Parallel L-System

Alfredo Russo - 967771

October 19, 2023

Abstract

This work is focused on extends the classic L-system formalism through the use of the GPU enabling concurrent execution of its components. In the following sections it is possible to find an introduction of the used approach, its importance and the methodologies used in order to implement it. The solution found is able to work with every kind of L-system except for stochastic and context-sensitive one. The findings illustrate that the parallel L-system has the capacity to enhance its efficiency and performance, along with reducing execution time, as the system's size grows.

1 Introduction

The L-system was invented by a biologist whose name is Aristid Lindenmaye with the purpose of describing the behavior of simple multicellular organisms, then extended to the grown behavior of the plants. Nowadays this system is used in different areas starting from biology through computer graphics, procedural content generation and architecture. A significant approach was shown in the work of Yoav I. H. Parish and Pascal Muller [1] where the L-system was particularly useful for the creation of complex and realistic urban areas and structures. As it is possible to imagine, the more complex and articulated the system is, the more time will be required to generate the result, so parallel implementation represents a significant step forward.

An L-system is a mathematical or formal model which operates on the symbol strings manipulation using production rules. This rules describes how the symbols inside the string have to be substituted with other symbols at each iteration of the system. Thus an L-system can be also defined as a rewriting system which is composed of two phases, derivation and interpretation. It will be explored only how to parallelize the derivation phase, examining how it can lead to performance improvements. In order to do that it will exploit the intrinsic parallelization ability of the derivation phase itself, thus the rewriting process which is an operation focused on a single symbol, therefore it doesn't require intra-threads communication and so, any synchronization mechanism. The same cannot be said about the result obtained at every iteration since it has to be concatenated, creating a dependency between threads.

2 L-System Formalism

An L-system can be defined as a tuple $G = (\Sigma, \theta, \Pi)$, where:

- Σ represents the set of the symbols related to an L-system. They can represents different kind of elements and actions.
- θ represents the axiom, the starting string or the initial state where the system starts to evolve itself.
- Π represents the production rules, a set of rules which describes how a symbol inside Σ has to be rewritten. Every production rules is in the form of $A \rightarrow B$ where A is a symbol to substitute and B is the set of symbols with which A must be replaced.

Furthermore there are different kind of L-system that can be classified as follow:

- **Context-free**: in this kind of L-system a rule refers only to an individual symbol and not to its neighbours.
- **Context-sensitive**: where a rule depends not only on a single symbol but also on its neighbours.
- **Deterministic**: there is exactly one production for each symbol
- **Stochastic**: there are several rules, each chosen with a certain probability in each iteration.

As previously said an L-system is composed by two phases, derivation and interpretation one. In the derivation phase starting from the axiom and according to the number of iteration chosen, the result is computed substituting every symbols with the corresponding set of symbols specified in the production rules. Let's assume we have the following L-System $G = (\Sigma, \theta, \Pi)$:

- $\Sigma = (A, B)$
- $\theta = A$
- $\Pi = (A \rightarrow AB, B \rightarrow BA)$

The result of the L-system with 3 iteration is:

- **Iteration 0:** A
- **Iteration 1:** AB
- **Iteration 2:** ABBBA
- **Iteration 3:** ABBBABBAABBBA...

In the interpretation phase instead, every symbol represents a command to a turtle that following them can lead to have different kind of graphical results. An example of this can be seen in the figure 1.

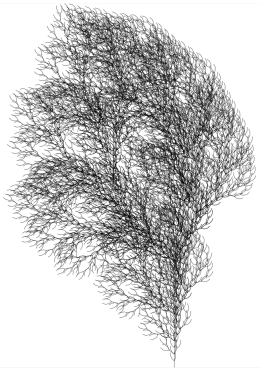

Figure 1: Interpretation of an L-system result (Pitaevskii tree).

3 L-Systems in Parallel

In order to paralllize the computation of the L-system is used the CUDA framework developed by NVIDIA Corporaiton. The CUDA framework allow us to exploit the power of the graphics processing unit (GPUs) in order to accelerate the computation of the L-system.

During my work, I explored two distinct approaches. In the first approach, each symbol of the initial string is assigned to each thread at every iteration, while in the second approach, at each thread is assigned a module composed of a series of symbols. These two types of approaches will be seen in detail later, see 3.2 and 3.3.

3.1 Setup

To initiate the parallel execution of the L-system, the initial step involves configuring the GPU to handle all the production rule-related information. Initially, these rules are prepared on the CPU before being transferred to the GPU. On the CPU side, representing the production rules is a straightforward task, achieved by using a map with keys of type char and values of type string. However, on the GPU side, the process is notably more complex.

To represent this information on the GPU, three arrays are employed. The first array stores characters and contains all the keys associated to a production rules. The second array is an array of character pointers, where each pointer directing to an array of characters that represents the string corresponding to a single production rule. To ascertain the length of a production rule's string, yet another array is utilized – an integer array, where each integer signifies the length of a production rule's value. It is possible to see a graphical representation in the figure 2.

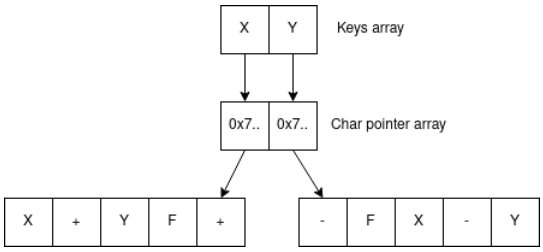

Figure 2: GPU representation of a production rule.

3.2 First approach

To provide a clearer explanation of the approach, I will illustrate how it operates within an iteration. In this method, during each iteration, each thread is tasked with handling a single character of the string. This process unfolds in three distinct phases, see figure 3:

- **Count Phase**: This initial phase serves the purpose of determining the amount of characters that will replace the current one. To achieve this, a kernel is launched, with n threads where n corresponds to the number of characters. Each thread computes its result, which is subsequently stored within an array in global memory.
- **Prefix-scan**: The objective is to generate an array of integers that serves as

offset positions, which will be utilized in the subsequent phase. This array is crucial for informing each thread about the starting position for their rewriting tasks. In order to achieve this results the **thrust library** is used and the results computed.

- **Rewriting Phase**: During this stage, each thread is responsible for a single character and carries out the rewriting process using the offset information computed in the preceding phase. Furthermore, the kernel is initiated with a total of n threads, matching the number of parameters in the input string. If there is no production rule available for a particular symbol, it is rewritten in the resulting output with no changes.

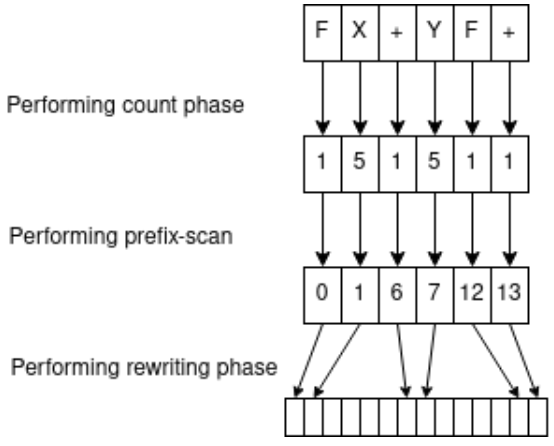

Figure 3: Graphical approach representation.

3.3 Second approach

The earlier method, which involves assigning a single character to each thread, yields noteworthy outcomes. However, this approach results in relatively low workloads for each thread, leading to increased overhead associated with global memory access and threads management.

To enhance the workload of threads, the second approach involves assigning a sequence of characters to each thread. This approach is divided into the same three phases presented earlier with certain modifications in alignment with the new methodology employed. The phases are the following, see figure 4:

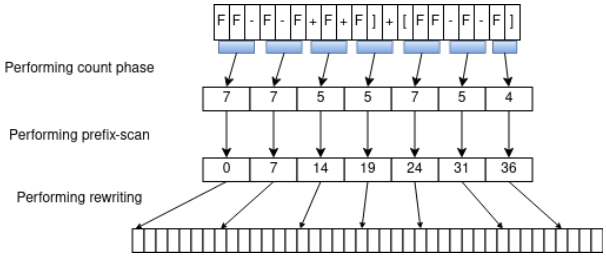

Figure 4: Graphical approach representation.

- **Count Phase**: During this phase, the computation of the characters number to replace the ones in the character sequence is carried out. A kernel is launched with m threads, where m corresponds to the number of sequences. As a result, computations are performed for each of them, and the required number of characters for each sequence is determined and stored in an array within global memory.

- **Prefix-scan**: This phase closely resembles the one in the previous approach, as both the input and the expected output remain the same, and the **thrust library** is employed for this computation as well.

- **Rewriting Phase**: During this stage, the rewriting process unfolds, with each thread responsible for computing results for the characters within the sequence assigned to it. These results are then written to the global memory array, exploiting the offset information derived from the previous phase. Notably, in this instance, the offset information pertains to each sequence rather than individual characters. The kernel is launched with m threads, matching the number of modules requiring rewriting. Similar to the previous approach, if a character is absent from the production rules, it is directly copied to the result array without any alterations.

4 Results

In this section, I will present the results achieved through the parallel algorithm for the L-system, comparing the two approaches against each other and against the results obtained from the CPU implementation. All these results were obtained using a hardware configuration that includes a Ryzen 9 5900HX processor, an RTX 3070 mobile GPU, and 16GB of RAM.

The L-system used in order to test the GPU implementation are directly chosen from Prusinkiewicz and Lindenmayer book untitled the algorithmic beauty of plants [2].

Within the table 1, you can readily discern the attained results while making a comparative analysis of the execution performance between the CPU implementation and the two distinct GPU implementations. This table allows for a comprehensive exploration of the recorded outcomes and their respective computational efficiencies on different processing units.

First approach results indicate a tangible enhancement in execution time, with a significant overall speedup of approximately 5x compared to the CPU execution. Furthermore, the second approach demonstrates an even more impressive speedup, which varies depending on the specific type of L-system being computed, see figure 5.

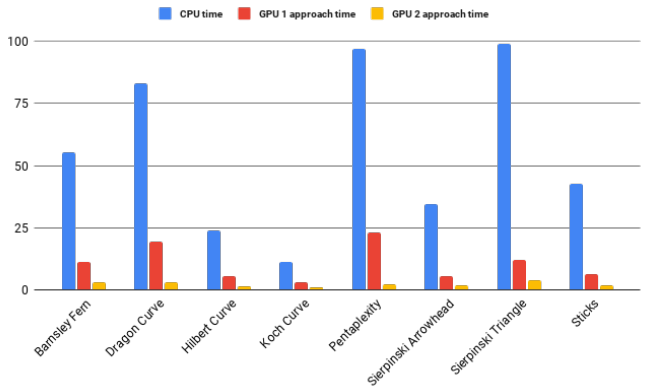

Figure 5: Time comparison.

In contrast to the initial approach, the second approach demonstrates a significant

Name, i	*CPU time* (s)	*GPU time 1 approach* (s)	*GPU time 2 approach* (s)
Barnsley Fern, 14	55.329	11.240	2.873
Dragon Curve, 28	83.137	19.262	3.235
Hilbert Curve, 13	23.780	5.699	1.571
Koch Curve, 12	11.272	2.887	0.835
Pentaplexity, 11	97.093	22.945	2.367
Sierpinski Arrowhead, 18	34.394	5.714	1.701
Sierpinski Triangle, 18	99.297	12.227	3.977
Sticks, 17	42.837	6.373	1.913

Table 1: The table contains all the necessary information for comprehending the results. Adjacent to each name, you can find the iteration number denoted with i, corresponding to the l-system for which the results were calculated. All time values are expressed in seconds.

speedup ranging from 13 times to 41 times, depending on the specific L-system for which the results are calculated. Notably, the L-system representing the Sierpinski Triangle and Pentaplexity one stands out as the most computationally intensive and time-consuming. It is precisely under these circumstances that this approach achieves its optimal results and capitalizes on the benefits of parallel execution.

The results obtained align precisely with our expectations and pertain to the L-system of a specific size. When dealing with a relatively smaller L-system, it is not advisable to compute the results using a GPU due to the overhead it introduces, which results in less efficient execution than the CPU one, see figure 6.

Observing the results, it becomes evident that the CPU implementation outperforms both GPU approaches in terms of execution time. Notably, the first GPU approach exhibits the poorest performance, particularly

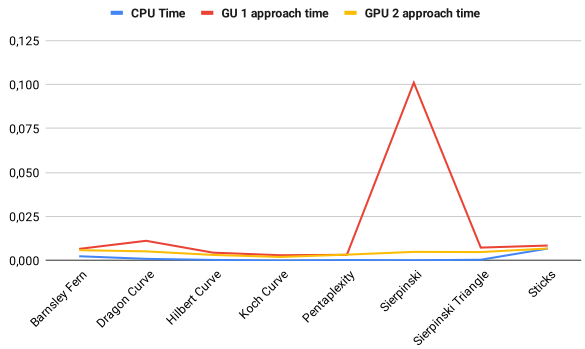

Figure 6: Small L-system time comparison.

when dealing with the Sierpinski Arrowhead L-system. In contrast, the second approach closely rivals the CPU in terms of execution time.

To gain a comprehensive understanding of the two approaches and their optimization, I opted to utilize NVIDIA Nsight Compute profiling tools.

The initial focus was on assessing kernel execution occupancy. To accomplish this, I

employed two distinct L-systems: the Barnsley Fern and the Pentaplexity, both of which yielded identical results. It's crucial to note that the selection of the number of threads per block is not entirely arbitrary. In the first approach, each thread is responsible for processing a single character, while in the second approach, it handles a module, which consists of a sequence of characters. As a result, the number of threads is directly tied to the number of characters and modules, respectively. Furthermore, it's worth keeping in mind that most L-systems start with relatively short initial strings, making it challenging to achieve high occupancy initially. However, this situation evolves over the course of iterations and ultimately yields outstanding results in the final iteration of the first approach, with occupancy surpassing the theoretical limit, as depicted in the figure 7.

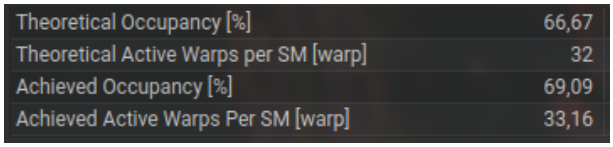

Figure 7: Occupancy results of the first approach related to Barnsley Fern L-system.

In contrast, the second approach does not achieve the same level of performance, as it employs fewer threads for the same number of character within an iteration. This is because, in the second approach, each thread is tasked with handling a sequence of characters rather than just a single character, however the occupancy is still good compared to the theoretical one, see figure 8.

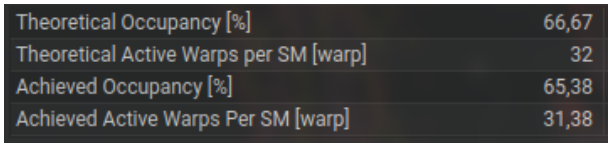

Figure 8: Occupancy results of the second approach related to Barnsley Fern L-system.

Another valuable metric for evaluating the effectiveness of the two approaches is memory throughput. Unlike occupancy, which improves as the system scales up, memory throughput behaves in the opposite manner, as one might expect. During the initial iterations of the system, the volume of data that needs to be transferred to the GPU and subsequently accessed is quite minimal. However, this situation changes as the system grows in complexity. Consequently, in the later iterations, memory throughput becomes a significant challenge for both of the two approaches. Nonetheless, the first approach experiences a more severe impact than the second approach, as illustrated in the figure 9 and 10.

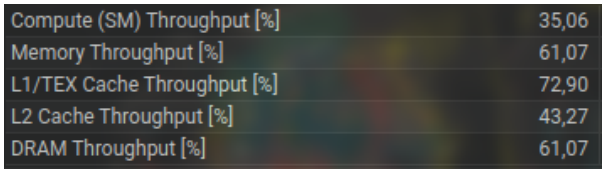

Figure 9: Memory throughput of the first approach related to Barnsley Fern L-system.

One potential solution to mitigate this challenge is kernel fusion. As mentioned in the preceding paragraph, the execution at

each iteration is divided into three distinct parts. However, it's important to acknowledge that the problem cannot be entirely eliminated, as it is intrinsic to the nature of the L-system. Each iteration is parallelized, and the execution of the system remains fundamentally disjoint from one iteration to the next. This characteristic also affects memory usage.

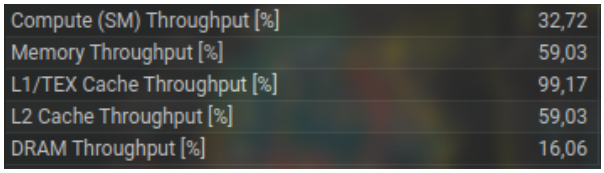

Figure 10: Memory throughput of the second approach related to Barnsley Fern L-system.

5 Future Improvements

This marks my initial endeavor to introduce CUDA parallelization into the L-system derivation phase execution, and there are opportunities for refinement in this endeavor. The foremost enhancement involves expanding the current implementation to encompass support for stochastic and context-sensitive L-systems.

Furthermore, we can elevate the efficiency of my work by parallelizing the L-system interpretation phase. As previously mentioned, L-system results are transformed into commands for a turtle. Given that, this method can be somewhat sluggish, especially when dealing with large L-systems. I've devised a mechanism within my work that generates SVG files. However, parallelizing the interpretation phase is more intricate due to its inherently sequential nature. Nonetheless, there exists a method to accomplish this task. It involves assigning a matrix to each symbol, which represents either a translation or rotation matrix, based on the identified symbol. These matrices can then be concatenated to form a resultant matrix. When this matrix is applied to the origin, it yields the desired endpoint.

References

[1] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 301–308, New York, NY, USA, 2001. Association for Computing Machinery.

[2] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. The algorithmic beauty of plants. In *The Virtual Laboratory*, 1990.