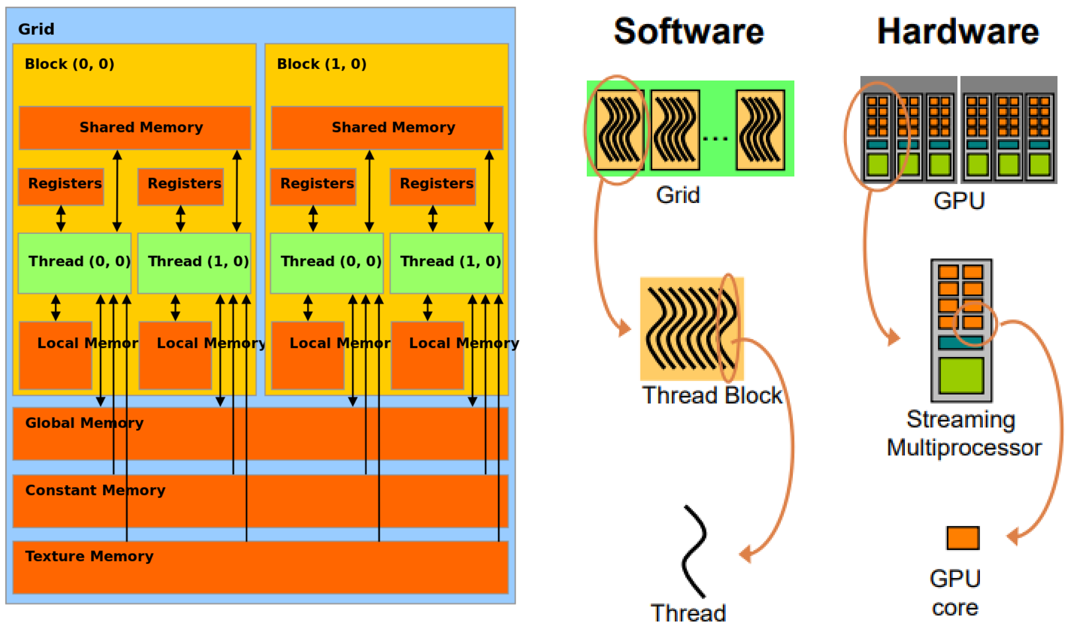
2.1 Graphic Processing Units (GPUs)

A GPU is an highly parallel microprocessor composed of thousands of cores, managed through a computational hierarchy, and featuring an internal memory hierarchy. Initially designed to render complex 3D scenes, nowadays they are also used to accelerate the elaborations of large sets of data in scientific and technical applications. Given that they are not used solely to solve graphic-related problems, they are also called General Purpose GPUs. When approached in this way, the device is seen as a co-processor by the CPU, with an higher throughput computation and an high bandwidth memory, which can be used for the implementation of highly parallelizable portions of code. To correctly harness the GPGPU capabilities, the CPU has to coordinate every operation involving the co-processor: the movement of data from the main memory to the device's memory, the launch of the various functions, called *kernel*, and the retrieval of results.

The processors of the device are split into *cores*, also called *streaming multiprocessor*, each equipped with a *locally shared memory*, whose content is shared by all processors contained within the core. A streaming multiprocessor can be composed of one or more blocks, each of which is characterized by a *warp scheduler*, allowing the dispatch of multiple common instructions per clock cycle (typically 32 instructions). The device, thus, exhibit parallelism both across the cores and within each core. From the programming point of view, kernels are executed by parallel threads (virtualized processors), which are grouped into blocks (virtualized streaming multiprocessors). Thread blocks can be further grouped into grids. The hardware has the responsibility to assign, at runtime, virtual blocks to the cores, whenever one of them is available for executing new computations.

The memory hierarchy is composed of different levels, each granting a certain performance-size trade-off. A complete scheme can be seen in Figure 2.1, however only three levels are relevant for the purpose of this project:

- *global memory*: accessible by all processors of the device and by the CPU of the system. It is the largest memory contained on the device (in the order of several GB) and it is separated from the hardware used to implement the GPU core (streaming multiprocessors, processors, caches, ...). For this reason, its access times are higher than the ones regarding other internal memories (like caches, shared memories or registers). Its contents have the lifetime of the launched application;



Source: [1]

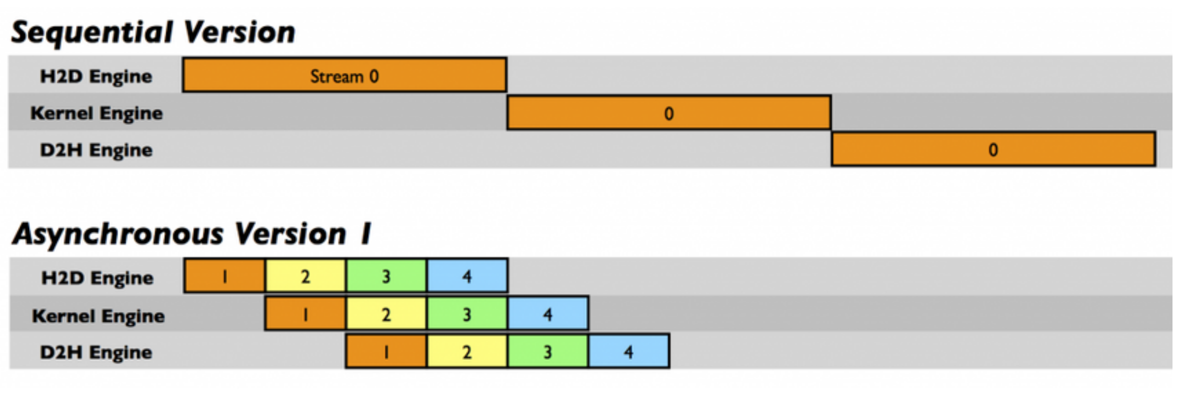
Figure 2.1: On the left, the GPU memory hierarchy scheme. On the right, the GPU virtualization of hardware resources is shown.

- *local shared memory*: accessible by all processors pertaining to the same core, it allows for better read/write performance with respect to the global memory. This is due to its implementation on the GPU core, its vicinity to the processors and the underlying technology, typcally similar to that of an L1 cache. It can contain much less data than the global memory (in the order of 10-100 KB). The data contained within it have the lifetime of the block;
- *registers*: fastest memory of the device. They are accessible only by the related thread and, for this reason, their data is flushed whenever the kernel completes its execution. Whenever a thread has to store more data than the one the registers can contain, they are moved into the *local memory*, a section of the global memory dedicated to the single thread. Accessing the local memory worsens the performance of read/write accesses.

Another component harnessed by the GPU device to increase parallelization are the *CUDA streams*. They are basically execution queues that operate independently from each other. Every queue contains a series of CUDA operations that the GPU has to perform in order. Using streams allows for:

- concurrent execution of more than one kernel per GPU;
- concurrent data transfers both from and towards the main memory;
- concurrent execution on device/CPU and data transfers from CPU or device.

An example of the advantages of using streams can be seen in Figure 2.2, where the same program is executed on the same set of data with and without their utilization. In the second case, the kernels are executed on a smaller subset of data, allowing for movement of data to be placed in parallel to the kernel execution.



Source: [2]

Figure 2.2: Execution of a program with and without using streams.

2.1.1 CUDA libraries

CUDA (Compute Unified Device Architecture) [3] is a software layer that provides direct access to the GPU's virtual instruction set and computational elements to higher level programming languages like C and C++. It comprises a development toolkit for compiling, debugging and profiling GPGPU's applications.

2.1.2 NVIDIA Jetson Nano

NVIDIA Jetson Nano [4] is an embedded computing module and a developer kit from the NVIDIA Jetson family. It has a quad-core ARM Cortex-A57 CPU, 4 GB of RAM and a 128-core NVIDIA Maxwell GPU that allows it to reach a maximum of 472 GFLOPS. It contains only one streaming multiprocessor, which is divided into 4 blocks, each with $\sim$ 49KB of shared memory and 32768 registers.

2.1.3 NVIDIA RTX 3070

NVIDIA RTX 3070 [5] is a GPU of the GeForce 30 series, based on the Ampere architecture. It has 8GB of global memory, shared by all 5888 CUDA cores the device contains, divided between 46 streaming multiprocessors. Each core is composed of 4 blocks, characterized by 65536 registers and $\sim$ 49KB of shared memory. It can reach up to 20.3 TFLOPS for FP32 and FP16 operations.

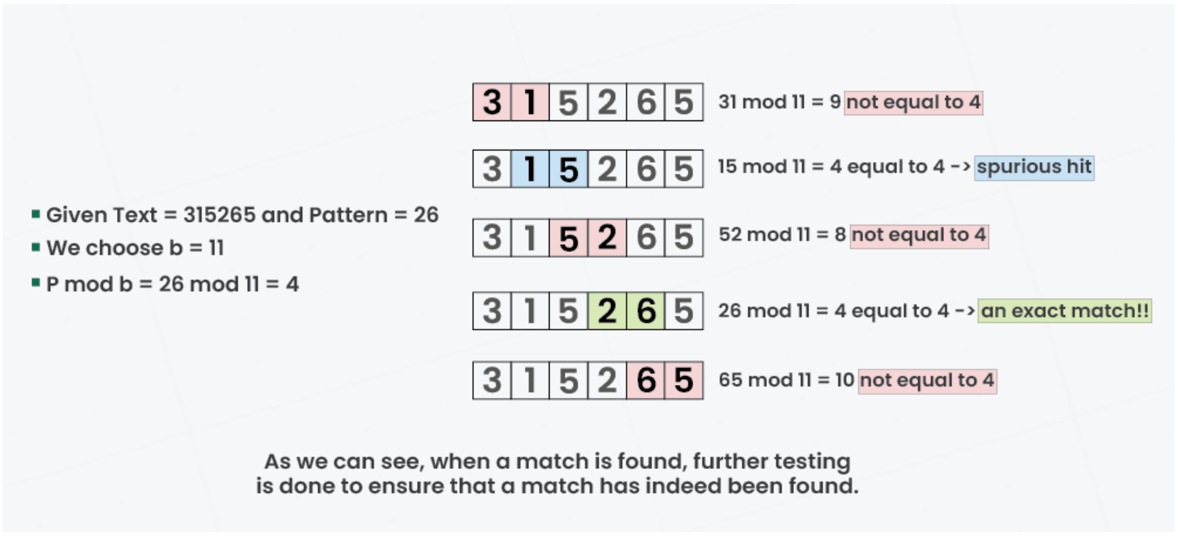
2.2 String Matching Algorithms (SMA)

In this section, a brief explanation of the three algorithms implemented for GPU execution is provided. The selection of the state-of-the-art and research-level algorithms to parallelize has been performed based on two properties: the solutions had to be exact (the end result should show the exact position of the matching, not only if the substring has been located or not) and the search had to be for one pattern at a time. The chosen algorithms are:

- **Rabin-Karp** [6], based on hashes computation for faster sequential comparisons;
- **Knuth-Morris-Pratt (KMP)** [7], based on the observation of mismatches, to bypass the re-examination of previously matched characters;
- **Boyer-Moore** [8], based on pre-processing the pattern to identify portions of the text to skip.

2.2.1 Rabin-Karp

Developed by Richard M. Karp and Michael O. Rabin in 1987, it filters out the positions of the text where no matching could occur by calculating and comparing string hashes, thus speeding up the search process. Therefore, every possible substring of the text is mapped to an integer and later compared to the pattern's hash. When the two hashes differ, there isn't any match, while, when they equal each other, there is a possibility of a match, since not every substring can be mapped to a unique value. In this case, a direct comparison between the two string's characters has to be performed in order to be sure that the found match is a correct one. An example of the search process is shown in Figure 2.3. To hash the text's substrings and the pattern, a rolling hash function is generally used,


Given Text = 315265 and Pattern = 26
We choose b = 11
P mod b = 26 mod 11 = 4
3 1 5 2 6 5
31 mod 11 = 9 not equal to 4
15 mod 11 = 4 equal to 4 -> spurious hit
52 mod 11 = 8 not equal to 4
26 mod 11 = 4 equal to 4 -> an exact match!!
65 mod 11 = 10 not equal to 4
As we can see, when a match is found, further testing is done to ensure that a match has indeed been found.

Source: [9]

Figure 2.3: Rabin-Karp algorithm execution example.

based on the formula described in Equation 2.1, where s is the string to encode while p and n are design parameters. For the project implementation, the pair of values chosen is $(p, n) = (256, 101)$.

$$\begin{aligned} hash(s) &= (s[0] + s[1] \cdot p + s[2] \cdot p^2 + \cdots + s[n-1] \cdot p^n - 1) \bmod n \\ &= (\sum_{i=0}^{n-1} s[i] \cdot p^i) \bmod n \end{aligned} \tag{2.1}$$

The rolling version of the formula is constructed in a way that makes use of the loop to rapidly remove the component of the previous string and add the one related to the following substring. The formula for updating the hash function during the loop movement is represented in Equation 2.2, where h is the highest power of p to associate to the new character's hash component.

$$hash(s) = (p \cdot (hash(s-1) - s[i] \cdot h) + s[i + len(pattern)]) \bmod n \tag{2.2}$$

This optimization strategy grants an average performance of $\Theta(m + n)$ and a worst-case performance of $\Theta(m \cdot n)$, where n is the text's length and m the length of the pattern string. The worst-case scenario is encountered when at each value of i a possible match is identified.

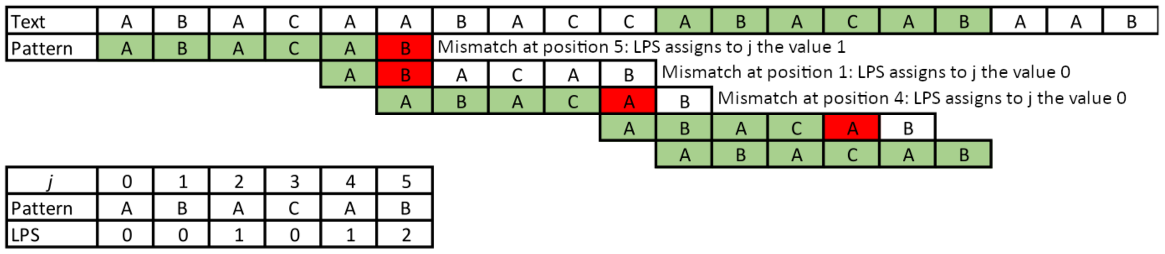
2.2.2 Knuth-Morris-Pratt (KMP)

Developed by James H. Morris, Donald Knuth and Vaughan Pratt in 1977, it is based on the idea that, whenever a mismatch occurs, some of the characters, which will appear on the next window, were already examined and, thus, can be exploited. In order to make use of the mismatch's data, a pre-processing step is required to build a prefix table, also known as LPS (Longest Proper Prefix which is also a Suffix), for the pattern. Inside the LPS table, which has the same size as the pattern string, is stored, for each index i, the length of the longest proper prefix that is also a suffix for the first i characters. An example of the content of the table can be seen in Table 2.1.

j	0	1	2	3	4	5	6	7	8	9
p[j]	A	B	C	D	A	B	F	E	A	B
LPS	0	0	0	0	1	2	0	0	1	2

Table 2.1: Prefix table values for pattern ABCDABFEAB.

During the text's scan, whenever a mismatch is identified, the LPS table is accessed to check if a prefix similar to the suffix of the pattern's substring, ending with the character preceding the one which causes a mismatch, can be found. In the case it exists, the value is used to determine the index for the start of the new comparison cycle. Otherwise, the pattern is searched starting from the first character. Either way, the index for the text's scan doesn't change, causing many comparisons to be avoided. The process is demonstrated in Figure 2.4.



Mismatch at position 5: LPS assigns to j the value 1

Mismatch at position 1: LPS assigns to j the value 0

Mismatch at position 4: LPS assigns to j the value 0

j	0	1	2	3	4	5
Pattern	A	B	A	C	A	B
LPS	0	0	1	0	1	2

Figure 2.4: KMP algorithm execution example.

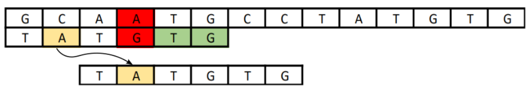
Due to its optimization strategy, the algorithm has a worst-case performance of $\Theta(n)$, where n is the length of the text. It has a small overhead of $\Theta(m)$ for the computation of the LPS, where m is the length of the pattern to be searched.

2.2.3 Boyer-Moore

Developed by Robert S. Boyer and J Strother Moore in 1977, this string search algorithm constructs two additional tables, based on the properties of the pattern string, so as to skip significant portions of the text. The tables' content is computed using two different approaches, which are exemplified in Figure 2.5:

- **Bad-Character rule**, which proposes a shift of the pattern to the right every time the comparison fails. The shift has the goal of placing an occurrence of the text's symbol, which caused the mismatch, in line with the mismatched occurrence in the text. When no equal character can be found inside the pattern string, the rule proposes a shift of the pattern's head past the point of mismatch.

- **Good-Suffix rule**, which shifts the pattern in order to place a substring, similar to the common suffix located on the point of mismatch, in line with the mismatched occurrence. In case the text's suffix cannot be found inside the pattern, a prefix has to be searched and moved to the failed comparison point. When neither a prefix or a suffix is found, the pattern's left end is moved to the right of the point of mismatch.



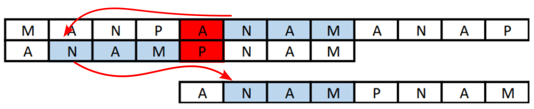


Figure 2.5: On the left, an example of the Bad-Character rule. On the right, an example of the Good-Suffix rule.

Both the tables are evaluated at runtime and the resulting shift of the pattern is determined by the greatest of the values proposed by the two tables.

The algorithm reaches a worst-case performance, in case a pattern is found, of $\Theta(n \cdot m)$, with an overhead for the pre-processing elaborations of $\Theta(m)$. n indicates the length of the text string while m is the pattern's length.

CHAPTER 3

Implementation details

In this chapter, the techniques and strategies used to shape the sequential algorithms into their parallel versions are explained. Then, a general description of the host code built to control the device's operations is reported.

3.1 Device code

Given the strong sequential nature of the algorithms' defining tactics, the main approach behind their GPU implementation is through a "Divide and Conquer" strategy. Other approaches to manipulate the computations, making them more optimal from a theoretical point of view, exist but, in practice, they all obtained worse performance than a simple adaptation. An example of this are the three Rabin-Karp implementations reported by S. Ashkiani, N. Amenta and John D. Owens [10].

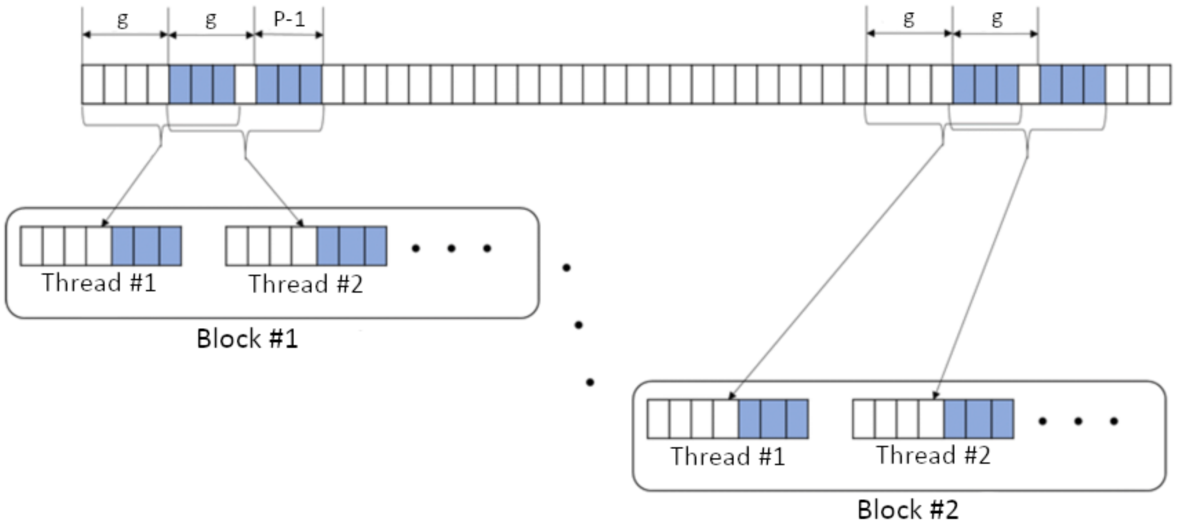


Figure 3.1: Mapping of the text's block to the device's threads.

Using the "Divide and Conquer" strategy, the text to be searched is divided into multiple blocks of equal lengths, each of which is mapped to a specific thread. Thus, the algorithms maintain the same structure and operations, with the addition of several statements to associate each thread to a specific block and for the possible utilization of the shared memory. The length of each block is a parameter that has to be specified at runtime by the user, called *granularity.* Its value is then modified by the

	Sequential version	**Parallel version**
Rabin-Karp	$\Theta(m \cdot n)$	$\Theta(\frac{m \cdot n}{PR_{SM} \cdot SM_{device}})$
KMP	$\Theta(n)$	$\Theta(\frac{n}{PR_{SM} \cdot SM_{device}} + P)$
Boyer-Moore	$\Theta(m \cdot n)$	$\Theta(\frac{m \cdot n}{PR_{SM} \cdot SM_{device}})$

Table 3.1: Worst-case efficiency of the algorithms before and after parallelization.

device code to make sure that the substrings associated to the last characters of the chosen interval are checked. The sizes of the block, as seen by both the host and each individual device thread, are defined as:

$$\begin{aligned} B^{host}_{size} &= g \\ B^{device}_{size} &= g + P - 1 \end{aligned} \tag{3.1}$$

where g is the granularity, or size of the block as dictated by the external input, and P is the pattern's length. An illustration showing the splitting of the input text can be found in Figure 3.1. A blue color is used to highlight the additional characters each thread has to consider for correctly controlling the block requested by the user. Table 3.1, instead, reports the worst-case scenario efficiency for each algorithm, obtained due to the reduction of the n value. It is to mention that their values don't depend directly on g but rather on the computing capabilities of the hardware architecture and the amount of parallelization it can achieve. This is demonstrated by PR_{SM}, the number of processors inside each core, and SM_{device}, the number of streaming multiprocessors, which define the maximum level of parallelization achievable by the device.

For each algorithm, two kernels have been defined:

- a **naive** kernel, which performs the research inside the text without making use of the advantages provided by the shared memory and through one stream only;
- an **optimized** kernel, which makes use of the shared memory to quickly access the pattern's data and its pre-processed information, speeding up the elaborations, and allows for more than one stream to be created (the number of streams is a parameter requested to the user at start-up), permitting further parallelization.

3.2 Host code

The main program requires two paths as arguments: one for the text string and one for the pattern string. The pattern file can contain up to eight different patterns, each specified on a separated line. The program, based on the number of patterns, will execute a single pattern search or run multiple searches concurrently using two different structures. The general flow of operations for managing the GPU in both type of scans can be seen in Figure 3.2.

After reading from the indicated files the text and pattern's strings and retrieving the parameters from user input, the program is tasked with the configuration of the device. The blocks are statically set up to allow 256 threads to work concurrently, while the grid is managed dynamically, based on the text's size and the granularity requested by the user. Next, based on the request of the user, the stream data structures are allocated, together with the GPU memory to host the pattern, its pre-elaborated information and the text string. A memory transfer is then executed for the pattern information. This last operation ends the general preparation phase and it is followed by an intermediate phase for custom instruction execution, depending on the chosen algorithm to run (pre-processing of the pattern string and memory transfer of the resulting table or tables to the device gloabal memory). Finally, the "execution" loop can start. The loop runs as many times as the number of streams to launch concurrently and it is composed of three main operations:

1. an **asynchronous memory transfer** of the text's information from the main memory to the GPU global memory;
2. an **asynchronous kernel launch**, to execute the preferred algorithm on the copied portion of the text;
3. an **asynchronous memory transfer** to recover the results' information.

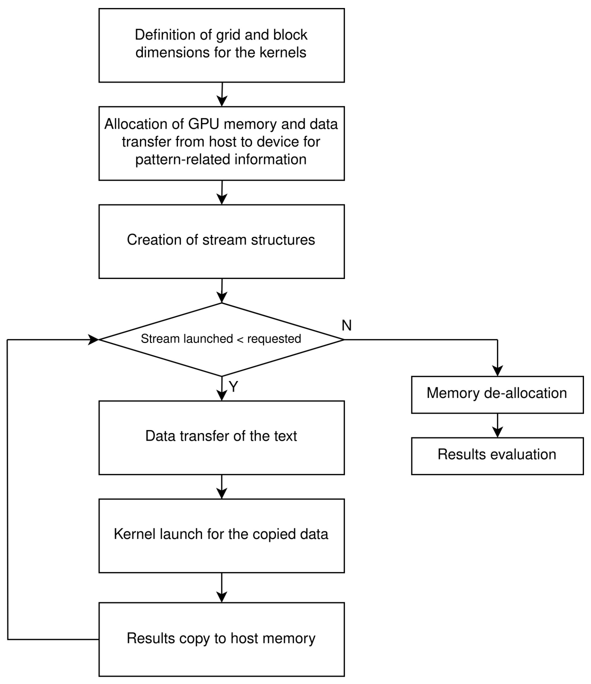

Definition of grid and block dimensions for the kernels
Allocation of GPU memory and data transfer from host to device for pattern-related information
Creation of stream structures
Stream launched < requested
N
Y
Memory de-allocation
Results evaluation
Data transfer of the text
Kernel launch for the copied data
Results copy to host memory

Figure 3.2: Pseudo-code for the GPU utilization portion of the code.

The process previously described is related to the single pattern's search of the text. The multiple pattern search follows the same flow while presenting some key differences:

- only the Rabin-Karp algorithm can be selected;
- only eight patterns can be searched concurrently, each mapped to a single stream;
- the text is transferred to the device memory outside the execution loop. Only the stream's pattern and computed results are transferred using asynchronous operations.

CHAPTER 4

Results

To test the newly developed program, texts, provided by the SMA Research Tool [11], were used. For the following discussions, the text of the Bible was used as reference. After defining the text to use, the main hurdle for the testing of the application was related to the selection of significant values for the three parameters at user's disposal: the pattern, the granularity and the number of streams to launch in parallel.

Initially, a simple classification of the solutions was used to identify the number of streams. For the naive versions, only one stream was launched, while, for the optimized ones, their number was based on the CUDA_DEVICE_MAX_CONNECTIONS environment variable, which defines the number of compute and copy engine concurrent connections (work queues). The default value for this variable is eight and so it is the amount of streams that were chosen to be launched. After this step, a constraint on the string interval's length was researched. To find a suitable value, a simple pattern was used to measure the average execution time of each algorithm, when different granularity values where set up. The g values considered were 100, 200, 500, 1000 and 10000. The result of this measurement test is represented in Figure 4.1.

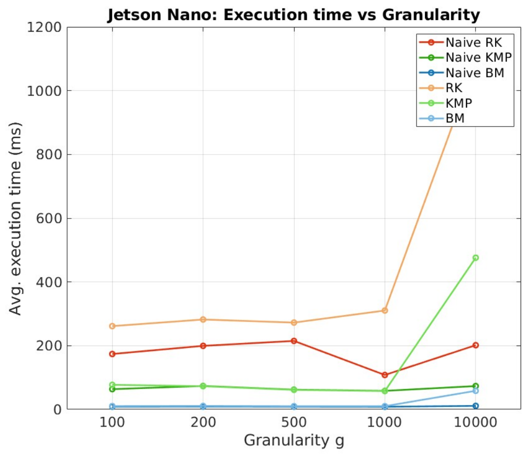


Figure 4.1: Avg. execution time with different g values.

As it can be seen, the time required to complete the scan of the text remained almost stable for lower values of g ($g \leq 1000$), while increasing considerably with higher lengths. For some algorithms, when the granularity is equal to 1000 the average execution time is lowest. This represents an optimal trade-off between the length of the substring to account for by each thread and the sequential properties' utilization of the solutions employed. For this reason, 1000 was chosen as g value for the subsequent tests. Finally, a test was executed to measure the execution times when various patterns, of different lengths, were to be found. The results can be seen in Figure 4.2. On the left figure, the CPU's execution time, the naive implementation's execution time and the optimized solution's execution time can be seen compared for each algorithm. On the right side, the figure shows the total time, required by the GPU, to copy the data in the global memory, launch the kernel and transfer the results to the main memory of the system.

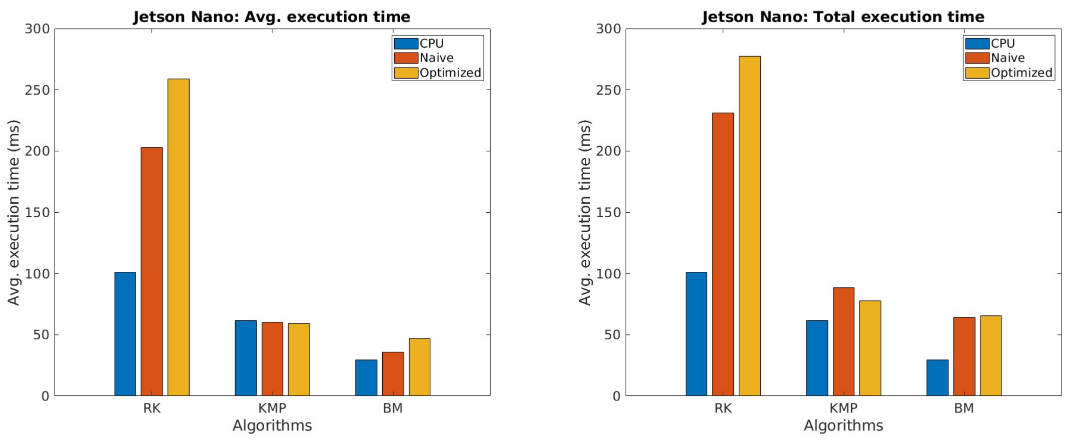


Figure 4.2: Execution times of the algorithms with and without data transfer times.

As can be noted by the figure on the left, the GPU implementations require, at best, the same time interval as the CPU to perform the scan of the text, while, at worst, 2.5 times the amount. This is due to the "Divide and Conquer" strategy being a poor match for the Jetson Nano hardware. Due to the board having only one streaming multiprocessor, virtual blocks are limited to a sequential execution. Also, due to the core having only 128 processors available, blocks have to splitted in half to be executed, since they contain 256 different threads (value chosen to optimize shared memory usage). Another fact to be noted is that there is a difference between the naive and optimized solutions' execution times. The optimized versions, which were expected to perform better, revealed to have worse performance. This is a consequence of the adoption of CUDA streams, which created overheads that impacted both the launch and execution of the kernels. On the contrary, their usage allowed, at the same time, to reduce the time taken for the data transfers. This can be seen on the right figure, where the naive implementations reach their optimized versions, when data transfer times are taken into account. The time increase for the optimized kernels is also due to the copy operations from the global memory to the shared memory of the pattern's information and the synchronization operations.

	Global memory accesses (naive)	**Global memory accesses (opt)**	**Accesses reduction (%)**	**Shared memory accesses (opt)**
Rabin-Karp	$\sim 4.05M$	$\sim 3.96M$	2.21%	$\sim 325K$
KMP	$\sim 1.16M$	$\sim 558K$	52.07%	$\sim 21K$
Boyer-Moore	$\sim 473K$	$\sim 98K$	79.38%	$\sim 5.7K$

Table 4.1: Memory accesses for naive and optimized GPU implementations on the Jetson Nano.

The same test was launched on top of the RTX 3070 and the results are shown in Figure 4.3. Now, the strategy is better exploited, as can be seen by the reduction of the height of the bars related to the naive and optimized GPU implementations. Still, the optimized solutions have an higher execution time than the non-optimized ones, even when considering the data transfer times. Thus, the usage

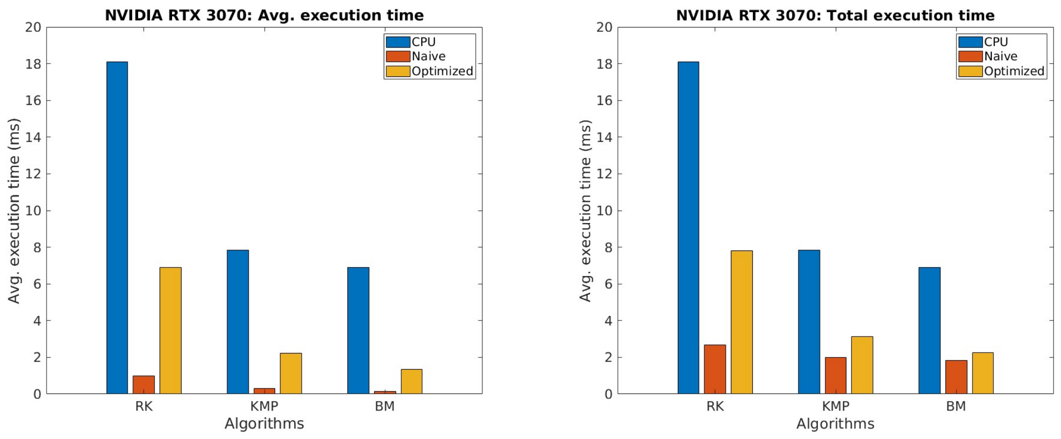


Figure 4.3: Execution times of the algorithms with and without data transfer times.

of streams has to be limited to multi-pattern searches inside the text, when CPU times increase proportionally to the number of text's scan. In this situation, streams allow to search concurrently for different patterns inside the same text, therefore requiring less time to produce results. A test to search five patterns was performed on top of the CPU and GPU of the Jetson Nano and its results can be seen in Figure 4.4.

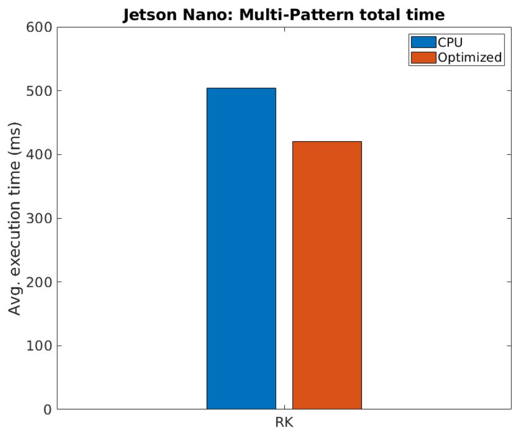


Figure 4.4: Total time of the Rabin-Karp for searching five patterns.

An additional evaluation can be performed on the memory accesses of the kernels. By observing Table 4.1, where load accesses to the memories are registered, it can be seen that using the shared memory helped reducing accesses to the high latency memory up to 80%. This reduction is mostly dependant on the algorithm. The Rabin-Karp is based mostly on hashing the text's substring and this explains the small reduction percentage, while KMP and Boyer-Moore, which are based on skipping as many characters as possible, are characterized by an higher reduction rate. This reduction, though, is impacted by the fact that each block has to retrieve the data and to place it inside the shared memory.

CHAPTER 5

Conclusions

The problem of string matching is an ever present one and it needs constant research to find better strategies to reduce latency and improve throughput, due to how frequent computing systems have to perform scans of databases, texts, websites and many other sources of information to provide users with the correct results. One alternative to the algorithms' enhancement is offered by hardware devices called GPU. These modules offer great parallelization opportunities by having thousands of cores which can work concurrently.

The purpose of the project was to demonstrate how GPUs could obtain higher performance for this specific problem. Three state-of-the-art algorithms (Rabin-Karp, KMP, Boyer-Moore) were implemented as kernels using the CUDA library and evaluated on two different platforms: the NVIDIA Jetson Nano and the NVIDIA RTX 3070. Due to their sequential nature, algorithms were tricky to parallelize and this caused the use of a more straightforward approach, which required to divide the problem of the scan between different threads. However, this method had poor results on the Jetson Nano device, due to its low number of concurrent units, while it showed up to 9 times the reduction of execution time on the RTX 3070 device. Optimizing the algorithms showed that using streams, for this particular problem, causes more disadvantages due to the overheads of using such structures. Therefore, it is advised to make use of them only when multiple pattern have to be found. The usage of the shared memory, instead, is greatly advised to allow faster accesses for frequently-accessed information, as for the pattern's data.

Bibliography

[1] `https://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Memory.svg/853px-Memory.svg.png`

[2] Abdelrahman, Ahmed & Fouad, Mohamed & Salama, Gouda & Dahshan, Hisham. (2018). Enhancing the Actual Throughput of the AES Algorithm on the Pascal GPU Architecture. 10.1109/ICSRS.2018.8688724.

[3] `https://developer.nvidia.com/cuda-toolkit`

[4] `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`

[5] `https://www.nvidia.com/it-it/geforce/graphics-cards/30-series/rtx-3070-3070ti`

[6] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," in IBM Journal of Research and Development, vol. 31, no. 2, pp. 249-260, March 1987, doi: 10.1147/rd.312.0249.

[7] Knuth, Donald Ervin, James H. Morris and Vaughan R. Pratt. "Fast Pattern Matching in Strings." SIAM J. Comput. 6 (1977): 323-350.

[8] Boyer, Robert S. and Strother Moore. "A fast string searching algorithm." Commun. ACM 20 (1977): 762-772.

[9] `https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching`

[10] Saman Ashkiani, Nina Amenta, and John D. Owens. 2016. Parallel Approaches to the String Matching Problem on the GPU. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16). Association for Computing Machinery, New York, NY, USA, 275–285. https://doi.org/10.1145/2935764.2935800

[11] Simone Faro et al. "The String Matching Algorithms Research Tool". In: Proceedings of the Prague tringology Conference 2016. Ed. by Jan Holub and Jan Žďárek. Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111. ISBN: 978-80-01-05996-8.