Cover page

Exam information
NDAB12006E - Bachelorprojekt i datalogi (Lotte og Ulrik)

Handed in by
Ulrik Stuhr Larsen
tvr168@alumni.ku.dk

Lotte Maria Bruun
xts194@alumni.ku.dk

Exam administrators
DIKU Eksamen
uddannelse@diku.dk

Assessors
Troels Henriksen
Examiner
athas@di.ku.dk
☎ +4535335718

Patrick Bahr
Co-examiner
paba@itu.dk

Hand-in information

Titel, engelsk: A Language for Parallel Generation of L-Systems
Tro og love-erklæring: Yes

UNIVERSITY OF COPENHAGEN

Bachelor thesis at University of Copenhagen

A Language for Parallel Generation of L-Systems

Lotte Maria Bruun

Ulrik Stuhr Larsen

Supervised by Troels Henriksen

Juni 2020

Lotte Maria Bruun
Ulrik Stuhr Larsen

A Language for Parallel Generation of L-Systems
Bachelor thesis at University of Copenhagen, Juni 2020
Supervisor: Troels Henriksen

University of Copenhagen
Faculty of Science
Bachelor Degree in Computer Science
Universitetsparken 1
2100 Copenhagen

Abstract

This thesis describes the theory of L-systems and how their derivations and visual interpretations can leverage a parallel computing architecture to achieve very high performance. We introduce the theory of L-systems as well as a description of the algorithms for parallel generation and visualisation of L-systems presented by the article *Parallel Generation of L-Systems*(Lipp *et al.*, 2009).

We present our own domain specific language for L-systems. The syntax supports the definition of deterministic, context-free L-systems which can be bracketed or non-bracketed. The language is implemented by a compiler whose target is to produce highly parallel programs in the Futhark programming language. The strengths and weaknesses of Futhark is discussed in the *Methods and Materials* chapter. The compiler can generate code for our four parallel algorithms for derivation and our two parallel algorithms for visual interpretation. The strategies are benchmarked with multiple L-systems and different iteration numbers with 10 runs per test setting. The results show that our fastest derivation strategy can generate up to 17695 symbols per microsecond (17.7 billion symbols per second) and the visual interpretation algorithms can interpret up to 113 symbols per microsecond (113 million symbols per second) for our test cases.

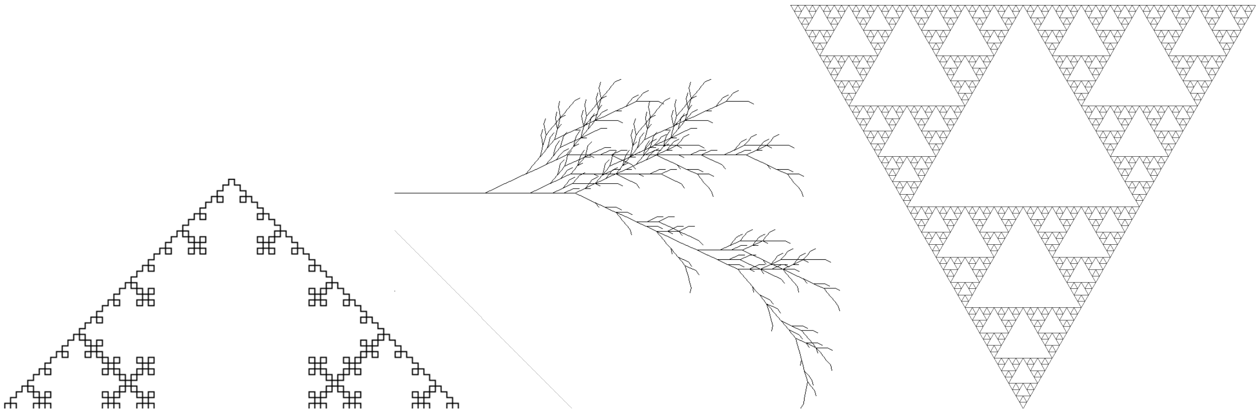
Contents

1 Introduction

Lindenmayer systems (abbreviated L-systems) are formal grammars and rewriting systems for generating geometric structures. They were originally created by biologist Aristid Lindenmayer to simulate plant growth (Prusinkiewicz and Lindenmayer, 2004). However, they are also immensely useful for producing other kinds of self-similar patterns, also known as fractals. L-systems can describe the structure of a fractal as an axiom, an alphabet of symbols and a set of productions using those symbols. The axiom is an initial string which is used as the basis for the L-system. Each symbol in the axiom is rewritten using the L-system's productions to get a new string. This new string is again rewritten and so on for the number of iterations decided by the user. The formal description of L-systems is covered in section 2.1.

However, a string describing a pattern or a simulation of plant growth is not especially interesting alone. A string of an L-system can also be represented graphically as seen in Figure 1.1. Each symbol in the grammar is given a visual interpretation so an arbitrary string created by an L-system can be converted to a picture. Visual representation is covered in section 2.1.3.

So far we have only mentioned the theoretical aspects of L-systems but to use them in an efficient way, an implementation is needed. As it turns out, a naive sequential



(a) Koch curve (4 iterations)

(b) Fractal plant (5 iterations)

(c) Sierpinski triangle (6 iterations)

Figure 1.1: Examples of visual representations based on L-systems. They are produced and drawn by our parallel implementation.

implementation of L-systems quickly becomes infeasible to run when increasing the iteration number. Consider a system which has an axiom "*A*" and *A* is rewritten to *AA* in each iteration. As every symbol is is rewritten to two symbols, the resulting string will double its size and so the string will grow exponentially and have 2^n symbols after n iterations. Many L-systems have an exponential growth as it is common to rewrite a symbol to multiple symbols. We cannot eliminate the property that L-systems grow exponentially but we can try to handle the data in an efficient way. The rewrite of each symbol in a string is independent of the other rewrites, so an iteration on a string can be done in parallel. Parallel derivation is covered in section 2.2.2. The visual interpretation can be done in parallel as well and that is covered in section 2.2.3.

We would like to note that it is also possible to represent some exponentially growing L-systems so they use less than exponential space (St-Amour *et al.*, 2007). However, we have chosen to focus on time efficiency and not space efficiency for our project.

For this thesis, we have created two main works of our own:

1. A language for defining L-systems.

2. A compiler to make GPU parallel programs with two purposes: producing instances of L-systems and visual interpretation of L-systems.

We have also implemented an interpreter for the purpose of testing the correctness of the results of the compiler.

The language makes it possible to define L-systems in a notation close to the theoretical definition of L-systems. This is much easier and more compact than describing L-systems directly as the representations used by parallel programming, which is explained in section 2.3.2. Our implementation is inspired by the article *Parallel Generation of L-Systems* (Lipp *et al.*, 2009), which is described in section 2.2. However, we have made changes to their methods and implemented several methods for comparison of efficiency.

Methods and Materials

2

2.1 Lindenmayer Systems

L-systems are a way of producing fractal-like forms (Santell, 2019). A formal grammar is used to describe the syntax of the fractal. The L-system takes an axiom which is a string of characters from the alphabet of the grammar. It rewrites the axiom by expanding each character with one of the grammar's rules. E.g. if the axiom was *AB* and there was a rule that *A* goes to *B*, and *B* to *AB*, then a new string *BAB* would be formed. The L-system can rewrite this new string and so on in an iterative process to produce new strings. The string after each iteration gives a fractal-like form when each symbol is given a visual representation. Theoretically, the iterative process could be infinite which would indeed produce a fractal but of course that is not practically possible. Thus, the L-system needs an iteration number. This also means that an L-system cannot practically produce a fractal as fractals are infinitely deep and thus the L-system would need an infinite number of iterations. However, for the sake of text simplicity, we will call a fractal-like form a fractal for the remainder of the thesis.

2.1.1 Formal Grammars for Fractals

A formal grammar is a description of how to produce valid words in a language. A grammar consists of an alphabet of symbols and a number of rules that are called *productions*. Productions contain only the symbols from the grammar's alphabet. An example of a rule is:

$$A \rightarrow BC$$

This means that applying the rule on *A*, you get the string *BC*. The application of productions is called *transformation*, *derivation* or *rewriting*. You can talk about both a transformation of a single symbol and a transformation of a whole string, where a production is applied to every symbol in the string. A symbol can also be called a character.

In L-systems, the left-hand side of the arrow is called the *predecessor* and the right-hand side is called the *successor* (Santell, 2019). The production describes a possible

$n = 1$	$n = 3$
FL	*FL*
FL+RF+	*FL+RF+*
	FFL+RF++-FL-RF+
	FL+RF++-FL-RF++-FL+RF+--FL-RF+

Table 2.1: Producing strings for dragon curves with n iterations.

transformation of the predecessor, here *A*, to the successor, *BC*. The successor can have as few or as many characters as needed or even none, but the predecessor can contain only one character that is rewritten by that production. The predecessor can also contain other properties like conditions for application which are explained in section 2.1.2.

As example we can look at the grammar of a *dragon curve* (Prusinkiewicz and Lindenmayer, 2004):

$$\begin{aligned} \text{Axiom: } & FL \\ \text{Productions: } & L \rightarrow L + RF + \\ & R \rightarrow -FL - R \end{aligned}$$

Here the characters *F*, + and - are constants which means that they are not transformed when rewriting the string. Thus they do not need a production. In Table 2.1, the L-system iterations, *n*, of the dragon curve can be seen for $n = 1$ and $n = 3$.

2.1.2 Types of L-Systems

There are several kinds of L-systems where a few will be explained here.

An L-system can be either *context-free* or *context-sensitive*. It is *context-sensitive* if the grammar contains productions that can only be used if the predecessor is preceded or succeeded by a specific symbol (Santell, 2019). This contrasts *context-free* L-systems which are characterized by having no predecessors that uses context. An example of a context-free production is $C \rightarrow AB$ which transforms C into AB. When writing context-sensitive productions, we need a notation for marking what is context and what is the symbol in being transformed. The notation $X < Y$ means X precedes Y, $Y > X$ means X succeeds Y and $X < Y > Z$ means Y is preceded by X and succeeded by Z. In these examples of predecessors, X and Z are the context and Y is the symbol being transformed. An example of a context-sensitive production is $A < C > B \rightarrow AB$ where C is transformed to AB if it is preceded by a A and succeeded by a B.

Another type is *stochastic L-systems* (Santell, 2019). They contain several productions for each predecessor where a random production is chosen based on the probability for each production. That means that each production for a specific predecessor will be given a probability of how likely it is that it is chosen for application. E.g. we could define an L-system with the productions $A \xrightarrow{0.4} AB$, $A \xrightarrow{0.4} A$ and $A \xrightarrow{0.2} B$. Here A has a 40% chance of being derived to AB, a 40% chance of A and 20% chance of B. Stochastic L-systems are useful for e.g. producing plant fractals because there is some randomness in the growth of a plant. If an L-system is not stochastic, it is *deterministic* which means that there can be at most one applicable rule to each symbol.

An L-system can also be *parametric* (Prusinkiewicz and Lindenmayer, 2004). Each symbol can have a list of *parameters* associated with it where a parameter is a real number. A symbol with a parameter list is called a *module* and a combination of parametric symbols is a *series of modules*. The values of a parameter list is set by the production constructing its module. Optionally conditions on the parameter values can be used to choose between multiple productions for a symbol. An example of a parametric production is $a(i, j) : i < 0 \rightarrow a(1, j + 1)bc$ which matches on the production only if the first parameter of a is negative. If we have the string $a(2, 3)a(-7, 5)$, then the first module $a(2, 3)$ cannot be transformed by the rule as $2 \not< 0$ while module $a(-7, 5)$ satisfies the condition by $-7 < 0$. Thus the transformed string in this case is $a(2, 3)a(1, 6)bc$ assuming there are no applicable productions to $a(2, 3)$.

The last type we will consider is *bracketed L-systems* (Santell, 2019). It is a bit different than the other discussed types as it is does not affect the derivation of an L-system. Instead it is useful for visualisation. However, it should be noted that when combined with context-sensitivity, the derivation will become more complex which is described in section 2.2.3.2. We will return to bracketed L-systems in section 2.2.3.2, when the methods for visualisation has been covered.

All of the types can be mixed and matched as needed. Even context-free and context-sensitive can mixed in the sense that an L-system can contain both context-free and context-sensitive productions. However, the L-system will be considered context-sensitive as a whole if it has any context-sensitive rules.

For this project, we have chosen to implement context-free, bracketed L-systems as a sample of the full range of L-systems. However, we will discuss how our implementation can be extended to cover all of the types of L-systems in section 5.

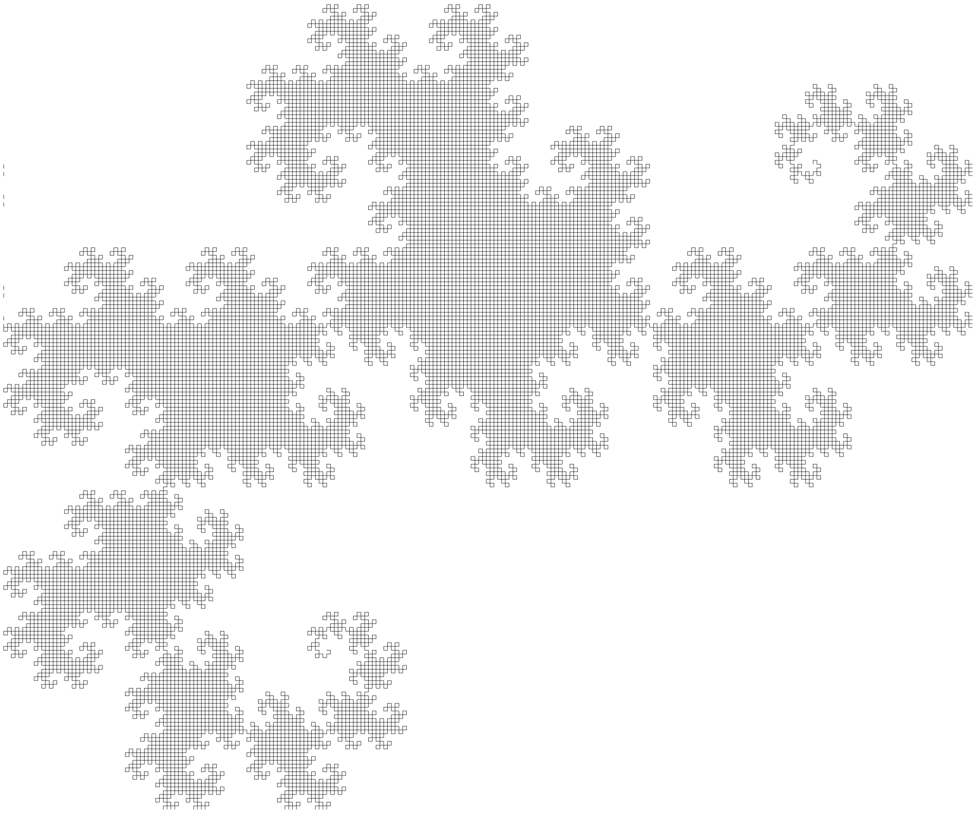


Figure 2.1: Dragon curve with 15 iterations produced and rendered by our parallel implementation.

2.1.3 Visualisation of L-Systems

An L-system only produces a string representation of the fractal. However, fractals are typically represented visually. We can use turtle graphics to visualise the string representation of a fractal. The “turtle” is a cursor on an empty canvas. The turtle has a direction and a position on the canvas. The turtle can do each of the following possible actions: *turn*, *move* or *draw*. *Turn* turns the turtle a number of degrees, *move* moves the turtle a number of steps along the turtle direction and *draw* is a move where the turtle draws a line between the starting point and the end point. Turtle graphics can be used by giving each character in the grammar an action. As an example, we use at the dragon curve again and give the characters the following turtle actions (Prusinkiewicz and Lindenmayer, 2004, p. 11):

$$\begin{aligned} F:&\quad \text{draw}\\ -:&\quad \text{turn left } 90^\circ\\ +:&\quad \text{turn right } 90^\circ\\ L:&\quad \text{no action}\\ R:&\quad \text{no action} \end{aligned}$$

With this interpretation list, we can convert a dragon curve string to a list of turtle commands and perform each command in the list to produce a picture. When using this graphical interpretation, the dragon curve looks as in Figure 2.1.

As promised, bracketed L-systems will now be covered. Bracketed L-systems are used for creating branches, where a branch is started by [and is ended by]. When the visualisation of the branch has ended and it has reached the end bracket], it will go back to the turtle position before the branch. This means that multiple parts of the created fractal use the same starting position. The bracket notation makes it easy to implement nested branching as the starting and ending tags are different. A stack can be used to implement the branching where [pushes a position on the stack and] pops a position and updates the turtle position to the popped position.

2.1.3.1 Turtle Commands as Matrices

We need to have a representation for turtle commands before turtle graphics becomes useful for computation. You can represent the commands with transformation matrices and the turtle state as a matrix as well. The reasoning behind the construction of these matrices are not relevant to the project so we will just present definitions and use them in our implementation. The turtle state, also called current coordinate system, can be represented by (Ju *et al.*, 2004, p. 6):

$$S = \begin{pmatrix} u_1 & u_2 & 0 \\ -u_2 & u_1 & 0 \\ p_1 & p_2 & 1 \end{pmatrix}$$

where (p_1, p_2) is the turtle's position and (u_1, u_2) is a vector describing direction and step size. We need two transformation matrices for turtle commands (Ju *et al.*, 2004, p. 44):

$$L_M(d) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d & 0 & 1 \end{pmatrix}$$

$$L_T() = \begin{pmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$L_M(d)$ moves the turtle d steps and $L_T(\alpha)$ turns the turtle with degree α. Notice that these matrices only work for 2D spaces but they can also be made as 3D. The draw command can be done with L_M where the start and end points for the line is saved. A command is applied by pre-multiplying the appropiate matrix to the turtle state S (Ju *et al.*, 2004, p. 5):

$$S_{new} = L \cdot S$$

2.1.4 Parallelism in L-Systems

When working with context-free L-systems, each transformation of a character is independent of the other transformations in a single iteration. This means that it does not matter in which order each symbol of the string is derived as long as the successors are put together correctly afterwards. Therefore it can be done in parallel. However, to transform the symbols of the next iteration, we have to know which symbols are in that next iteration. The symbols are found by transforming all the symbols from the current iteration and concatenating the results in a single string. Thus the next iteration is dependent on the current iteration and each iteration will be handled sequentially.

However for many L-systems, a lot of threads will be needed. The number of symbols generated in the last iteration will be $O(n \cdot m^i)$ where *n* is the number of characters in the axiom, *m* is the longest successor and *i* is the number of iterations. There will be $n \cdot m^i$ symbols as in the worst case only the longest production is applied so in the first iteration each of the n symbols are rewritten to m successors so the string is $O(nm)$ afterwards. In the second iteration, the nm symbols are rewritten with m symbols each again so there are $nmm = nm^2$ and so on. Notice that *m* is a really loose bound as L-systems only rarely are able to use just the longest successor and successors often contain constant symbols which derive to themselves.

Ideally, each transformation should have its own thread such that all transformations are done in parallel in a single iteration. However in practice, it is faster not to utilise full parallelism and instead divide data into equal chunks that are handled in parallel (Henriksen, 2017, p. 18). This is handled by the programming language, we use for our implementation (Futhark). Thus, there is no need to go into more details with that and we can assume for our asymptotic time analyses later in the report that the programs are fully parallel.

A GPU is able to run thousands of threads in parallel but each thread can only do a small amounts of computations. Each transformation of a character is a very small computation as it only needs to find an appropriate production and transform the character accordingly.

Therefore, it would be appropriate to use a programming language that specialises in GPU parallelism like Futhark.

Interpretation can be done in parallel as well but this is described in section 2.2.3.

2.2 Previous Work

The main inspiration sources for this project is the article *Parallel Generation of L-Systems* (Lipp *et al.*, 2009). It describes a way to use a GPU or a CPU to parallelize L-systems by exploiting the independency of derivations in L-systems. It also describes ways to use parallelism in the visual interpretation of L-systems. We will focus on the GPU approach. Their approaches are made to be efficient on large L-systems and utilize thousands of parallel GPU threads. Be aware that a highly parallel implementation likely will not outperform single-core implementations on small L-systems because of the overhead of starting the GPU. The rest of the section will explain the work presented in the article.

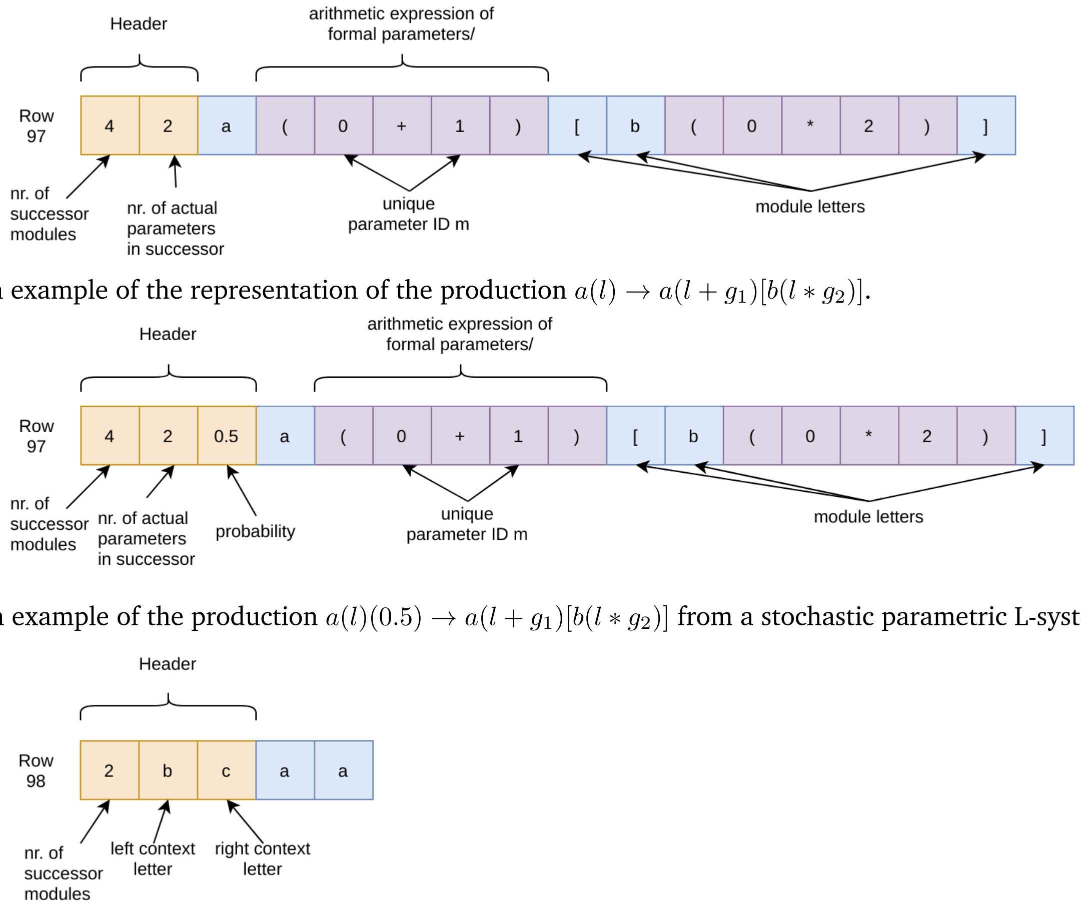
The approach of the article has two phases, derivation and interpretation:

1. Derivation: There is a pre-step to derivation: Representation. Representation is needed since derivation cannot be done by a computer before we have a way to represent the strings and productions as data. Representation is described in section 2.2.1. An efficient, parallel way to derive is of course also essential, and this is described in section 2.2.2.

2. Interpretation: The visualisation is done using turtle graphics. Two parallel concepts are given for this in section 2.2.3.

2.2.1 Representation

This is a description of the runtime representation which is used by the parallel program. Presumably, the authors have implemented the representations directly in CUDA and a CPU language, as opposed to making a DSL for L-systems and compile to a parallel language like we do. The production rules are stored in a 2D array where each successor is stored in the row index by the ASCII value of the predecessor. Collision chains are used to resolve collisions where there are multiple successors for the same predecessor such as in stochastic L-systems. Collision chains solve the problem by taking the elements that are stored in the same slot and put them in a linked list and make the slot point



(a) An example of the representation of the production $a(l) \rightarrow a(l + g_1)[b(l * g_2)]$.

(b) An example of the production $a(l)(0.5) \rightarrow a(l + g_1)[b(l * g_2)]$ from a stochastic parametric L-system.

(c) An example of the production $b < a > c \rightarrow aa$ from a context sensitive L-system.

Figure 2.2: Representations for different types of L-systems (based on Lipp *et al.*, 2009)

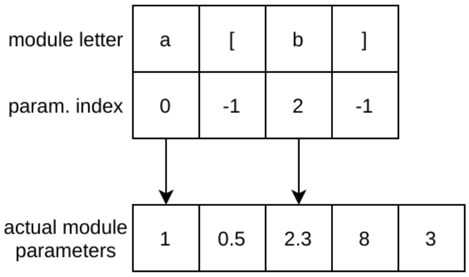


Figure 2.3: An example of a module string representation. In this example, *a* has parameters 1 and 0.5 and *b* has 2.3, 8 and 3. [and] do not have any parameters.

to the list (Cormen *et al.*, 2009, p. 257). Just before any rule is a header containing information on the rule. The specific contents of a rule header depends on the type of L-system being implemented. It is used to hold information on the rules which is useful for deriving symbols when the productions are used. With the different types of L-systems, the contents of the header are:

- All L-systems will have the number of successors in the header.
- Parametric L-systems will hold the number of parameters in the successor.
- Stochastic L-systems will have the probability of the production.
- Context-sensitive L-systems will hold the left and right context letters in the header.

The types of L-systems can be combined. The representations for each basic L-system type is visualized in Figure 2.2.
In parametric L-systems, the parameters are translated into unique numerical IDs to allow $O(1)$ lookup in a table holding the parameter values. The representation of modules uses three arrays. A visualised example is given in Figure 2.3. The first array contains the n letters of the module string. The second array is length n and contains the indices to the parameter values. This is done because the modules can have multiple parameters so the index is an offset into the parameter value array. If a character does not have a parameter, the offset is set to -1. The third array is the array containing the parameter values.

2.2.2 Parallel Derivation

The derivation is done by expanding a string iteratively where each expansion iteration is done in parallel in the GPU. The derivation requires a production array, an axiom and a number of iterations. All the information is uploaded to the GPU. The iterations are performed sequentially but each iteration is performed in parallel. In each iteration the current string is derived. One iteration is handled by three passes. An example of running the passes is visualized in Figure 2.4. The three steps are:

1. Each kernel is launched with n threads where each thread is assigned $m = inputSize/n$ subsequent modules/characters. The first pass counts the total number of successors needed for all symbols assigned to the thread. This is done by a simple lookup in the corresponding production headers. If the L-system is parametric then this step will also count the number of parameters in the same way as it counts the number of successors. This is basically a map which does a look-up in the production array.

2. The second pass does a sum-scan on the result from the first pass. This is used to calculate offsets and the length of the derivation result for when the third pass actually does the rewriting. If the L-system is parametric then it will also do a sum-scan on the parameters. It only requires one scan as symbols and parameters can be included in a single array. The sum result on the parameters is used to calculate offsets into the parameter value array.

3. n threads are launched again. The third pass does the actual rewriting of the string. It fetches the production for each assigned module/character. First it should choose an applicable production and this step depends on the type of L-system:

   - If the L-system is context-sensitive then the preceding and succeeding symbols are compared with the letters in the production header to check if the production can be used.

   - If the L-system is bracketed context-sensitive then finding the context becomes more complicated because of bracketed L-systems' use of a stack. In this kind of system a parallel hierarchy extraction step is needed which will be explained in section 2.2.3.2.

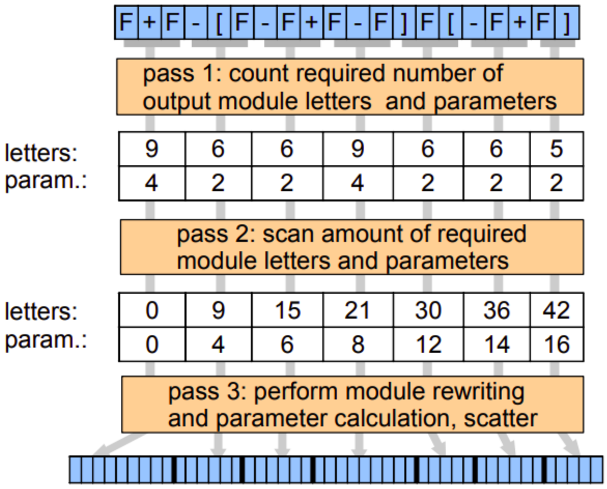

F + F - [F - F + F - F] F [- F + F]
pass 1: count required number of output module letters and parameters
letters:
param.:
9 6 6 9 6 6 5
4 2 2 4 2 2 2
pass 2: scan amount of required module letters and parameters
letters:
param.:
0 9 15 21 30 36 42
0 4 6 8 12 14 16
pass 3: perform module rewriting and parameter calculation, scatter

Figure 2.4: The three passes performed in each iteration. The parameters have been omitted. (Lipp *et al.*, 2009, p. 4).

- If the L-system is stochastic then for every symbol in the string, a random number for each applicable production is chosen. The number is multiplied by the probability in the header and the production with the highest product of probability and random number is used for derivation. The article says that the random number is found by using the position in the string to index into an array with random numbers. However, this does not make sense since it would cause the same random number to be chosen for every production. We think they mean the position of the production in the production array which can be used for indexing into a random array. They would use a new random array every time a production should be chosen so a production will not get the same random number each time it is applicable.

When a production is chosen for application, a thread will insert the successors into a resulting string. If no applicable production is found for a symbol, the symbol will be copied into the resulting string. If the L-system is parametric, it will evaluate the parameters for all successor modules before inserting them. All of the successors are inserted in parallel using the array with successor offsets from the second pass.

An example of derivation using the passes is shown in Figure 2.4. Notice that because each thread handles more than one module, the intermediate result arrays are shorter than the input array.

2.2.3 Parallel Interpretation

Derivation results in a string which can be interpreted to a visual representation. For this, turtle graphics are used (explained in section 2.1.3). 4x4 matrices can be used to implement the turtle states. Each symbol have a turtle command which can be represented by a 4x4 matrix transformation (except push and pop). These transformations are applied to the turtle state to move the turtle for each symbol in the string. Thus a turtle state is dependent on every one of the previous states, so it might appear like it can only be accomplished sequentially. However, this is not the case. The article gives two suggestions of how to use parallelism for interpretation, one for non-branching L-systems and one for branching.

2.2.3.1 Visualisation of Non-Branching L-Systems

Strings of non-branching L-systems are divided into chunks which can be interpreted independently. The interpretation is done in three passes:

1. It divides the string into n chunks and they are each given to a thread. The threads multiply the symbol matrices in their chunk to one transformation matrix. It also counts the number of generated geometry objects, which is the number of draw commands (move and turn does not make geometry). Two arrays are created, one to store the transformation matrices and the other to store geometry amounts.

2. It scans over the transformation matrices with matrix multiplication as scan operator. The starting accumulator is the identity matrix. This results in an array with the turtle state transformations at the beginning of each chunk. Then another scan over the geometry amount array is used to calculate offsets for the geometry.

3. It looks at the n string chunks again and multiply the symbol matrices. The i^{th} chunk is pre-multiplied by the i^{th} transformation matrix from pass 2. Every time there is a geometry generation (a draw command), the object position is computed using the object offset array calculated in pass 2. The positions of objects are stored in a vertex buffer object (VBO).

An example is shown in Figure 2.5.

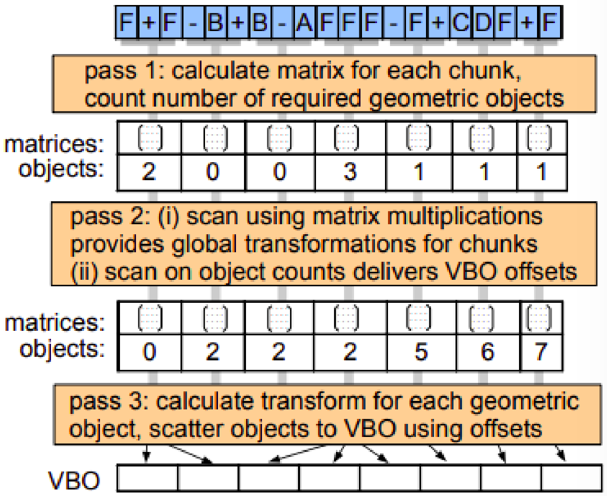

F + F - B + B - A F F F - F + C D F + F
pass 1: calculate matrix for each chunk,
count number of required geometric objects
matrices:
objects:
2
0
0
3
1
1
1
pass 2: (i) scan using matrix multiplications
provides global transformations for chunks
(ii) scan on object counts delivers VBO offsets
matrices:
objects:
0
2
2
2
5
6
7
pass 3: calculate transform for each geometric
object, scatter objects to VBO using offsets
VBO

Figure 2.5: The three passes of visualisation of non-branching L-system (Lipp *et al.*, 2009, p. 5).

2.2.3.2 Visualisation of Branching L-Systems

The interpretation of branched L-systems are not as easy as for non-branched systems since push and pop cannot be applied with matrix transformations. However, the branches also create an opportunity for parallelism since they can be interpreted independently. The algorithm uses parallel hierarchy extraction. A branch hierarchy refers to how deeply a bracket pair is enfolded within other bracket pairs. We also call the bracket pair a push/pop pair as a starting bracket corresponds to a pop and an end bracket corresponds to a push (explained in section 2.1.3). The idea is to extract the positions and depths of the bracket pairs. These are stored in buckets (arrays) where each bucket contains the pairs for a single depth. When you do this, the positions of a pop and its corresponding push will always by next to each other in the bucket because the bucket only contains one depth.

The algorithm contains five passes. It assumes that the maximal depth of the hierarchy is known so the bucket arrays can be allocated before running the passes. It works with chunks which are processed in parallel, just like the other algorithms.

1. First, the depth at the end of a chunk is computed relatively to the depth at the beginning of the chunk. This is done by adding 1 for each push and subtracting 1 for each pop. An array is filled with the results for the chunks.

2. It scans over the array from the pass 1 to get the absolute depths for the start of the chunks.

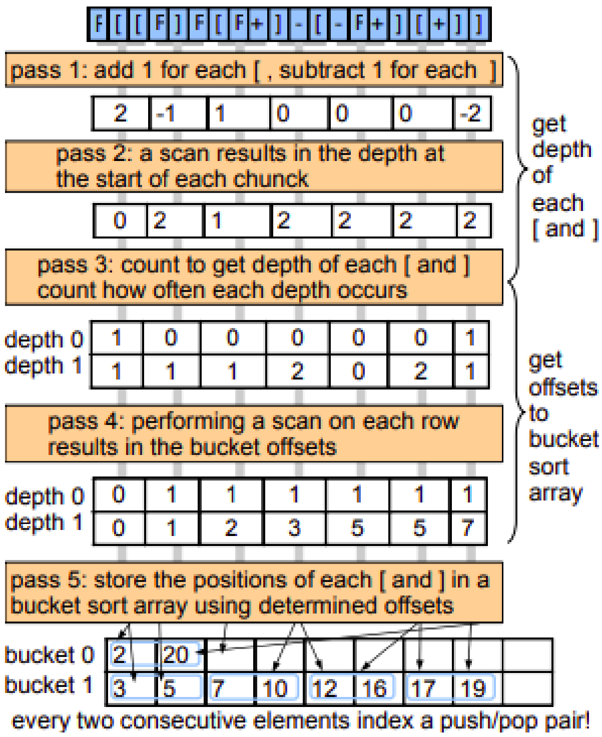

F [[F] F [F +] - [- F +] [+]]
pass 1: add 1 for each [, subtract 1 for each]
2 -1 1 0 0 0 -2
pass 2: a scan results in the depth at the start of each chunck
0 2 1 2 2 2 2
pass 3: count to get depth of each [and] count how often each depth occurs
depth 0 1 0 0 0 0 0 1
depth 1 1 1 1 2 0 2 1
pass 4: performing a scan on each row results in the bucket offsets
depth 0 0 1 1 1 1 1 1
depth 1 0 1 2 3 5 5 7
pass 5: store the positions of each [and] in a bucket sort array using determined offsets
bucket 0 2 20
bucket 1 3 5 7 10 12 16 17 19
every two consecutive elements index a push/pop pair!
get depth of each [and]
get offsets to bucket sort array

Figure 2.6: The passes of visualisation of branching L-systems (Lipp *et al.*, 2009, p. 6).

3. This pass has two jobs. Firstly, it computes the absolute depth for each push/pop pair which can later be used to get an offset for each pair in the final bucket array. Secondly, the pops and pushes at each depth in the chunk are counted and stored in global 2D array. Each row represents a depth and column k is how many times the depth occurs in the k^{th} chunk. This means that when indexing with $[i, j]$, you get the number of times depth i occurs in chunk j.

4. It sum-scans over each row in pass 3 to get the bucket offsets for each chunk.

5. In this pass, the position of each push and pop is stored in the bucket array using the depths and offsets from pass 4. The starting offsets into the bucket array for a chunk c is the c^{th} column of the result from pass 4. A chunk might have multiple pushes/pops at the same level so the chunk needs to keep track of the offset to use. This results in an array where every two consecutive elements in a row correspond to a push/pop pair.

You can see an example in Figure 2.6.

The memory usage of the bucket array can be optimized. The article does not specify how to determine a length for the buckets but we assume that they set it to the length of the string to be sure they are long enough. This is because the largest possible number of push/pop pairs occurs when the string contains only brackets. In that case there is $^n/_2$ push/pop pairs where n is the length of the string. The total number of pushes and pops are therefore n in this case. Before the algorithm, you do not know how many pairs will be in each bucket so every bucket is set to be n long. Hence the 2D bucket array can be very inefficient because a lot of elements will not be filled. To reduce the memory usage, the bucket array can made into a 1D array instead. You can use the sum of the chunk offset and the count of pop and push commands at an index j to map the j^{th} bucket element to its place in the 1D array.

The bucket arrays are used for parallel visualisation. The interpretation uses a parallel work-queue. The idea is that a single thread starts doing the interpretation of the string sequentially and when it finds a push, it creates two new work items, one for each branch. So one thread takes the work inside the push/pop pair and another takes the work after the pop. To do this the bucket array must be integrated into the string by writing the positions of pop positions directly in the string by its corresponding push. The integration is accomplished by first assigning the 1D bucket array to multiple threads. Each odd element (the pops) is given a parameter containing the position to the module referenced by the previous element (its associated push). As an optimization the offsets needed for the geometry can be included in pass 1 and pass 2.

2.3 Parallelism in Futhark

2.3.1 Introduction to Futhark

Futhark is a programming language which is designed for writing efficient parallel code, especially using the GPU. This makes it an ideal choice for implementation of parallel L-systems (cf. section 2.1.4).

When working with Futhark, it is important to keep in mind the language's strengths and weaknesses. We have included some of the peculiarities here.
Firstly, Futhark is not parallel per default. The programmer has to identify sources of parallelism himself and use Futhark's built in parallel functions. We will look into runtimes for a selection of the parallel functions in section 2.3.3.1. This is included to give an intuition for the efficiency of a Futhark program.
Secondly some parallel functions have conditions of use to make them parallel. One that is important for this project is that functions given to `scan` and `reduce` has to be associative, because a parallel solution does not make the computations in order. Be

aware, that mistakes of this kind will not result in compile time errors and will only be visible as wrong results.

Thirdly, arrays must be regular which means that all internal arrays on the same level has the same length. This is done because nested parallelism is "significantly easier to map to hardware"(Henriksen, 2017, p. 23) under the assumption of regularity. Nested parallelism is when a parallel function contains another parallel function and Futhark utilizes all levels of nested parallelism assuming there is enough available GPU power. A programmer can work around the regularity requirement however, by manually flattening irregularities or simply modify it to be regular (Henriksen, 2017, p. 24).

Lastly, every Futhark type cannot be cleanly represented in the target language (Elsman *et al.*, n.d.(b), Basic Usage, sec. 2.2.1). If a program tries to return such a type it will give an error message saying that it cannot show opaque types. An example of a opaque type is arrays of tuples.

2.3.2 DSL or Direct Implementation in Futhark

As mentioned before, we will write a domain specific language (DSL) for L-systems that is close to the formal notation and write a compiler from the DSL to Futhark. Possibly, the reader would think that Futhark sounds perfect for the job so why do we not just implement L-systems directly in Futhark? A DSL seems like an unnecessarily complication on top of a perfectly fine parallel programming language. However, there are a few problems with that.

Firstly in Futhark, the representations of productions are fairly complex and written for the ease of computation, not for the comprehensibility of humans (cf. section 2.2.1). The theoretical notation for L-systems (the grammar notation) is much easier to understand and write for humans without making mistakes. Furthermore, the matrix representation of each turtle command would also have to be made by hand.

Secondly, Futhark does not contain the same basic elements as L-system notation. Futhark does not have chars or strings so all letters in a grammar would have to be converted to numbers and a string would be an array of numbers. Futhark contains a lot of functionality which is not used by L-systems and the notation is much more complicated than what is needed to define an L-system. Thus a user would have use a lot of extra time to familiarize themselves with the workings of Futhark.

2.3.3 Work-Depth Asymptotic Analysis

To make a highly effective parallel program, it is important to establish an intuition of the runtime of a parallel algorithm. A useful tool for this is *work-depth asymptotic analysis* (Oancea, 2018). There are two main concepts (Oancea, 2018, p. 27): The work complexity and the depth (also known as step complexity or span). The work complexity is the number of executed operations when running a program. Depth is the number of required sequential steps when running a program. They are denoted by $W(n)$ and $D(n)$ respectively where n is a measure for the size of the workload. For the sake of simplicity, the computations of work and depth assumes that an infinite amount of processors are available.

A parallel program is work efficient if its asymptotic work is equal to the asymptotic work of the same sequential algorithm (Oancea, 2018, p. 27). The parallel algorithm cannot use less operations than the least amount of needed work and the sequential algorithm will by definition use the least amount of work or it would not be optimal. Thus the asymptotic work of a parallel algorithm can never be smaller than the asymptotic work of the same sequential algorithm.

Work-depth asymptotic analysis does not take the number of cores in the processor into account. However, these are clearly relevant to the runtime of a parallel program. For this we can use *Brent's theorem* (Oancea, 2018, p. 27):

$$\frac{W(n)}{P} \leq T \leq \frac{W(n)}{P} + D(n)$$

Where P is the number of cores and T is the time complexity of the program.

2.3.3.1 Analysis of Parallel Futhark Functions

In this section, we will go through work-depth analyses of a selection of Futhark functions. It is merely a section that illustrates the work/span model, and Futhark's behaviour under that model, and not our own work (analyses reference: Henriksen and Elsman, 2019). We will use these cases as a basis for asymptotic analysis of our Futhark programs for L-systems.

To do work-depth analysis on Futhark programs, we need to some establish some basic cases:

$$
\begin{aligned}
W(v) &= 1 \\
D(v) &= 1 \\
W(e_1 \oplus e_2) &= W(e_1) + W(e_2) + 1 \\
D(e_1 \oplus e_2) &= D(e_1) + D(e_2) + 1 \\
W(\backslash x \texttt{ -> } e) &= 1 \\
D(\backslash x \texttt{ -> } e) &= 1 \\
W(\texttt{[}e_1, \cdots, e_n\texttt{]}) &= W(e_1) + \cdots + W(e_n) + 1 \\
D(\texttt{[}e_1, \cdots, e_n\texttt{]}) &= D(e_1) + \cdots + D(e_n) + 1 \\
W(\texttt{(}e_1, \cdots, e_n\texttt{)}) &= W(e_1) + \cdots + W(e_n) + 1 \\
D(\texttt{(}e_1, \cdots, e_n\texttt{)}) &= D(e_1) + \cdots + D(e_n) + 1
\end{aligned}
$$

where v is a variable, e_i is an expression and $\oplus$ is a basic operation. As stated above, fetching variable values and each basic operation $\oplus$ have the work and depth 1. Of course when applying an operator each argument expression has to be evaluated. Function definitions have work and depth 1 as well as the function body is only evaluated in Futhark when the function is called. The work and depth of making an array or a tuple is 1, plus the evaluation of each element of course. So if each expression in the array or tuple is evaluated in constant time then work and depth is $O(n)$.

Now comes some more complex cases. We will use $[\![e]\!]$ for the result of evaluating expression e. `iota` e makes an array from 0 to x (exclusive x) so its work is the work of evaluating the input expression plus the number of elements (the work of each element is one). The elements are made in parallel so the depth is only the depth of the input expression plus 1 for initializing the array. If the argument is evaluated in constant time, the work is $O(n)$ and the depth is $O(1)$.

$$
\begin{aligned}
W(\texttt{iota } e) &= W(e) + [\![e]\!] \\
D(\texttt{iota } e) &= D(e) + 1
\end{aligned}
$$

`replicate` has the work of evaluating the two arguments and inserting the

$$
\begin{aligned}
W(\texttt{replicate } e_1\ e_2) &= W(e_1) + W(e_2) + [\![e_1]\!] \\
D(\texttt{replicate } e_1\ e_2) &= D(e_1) + D(e_2) + 1
\end{aligned}
$$

The work-depth of `take` is the same as `replicate`'s as they both make an array of length e_1. The difference is that `take` copies elements from an array instead of duplicating a single element as `replicate`.

The work of a let-binding is the work for evaluating the expressions and 1 for making the binding. The depth is the same.

$$W(\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e') = W(e) + W(e'[x \mapsto [\![e]\!]]) + 1$$
$$D(\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e') = D(e) + D(e'[x \mapsto [\![e]\!]]) + 1$$

When applying a function, the work is the work of evaluating the input, the work of evaluating the function body with the input and 1 for the function call.

$$W(e_1\ e_2) = W(e_1) + W(e'[x \mapsto [\![e_2]\!]]) + 1$$
$$D(e_1\ e_2) = D(e_1) + D(e'[x \mapsto [\![e_2]\!]]) + 1$$
$$\text{where } [\![e_1]\!] = \backslash x\ \texttt{->}\ e'$$

The work of map is the work of evaluating the two arguments and applying the function to each element in the array. The depth is the depth of evaluating the arguments and the greatest depth of applying the function to an element in the array. It is the maximum as the function is applied to each element in parallel. 1 is added to depth to start map.

$$W(\texttt{map}\ e_1\ e_2) = W(e_1) + W(e_2) + W(e'[x \mapsto v_1]) + \cdots + W(e'[x \mapsto v_n])$$
$$D(\texttt{map}\ e_1\ e_2) = D(e_1) + D(e_2) + max(D(e'[x \mapsto v_1]), \cdots, D(e'[x \mapsto v_n])) + 1$$
$$\text{where } [\![e_1]\!] = \backslash x\ \texttt{->}\ e'$$
$$\text{where } [\![e_2]\!] = [v_1, \cdots, v_n]$$

`map2` maps over two arrays instead of one, and its work-depth is similar except the function takes two arguments and work or depth of evaluating the second array is added respectively. You may suspect that `zip` and `unzip` has the same work-depth, as it is similar to `map2` where the function puts the arrays together or separates them. However, `zip` and `unzip` have no runtime cost in most cases as they can be fused into other operations by the Futhark implementation (Elsman *et al.*, n.d.(b), ch.2, sec.2.2).

In Table 2.2, a selection of the most useful parallel functions are shown. We will refer to this table when making asymptotic work-depth analysis of our Futhark programs later. Be aware that asymptotic notation eliminates all smaller terms which might be considerable. E.g. `filter`, `scan` and `reduce` have the same asymptotic work-depth but

Function	Asymptotic Work	Asymptotic Depth
`filter`	$O(n)$	$O(\log(n))$
`iota`	$O(n)$	$O(1)$
`map`	$O(n)$	$O(1)$
`reduce`	$O(n)$	$O(\log(n))$
`replicate`	$O(n)$	$O(1)$
`scan`	$O(n)$	$O(\log(n))$
`scatter`	$O(n)$	$O(1)$
`tabulate`	$O(n)$	$O(1)$

Table 2.2: Asymptotic work-depth of parallel Futhark functions (input functions are assumed $O(1)$) (Elsman *et al.*, n.d.(a))

`filter` is much slower than `scan` and `scan` is much slower than `reduce` (Elsman *et al.*, n.d.(a)).

2.3.3.2 Work-Depth Analysis of Example Program

Here is an example of a simple asymptotic work-depth analysis for a Futhark program. We have written an example program which is shown in Listing 2.1. The program calculates the product of the two input matrices.

```
 1 -- Matrix multiplication (the dot product of the rows of 'xss'
 2 -- with the columns of 'yss').
 3 let main [n][m][p] (xss: [n][p]f32) (yss: [p][m]f32) : [n][m]f32 =
 4   map (\i -> map (\j ->
 5                      reduce (+) 0 (map
 6                                     (\k -> xss[i, k] * yss[k, j])
 7                                     (iota p)
 8                                   )
 9                 ) (iota m)
10       ) (iota n)
```

Listing 2.1: Matrix product function in Futhark

We start by analysing the inner most part of the expression which is the `map` on line 5. Firstly it maps over an `iota` which has $O(n)$ work and $O(1)$ depth. The function in `map` only uses functions that are $O(1)$. This means that the whole `map` has $O(p+p) = O(p)$ work and $O(1+1) = O(1)$ depth. The `reduce` uses a function with $O(1)$ work and depth. The `reduce` and `map` steps are done sequentially which means the so far total is $O(p+p) = O(p)$ work and $O(1+\log(p)) = O(\log(p))$ depth. The reduce is in the function of the last `map` on line 4. In Futhark, nested parallelism is allowed so the `map` is not forced to be sequential. Thus the `map` over `iota m` has $O(mp)$ work and $O(1+\log(p)) = O(\log(p))$ depth. The `map` is in the function of another `map` over `iota n`.

It has $O(nmp)$ work and $O(\log(p))$ depth, which is the final asymptotic work and depth of the program. The sequential counterpart to this algorithm has $O(nmp)$ work as well so the program is work efficient.

3 Implementation

Our implementation is attached to the report. There is a user guide in appendix 8.1.

3.1 A Language for L-Systems

Our domain specific language (DSL) can be used to define deterministic, context-free L-systems that can be either bracketed or non-bracketed. It starts by defining the axiom in nonterminal *Axiom*, then it defines the productions in *Rules* and optionally visual interpretations can be defined with *Inps*. The visual interpretations are the turtle commands *move*, *draw* and *turn*. Move and draw takes a step length in pixels and turn takes a degree (not radians). The symbols can only be the upper- or lowercase letters A-Z. Notice that this disallows the usage of special characters like + or - though they are common in L-system definitions. This is discussed further in section 5. Another design decision is to only allow non-zero integers as arguments to the turtle commands. They cannot be zero as turning 0 degrees or moving or drawing 0 pixels would be the same as no interpretation for that character, so it would bring redundancy into the language. It could be changed to include decimal numbers by changing the regular expression and their representation types to floats in the implementation.

Using F#, we have implemented an interpreter and a compiler for the language which convert the given language to a representation understandable by the computer. The interpreter and compiler can both derive and visually interpret the L-systems of this language. The compiler can produce parallel programs using different parallel derivation strategies. Parallel derivation strategies are discussed in section 3.2 and parallel visual interpretation is discussed in section 3.3.

3.1.1 Formal Definition

$$
\begin{aligned}
\mathit{Prog} &\rightarrow \mathit{Axiom}\ \mathit{Rules} \mid \mathit{Axiom}\ \mathit{Rules}\ \mathit{Inps} \\
\mathit{Axiom} &\rightarrow \texttt{axiom:}\ \mathit{Str} \\
\mathit{Rules} &\rightarrow \texttt{rules:}\ \mathit{Rule} \\
\mathit{Rule} &\rightarrow \mathbf{char}\ \texttt{->}\ \mathit{Str} \mid \mathbf{char}\ \texttt{->}\ \mathit{Str}\,\texttt{,}\ \mathit{Rule} \\
\mathit{Inps} &\rightarrow \texttt{inter:}\ \mathit{Inp} \\
\mathit{Inp} &\rightarrow \mathbf{char}\texttt{:}\ \mathit{Mv} \mid \mathbf{char}\texttt{:}\ \mathit{Mv}\,\texttt{,}\ Inp \\
\mathit{Mv} &\rightarrow \texttt{move}\ \mathbf{num} \mid \texttt{draw}\ \mathbf{num} \mid \texttt{turn}\ \mathbf{num} \\
\mathit{Str} &\rightarrow \mathbf{char} \mid \mathbf{char}\ \mathit{Str} \mid \texttt{[}\mathit{Str}\texttt{]} \mid \texttt{[}\mathit{Str}\texttt{]}\mathit{Str}
\end{aligned}
$$

Where **char** is defined with the regular expression:

```
[a-zA-Z]
```

And **num** is defined with the regular expression:

$$\texttt{[-]?[1-9][0-9]}^*$$

All whitespace is optional and is ignored in the language.

3.1.2 Example

An example of an L-system is the binary tree which has the productions $A \rightarrow B[QA]WA$ and $B \rightarrow BB$ and the axiom A (Wikipedia, n.d.).

```
axiom: A
rules:
A -> B[QA]WA,
B -> BB
```

The interpretation for each character is optional. The program of the binary tree with interpretations would be:

```
axiom: A
rules:
A -> B[QA]WA,
B -> BB
```

```
inter:
A: draw 10,
B: draw 10,
Q: turn -45,
W: turn 45
```

There can also be characters that have no rule (constants), or characters with no visual interpretation. In this example, *W* and *Q* have no rules. The binary tree will be a continues example throughout the implementation chapter.

3.2 Derivation in Futhark

We have made four different parallel derivation strategies that can derive L-systems written in our DSL. The compiler can compile programs in the DSL to each of these strategies (ee user guide in appendix 8.1). For each strategy, Futhark code is included: This contains only the main loop and not e.g. conversion functions for ASCII. The rest of the code can be viewed in the attached code base. We have analysed the work-depth of each strategy to get an idea of the efficiency of each of them. The analyses draw on the work-depth of Futhark's main parallel functions in Table 2.2.

Because of Futhark's limitations, it is impossible to do exactly as described in the previous work of Lipp *et al.*, 2009. They manually divide the string into chunks and assign them to threads so they do some sequential work which makes the depth of their algorithms higher. This is not needed in Futhark as threads are handled by the parallel functions. Also they make extensive use of irregular arrays and parallel inline updates which are not possible in Futhark in the same way. We can work around these problems using the parallel functions but not exactly as described. For comparison with our strategies, we will start out by making an asymptotic work-depth analysis of the derivation passes from the previous work.

3.2.1 Work-Depth of Strategy from Previous Work

First we assume the handled L-system is a deterministic, context-free L-system so it handles the same L-system type as our strategies. This means that we have chosen to ignore all representations and work done for e.g. parametric systems. The algorithm uses chunking but this is not included in the asymptotic work-depth analysis. In practice, it is actually faster to use chunking than full parallelism if the level of chunking is chosen wisely because of the overhead of communication between threads (Henriksen, 2017, p. 18). Additionally, Futhark also uses chunking so it would be an unfair comparison if chunking were only taken into consideration for this strategy.

Assume that the axiom is a symbols long. At the first iteration, the first pass finds a production for each symbol and gets the length of the successor. This is done with a map that is $O(a)$ work and $O(1)$ depth. The second pass scans over pass 1 to get offsets, which takes $O(a)$ work and $O(\log(a))$ depth. The third pass rewrites the symbols to successors in parallel using the offsets, which takes $O(al)$ work and $O(l)$ depth as the worst case is that it only applies the longest successor of length l. The final work-depth of an iteration is thus $O(al)$ work and $O(\log(a) + l)$ depth but we would like the work-depth of all iterations in total. We know that the generated symbols for the last iteration is at most $a \cdot l^n$ when n is iteration number and a is the axiom length (cf. section 2.1.4). Thus the total work-depth is $O(a \cdot l^n)$ work and $O(\log(a \cdot l^{n-1}) + nl) = O(\log(a) + \log(l^{n-1}) + nl) = O(\log(a) + (n-1)\log(l) + nl) = O(\log(a) + nl)$ depth. Be aware that $\log(a)$ will often be very low compared to nl. A sequential counterpart to this algorithm would also have $O(a \cdot l^n)$ work because it has to look at all elements of the string. Therefore it is work efficient.

3.2.2 Strategy 1

3.2.2.1 Representations

The representation is very similar to the one discussed section 2.2. We have not worked with parametric L-systems and we have therefore simplified it by removing the parameter number from the header and the table for parameter lookup. The production rules are stored in a two-dimensional array. As mentioned Futhark does not have a char type so the symbol representation has to be a number. To optimise space usage we have chosen not to index with the ASCII values but instead we assign a unique ID to each character used in the L-system. Conveniently, we can also use the ID to index into the production array as each symbol can only have one production. This also means that the array for each production only contains the successor as the predecessor is given implicitly as the index. If a symbol is a constant, its production will be to itself, so $a \to a$ if a is a constant. A side effect is that the same character can have different IDs in different L-systems. This method comes with the trade-off that when the derivation is finished, the IDs need to be converted back to the ASCII values so the user can know which number represents which symbol. This is done by simply having an array with the ASCII values indexed by the IDs. It takes space linear to the number of characters in the L-system. The conversion

is then simply a map over the derivation result.
Futhark does not support irregular arrays which means all the representations of productions need to have the same length. Therefore, dummy values have been added as needed at the end of productions so they are all as long as the longest production in the set. We have chosen -1 as dummy value because our symbol IDs cannot be negative values so the dummy value can easily recognised.

An example could be an L-system with the productions $A \rightarrow AB$ and $B \rightarrow A$. The ID for A could be 0 and the ID for B would then have to be 1. The representation for the productions would be $[[0,1],[0,-1]]$ and the array for ASCII conversion would be $[65,66]$. The only other representation possible is $[[1,-1],[1,0]]$ and $[66,65]$ where A have ID 1 and B have ID 0.

3.2.2.2 Algorithm

```
let str = loop cur_str = axiom for i < n do
    let 2d = map (\x -> rules[x]) cur_str
    let flat = flatten 2d
    in filter (!= -1) flat
```

Notice that Futhark has sequential loops where the header contains a mutable variable initialisation. The variable is set to the loop result after each iteration and in this case it contains the current string. `i` starts from 0 and increments by 1 each iteration until it ends at $i = n$.
The algorithm has three passes:

1. *Retrieve successors*: It `map`s over the current string and makes a 2D array with the successors from the applied rules.

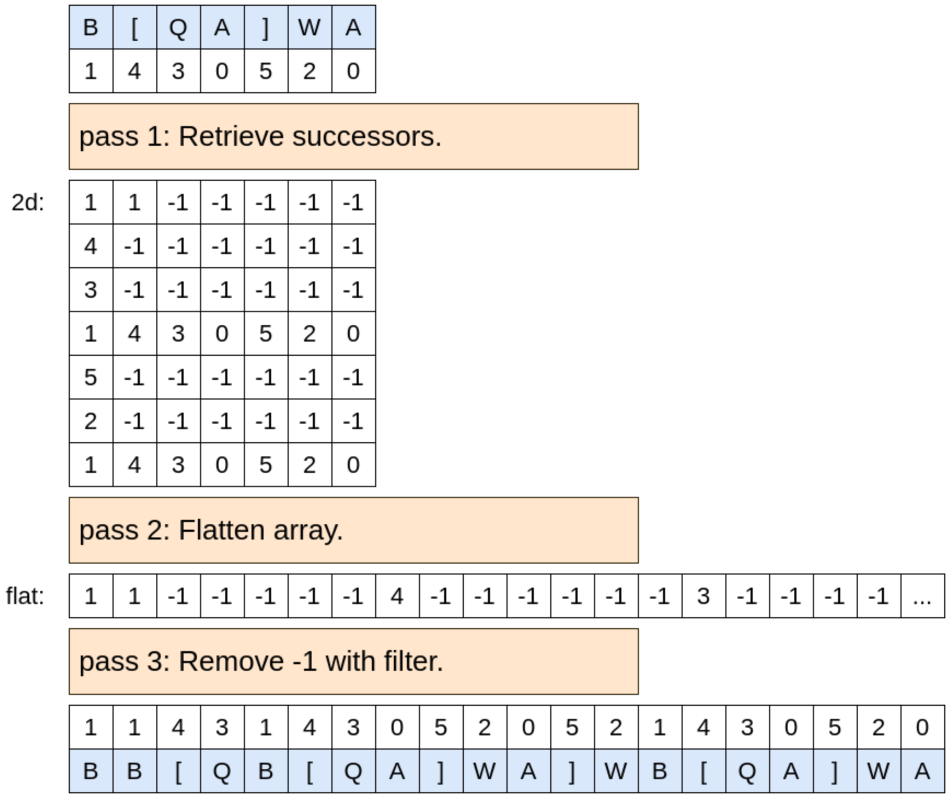
2. *Flatten array*: It `flatten`s the array to a 1D array

3. *Filter*: It `filter`s out the dummy values.

This is done iteratively to `n` iterations. You can see an example in Figure 3.1.

3.2.2.3 Work-Depth Asymptotic Analysis

We start by analysing the loop body and assume it is the first iteration. We say a is the number of characters in the axiom. The `map` input function uses only array indexing



B	[	Q	A	]	W	A
1	4	3	0	5	2	0

pass 1: Retrieve successors.

2d:

1	1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1
1	4	3	0	5	2	0
5	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1
1	4	3	0	5	2	0

pass 2: Flatten array.

flat:

1	1	-1	-1	-1	-1	-1	4	-1	-1	-1	-1	-1	-1	3	-1	-1	-1	-1	...

pass 3: Remove -1 with filter.

1	1	4	3	1	4	3	0	5	2	0	5	2	1	4	3	0	5	2	0
B	B	[	Q	B	[	Q	A	]	W	A	]	W	B	[	Q	A	]	W	A

Figure 3.1: The three passes of strategy 1 performed in an iteration on the binary tree fractal. Its grammar can be seen in section 3.1.2.

which is $O(1)$, so the `map` takes $O(a)$ work and $O(1)$ depth. `flatten` takes $O(1)$ work and depth so the total work-depth does not change. The result of `flatten` is an array with the length of all the applied productions concatenated. As all production arrays have been padded to be as long as the longest successor, the length of `flat` is la when l is the length of the longest successor. Therefore, `filter` is given an array of length la and takes $O(la)$ work and $O(\log(la))$ depth so the total work-depth is updated. It results in an array with a length of at most la symbols when no elements are removed. The work and depth of each iteration of the loop will grow with the iteration number because the string grows. The worst case is that `filter` removes no elements in any iteration in which case the loop takes $O(a \cdot l^n)$ work and $O(\log(a \cdot l^n)) = O(\log(a) + n\log(l))$ depth. It is work efficient because the same sequential algorithm would also have to look at all symbols generated in the last iteration.

3.2.2.4 Advantages and Disadvantages

A disadvantage of this approach is the waste of space. If the L-system grammar has a lot of variance in production length, the inner arrays of `map` could contain a lot of dummy values together, which has no information. This is removed by `filter` but if the number of symbols grows exponentially, then the number of applied productions

grows exponentially and so could the wasted space (depending on the L-system). The storage of the productions themselves also contain dummy values of course but this takes a constant amount of space for a specific L-system. However for systems with almost equally long productions, the space usage will be close to optimal.
As mentioned in section 2.3.3.1, `filter` is a relatively slow function compared to the other parallel functions in Futhark, even though the asymptotic depth is $O(\log n)$. We will see how much this affects runtime in section 4.
The biggest advantage is the simplicity of the approach where there is not needed a lot of operations on the data to complete an iteration. Thus we suspect that it will perform well for small strings where the overhead of doing the preparation work for efficient parallelization from the other approaches are larger than the gain. Also it could perform well for L-systems where all productions have (almost) the same length, as the depth of the `filter` operation depends on the length of the string including dummy values.

3.2.3 Strategy 2

3.2.3.1 Representations

The representation for this algorithm is exactly the same as the one for strategy 1, except each production has a header at index 0 containing the length of the production. E.g. the L-system with the productions $A \to AB$ and $B \to A$ would have the representation $[[2,0,1],[1,0,-1]]$ and $[65,66]$. In the representation in section 3.2.2.1, the dummy value needs to match the `filter` condition in the algorithm but in this algorithm the dummy value is never used so the exact value does not matter.

3.2.3.2 Algorithm

```
let str = loop cur_str = axiom for i < n do
    let m = length cur_str
    let lens = map (\x -> rules[x, 0]) cur_str
    let acc = scan (+) 0 lens
    let len = last acc
    let exacc = take m ([0] ++ acc)

    let flag = scatter (replicate len false) exacc (replicate m true)
    let tmp = scatter (replicate len 0) exacc cur_str
    let ind1d = segmented_scan (+) 0 flag tmp

    let ones = replicate len 1
    let ind2d = segmented_scan (+) 0 flag ones
    in map (\(x, y) -> rules[x, y]) (zip ind1d ind2d)
```

The algorithm has the following passes in an iteration:

1. *Get production lengths*: It `maps` over the string and uses the symbols as indexes into the production array. It retrieves the length of each applied production from their header and puts it in an array, `lens`.

2. *Get offsets for productions*: It `scans` over the production lengths to compute the offsets. However, Futhark's `scan` is inclusive and here an exclusive scan is needed. This is due to the fact that we want the first offset to be the starting accumulator, 0, and the last offset is currently the length of the string after the iteration so that one should be removed. The offset array is made in `exacc` (exclusive scan, hence the variable name).

3. *Make flag array*: It `scatters` over a boolean array which contains false except in the production offsets where it contains true.

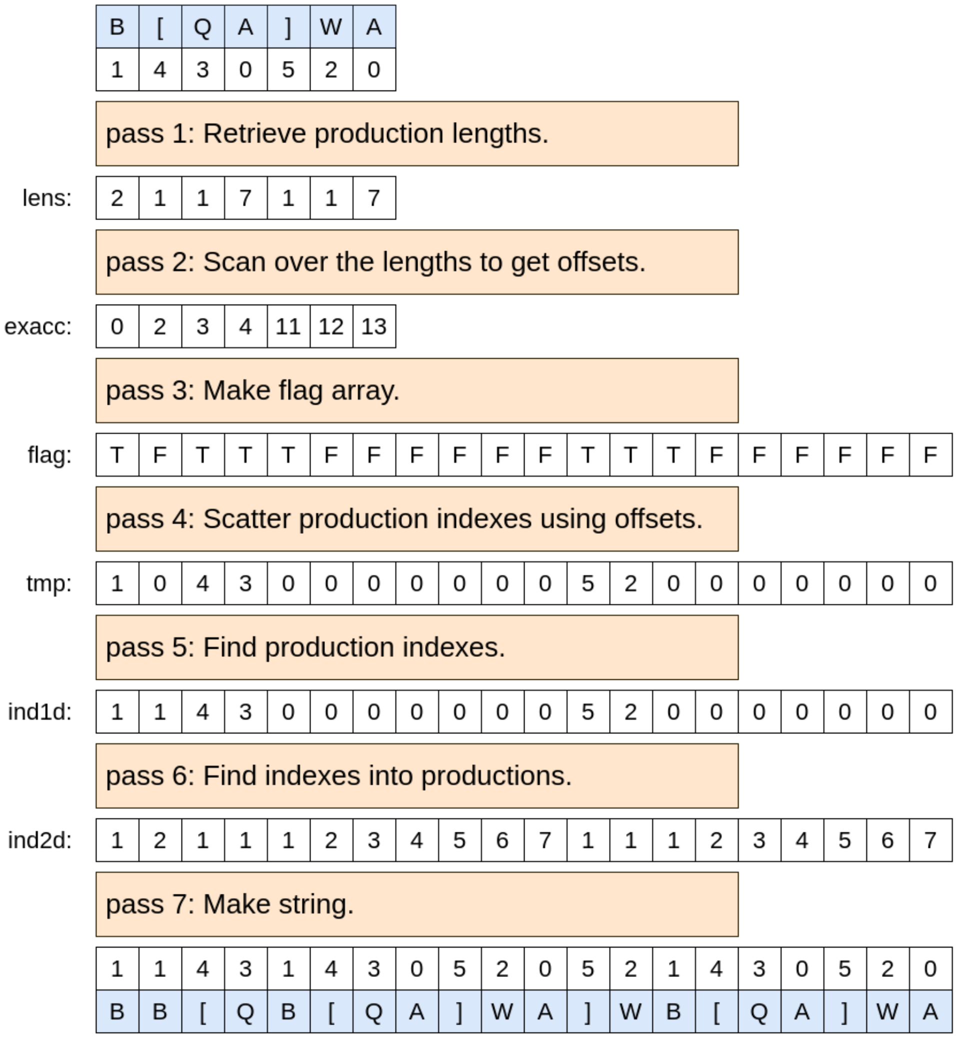
4. *Scatter production indexes using offsets*: It uses the current string to retrieve applied production indexes and put each index in the according production offset. The rest of the values are 0.

5. *Get production indexes*: It uses segmented scan which is a helper function that scans within the segments marked by a flag array (Elsman *et al.*, n.d.(b), sec. 7.1). It segment scans over pass 4 to put the indexes for the symbols into their corresponding segments. This means that the segment reserved for an applied production will contain the symbol index for that production.

6. *Get indexes into the productions*: It segment scans over an array with ones using the flag array. This results in an array where an element contains the index into the applied production.

7. *Make the string*: It `maps` over the two index arrays and uses them to find and insert the symbols in the final string.

Again a loop is used for iterating over the string, as this can only be done sequentially. You can see an example in Figure 3.2.



B	[	Q	A	]	W	A
1	4	3	0	5	2	0

pass 1: Retrieve production lengths.

lens:	2	1	1	7	1	1	7

pass 2: Scan over the lengths to get offsets.

exacc:	0	2	3	4	11	12	13

pass 3: Make flag array.

flag:	T	F	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F

pass 4: Scatter production indexes using offsets.

tmp:	1	0	4	3	0	0	0	0	0	0	0	5	2	0	0	0	0	0	0	0

pass 5: Find production indexes.

ind1d:	1	1	4	3	0	0	0	0	0	0	0	5	2	0	0	0	0	0	0	0

pass 6: Find indexes into productions.

ind2d:	1	2	1	1	1	2	3	4	5	6	7	1	1	1	2	3	4	5	6	7

pass 7: Make string.

1	1	4	3	1	4	3	0	5	2	0	5	2	1	4	3	0	5	2	0
B	B	[	Q	B	[	Q	A	]	W	A	]	W	B	[	Q	A	]	W	A

Figure 3.2: The passes of strategy 2 performed in an iteration on the binary tree fractal.

3.2.3.3 Work-Depth Asymptotic Analysis

Again, we start with the loop body and assume it is the first iteration. `length` takes constant work-depth. When a is the number of symbols in the axiom, the map on line 3 takes $O(a)$ work and $O(1)$ depth as the input function is $O(1)$. The `scan` on line 4 takes $O(a)$ work and $O(\log(a))$ depth. Lines 5-7 does not exceed this work-depth so we jump to line 8. Here the `replicate` on `len` takes $O(k)$ work and $O(1)$ depth when `len` $= k$ and the replicate on `m` takes $O(a)$ work and $O(1)$ depth. `len` and k is the length of the resulting string after the iteration. It is most likely that $k >$ `m` because strings normally become longer with each iteration but it is not ensured. Thus the total work is now $O(a + k)$. The `scatters` on line 8 and 9 takes $O(a)$ work and $O(1)$ depth so this does not change anything. The work-depth of `segmented_scan` is the same as `scan` so it takes $O(k)$ work and $O(\log(k))$ depth because `len` $= k$. Thus the total depth is updated to $O(\log(a) + \log(k)) = O(\log(ak))$. The `replicate` on line 12 does not exceed it and the next `segmented_scan` has the same work-depth. The final `map` has $O(k)$ work and $O(1)$ depth and `zip` is normally cost-free in Futhark. Thus that work-depth of the first iteration is $O(a + k)$ work and $O(\log(ak))$ depth. The worst case is that it only applies the production with the longest successor with length l in which case $k = la$. Notice that this bound is very conservative as often there are constants or small successors so k can be much smaller than al. Then work is $O(a + la) = O(la)$ and depth is $O(\log(a^2 l)) = O(2 \cdot \log(al)) = O(\log(al))$ of an iteration. This is the same as an iteration for strategy 1, so the work-depth for n iterations is $O(a \cdot l^n)$ work and $O(\log(a) + n \log(l))$ depth. A sequential algorithm doing the same passes would also have work as the number of generated symbols, so it is work efficient.

3.2.3.4 Advantages and Disadvantages

Compared to strategy 1, this strategy is much more space efficiently as it does not include the dummy values when applying productions. It still has dummy values in the production array which is a disadvantage but this takes only a constant amount of space for a specific L-system.

This algorithm passes over the data many times which will affect the runtime, even though it does not change the asymptotic work-depth. The benchmark tests will show how much this slows down the derivation.

3.2.4 Strategy 3

3.2.4.1 Representations

This representation stores all the productions in a one dimensional array. Every production has a header with the successor length so it is possible to find start and end points for productions in the array. The ID for a symbol is the index of the header in the production array. Since productions' indexes will not be in a row, there will be dummy values in the conversion array to ASCII as some indexes are unused.
For example, the L-system with the productions $A \to AB$ and $B \to A$ would have the representation $[2, 0, 3, 1, 0]$ and $[65, -1, -1, 66]$.

3.2.4.2 Algorithm

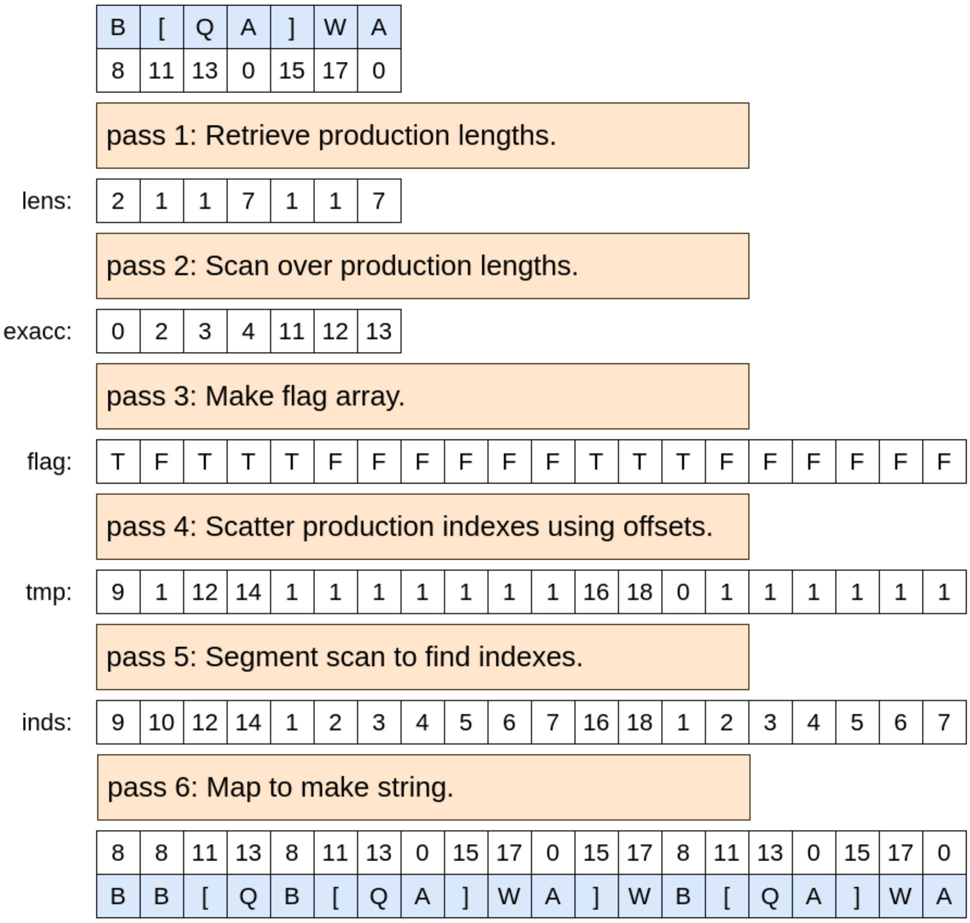
```
let str = loop cur_str = axiom for i < n do
    let m = length cur_str
    let lens = map (\x -> rules[x]) cur_str
    let acc = scan (+) 0 lens
    let len = last acc
    let exacc = take m ([0] ++ acc)

    let flag = scatter (replicate len false) exacc (replicate m true)

    let tmp = scatter (replicate len 1) exacc (map (+1) cur_str)
    let inds = segmented_scan (+) 0 flag tmp
    in map (\x -> rules[x]) inds
```

This strategy is much like strategy 2, except the last few passes become simpler with the new representation. The first three passes are the same as strategy 2, so you can look there for a more elaborate description of those passes. The algorithm has the following passes:

1. *Get production lengths*: Like strategy 2, it maps over the current string and uses the symbol IDs to index into the production array and fetch the headers of the applied productions.

2. *Get offsets for productions*: Like strategy 2, it scans over the production lengths to get offsets.

3. *Make flag array*: Like strategy 2, it makes the flag array with true in the production offsets, false otherwise.



B [Q A] W A

8 11 13 0 15 17 0

pass 1: Retrieve production lengths.

lens: 2 1 1 7 1 1 7

pass 2: Scan over production lengths.

exacc: 0 2 3 4 11 12 13

pass 3: Make flag array.

flag: T F T T T F F F F F F T T T F F F F F F

pass 4: Scatter production indexes using offsets.

tmp: 9 1 12 14 1 1 1 1 1 1 1 16 18 0 1 1 1 1 1 1

pass 5: Segment scan to find indexes.

inds: 9 10 12 14 1 2 3 4 5 6 7 16 18 1 2 3 4 5 6 7

pass 6: Map to make string.

8 8 11 13 8 11 13 0 15 17 0 15 17 8 11 13 0 15 17 0

B B [Q B [Q A] W A] W B [Q A] W A

Figure 3.3: The passes of strategy 3 performed in an iteration on the binary tree fractal.

4. *Scatter productions indexes using offsets*: It adds 1 to every ID of the current string because the successor of each production starts after the header which is one long. Then it scatters these values into an array with only 1's using the production offsets.

5. *Find indexes into production array*: It uses `segmented_scan` with the flag array and pass 4 to make it increment the values in segments for the productions. This means that the result of pass 5 is the list of indexes into the production array, to make the next string.

6. *Make the string*: It maps over the indexes from pass 5 to fetch the successor IDs from the production array.

An example is shown in Figure 3.3.

3.2.4.3 Work-Depth Asymptotic Analysis

Until and including line 8, the work-depth analysis is the same as for strategy 2 so we start from line 10 (9 is empty) and say we have $O(a + k)$ work and $O(\log(ak))$ depth. The `replicate` and `map` on line 10 both takes $O(a)$ work and $O(1)$ depth. The `scatter` does not change the analysis either with its $O(k)$ work and $O(1)$ depth. `segmented_scan` has $O(k)$ work and $O(\log(k))$ depth and the last `map` takes $O(k)$ work and $O(1)$ depth. Thus the total work-depth for n iterations is exactly the same as the two other strategies: $O(a \cdot l^n)$ work and $O(\log(a) + n\log(l))$ depth. With the same argument as strategy 2, the algorithm is work efficient.

3.2.4.4 Advantages and Disadvantages

This strategy is very similar to strategy 2 but because of the one-dimensional representation, the indexes to the production array can be made with fewer passes. Thus we predict that this strategy will perform better out of the two.

This strategy wastes some space in the ASCII conversion array while none in the production array as strategy 2. This strategy will waste as much elements as the total length of all productions which is at least one element per production except the last one. Thus strategy 3 will waste most space if the productions are long, while strategy 2 wastes the most when one production is much longer than the others.

3.2.5 Strategy 4

3.2.5.1 Representations

The representation is the same as for strategy 2. However, we have chosen to use ASCII values for this strategy instead of unique IDs but both ways are possible.

3.2.5.2 Algorithm

```
let str = loop cur_str = axiom for i < n do
   let k = length rules[0] - 1
   let m: i32 = length cur_str
   let lens: [m]i32 = map (\x -> rules[x, 0]) (cur_str :> [m]i32)
   let acc: [m]i32 = scan (+) 0 lens
   let len: i32 = last acc
```

```
    let to_write l =
      let (i, j) = (l / k, l % k)
      let symbol = cur_str[i]
      let dest_i = if i == 0 then j else acc[i-1]+j
      let to = rules[symbol,j+1]
      in if to == -1 then (-1, 0) else (dest_i, to)
    let (is, vs) = unzip (tabulate (m*k) to_write)
    let new_str = scatter (replicate len 0) is vs
    in new_str
  in str
```

The algorithm has the following passes:

1. *Get production lengths*: It `map`s over the current string and gets the lengths of the applied productions from their headers.

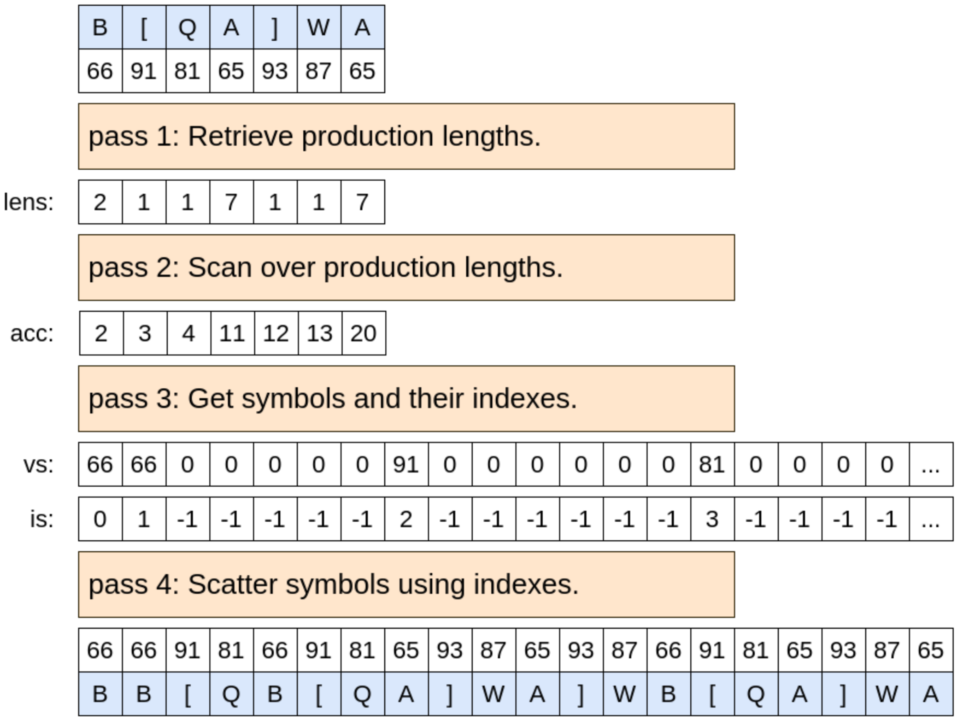
2. *Get production offsets*: It `scan`s over pass 1 to get the production offsets.

3. *Get symbols and their indexes into the final string*: The longest possible string after an iteration would be constructed by only applying the longest production to all symbols. Therefore, the upperbound on the length is mk when m is the length of the current string and k is the length of the longest successor. We take advantage of the upperbound by making two arrays of length mk: The symbols for the final string in `vs` and their indexes in `is`. We use a function, `to_write`, to get the two arrays with the following steps:

   a) Do integer division and modulo on the index in the array with the longest successor. Integer division is used to index in the current string while the modulo and the offset in `acc` together can be used to index into the result string with dummy values, `vs`.

   b) Get the ASCII value of the current symbol from the string.

   c) Compute the destination index for the symbol using offsets from `acc` and the previous step. The if-statement is only needed because Futhark's `scan` is not exclusive, so the offsets are displaced by 1 in `acc` and it does not contain the first offset, 0.



	B	[	Q	A	]	W	A													
	66	91	81	65	93	87	65													

pass 1: Retrieve production lengths.

lens:	2	1	1	7	1	1	7

pass 2: Scan over production lengths.

acc:	2	3	4	11	12	13	20

pass 3: Get symbols and their indexes.

vs:	66	66	0	0	0	0	0	91	0	0	0	0	0	0	81	0	0	0	0	...
is:	0	1	-1	-1	-1	-1	-1	2	-1	-1	-1	-1	-1	-1	3	-1	-1	-1	-1	...

pass 4: Scatter symbols using indexes.

66	66	91	81	66	91	81	65	93	87	65	93	87	66	91	81	65	93	87	65
B	B	[	Q	B	[	Q	A	]	W	A	]	W	B	[	Q	A	]	W	A

Figure 3.4: The passes of strategy 4 performed in an iteration on the binary tree fractal.

d) Use the current symbol as index into the production array and the modulo result to get the specific symbol from the production. This might be a dummy value, as the array for a production is always k long.

e) If the symbol from step 4 was a dummy value, -1, it puts a dummy value in each of the arrays. If it was an ASCII value, it puts the destination index in the index array and the ASCII value in the symbol array.

4. *Make the final array*: It `scatter`s the symbols from `vs` into the final array using the indexes from `is`. All dummy values are ignored by `scatter` as it cannot use -1 as an index.

An example is shown in Figure 3.4.

3.2.5.3 Work-Depth Asymptotic Analysis

We start by analysing the loop body with the assumption that it is the first iteration and the axiom has length a. `length` takes constant time so the first relevant operation is a `map`. It takes $O(a)$ work and $O(1)$ depth. The `scan` takes $O(a)$ and $O(\log(a))$ depth so we update the total depth. `last` takes constant time so it is irrelevant. The input function to `tabulate` only uses constant time operations, so that means `tabulate` requires $O(1)$

depth and $O(al)$ work when l is the length of the longest successor. Now, the total work-depth is $O(al)$ work and $O(\log(a))$. `unzip` is normally cost-free so it does not affect the analysis. In the worst case, `scatter` is $O(al)$ work and $O(1)$ depth so it does not affect it either. Therefore, one iteration takes $O(kl)$ work and $O(\log(k))$ when k is the length of the input string. In the worst case, only the longest successor is applied in all iterations so there are $a \cdot l^n$ after n iterations. Thus, the work-depth of n iterations of the loop is $O(a \cdot l^n)$ work and $O(\log(a \cdot l^{n-1})) = O(\log(a) + n\log(l))$. Like the other strategies, it is work efficient.

3.2.5.4 Advantages and Disadvantages

Generally we expect this strategy to perform well. It only uses one of the expensive `scan` operations and this is over the input string and not the output string like strategy 2 and 3. Also we do not need to make the `scan` exclusive because that is handled by the if-statement in line 10. This means that we can leave out the `take` from the other strategies.

Strategy 4 has fewer passes over the data as it can get indexes and symbols for the final array in just one pass which is beneficial for the runtime. However, it will probably perform badly if the L-system has both really long and short productions since it has many dummy values and would run out of threads sooner than strategy 2 and 3. It could also run out of memory sooner.

3.3 Visual Interpretation in Futhark

There are two steps to visualisation: Produce the geometry and convert it to a picture format. Geometry production is done by creating an array with line segments where each line segment consists of a starting point and an end point. The conversion to a viewable picture is not especially interesting for this project so we have used a Futhark drawing library to make png files (library description: Elsman *et al.*, n.d.(b), sec. 9).
We will not spend too much time describing work-depth asymptotic analyses for the interpretation as they are conducted much in the same way as for derivation. Also we have only one strategy so there are no strategies to compare the work-depth with. It is hard to compare with the strategy from the previous work as they have not included their implementation strategy in detail especially for the work-queue.
It should be noted that our implementation of visualisation is rotated differently than other implementations. Most of the L-systems should have a specific starting rotation

but we have not implemented that because the L-system still looks the same. It should be relatively straight forward to implement: Just extend the language to allow setting a starting rotation and then append a symbol in front of the axiom which rotates the starting rotation to be the one specified in the program. Another problem with the visualization is the fact that the draw library have the y-axis pointing down instead of up. This results in the drawings being upside down. We did not solve these issues because of time constraints and they are not as vital for the usage of the L-systems.

3.3.1 Non-Bracketed L-Systems

```
let main [n] (str: [n]i32): [][4]i32 =
  let ints = ...
  let mapres = map (\ x -> ints[x]) (str :> [n]i32)
  let scanres = scan scanfun ([[1,0,0],[0,1,0],[0,0,1]],0) mapres
  let (lastpos, finallen) = last scanres
  let exscanres = take n ([([[1,0,0],[0,1,0],[0,0,1]],0)] ++ scanres)
  let indices = map2 (\ (_, o) (_, i) -> if o == 1 then i else -1) mapres exscanres
  let (m1, _) = unzip exscanres
  let (m2, _) = unzip scanres
  let points = map (\ i -> [i32.f32 (f32.round m1[i,2,0]), i32.f32 (f32.round m1[i,2,1]), i32.f32 (f32.round m2[i,2,0]), i32.f32 (f32.round m2[i,2,1])]) (iota n)
  in scatter (replicate finallen [0,0,0,0]) indices points
```

Listing 3.1: Interpretation for non-bracketed L-systems (calculates line segments).

The `ints` array holds the interpretations for the specific L-system. Every symbol interpretation is represented as a tuple with an integer and a transformation matrix. The integer is 1 if it produces geometry and 0 if it does not. The algorithm for non-bracketed L-systems has the following passes:

1. *Get matrices and number of objects*: It `map`s over the string to get the symbols' transformation matrices from the interpretation array `ints`.

2. *Scan with matrix multiplication and get object offsets:* `scanfun` does matrix multiplication and integer addition on a tuple. After the `scan`, `scanres` holds the turtle coordinate system after each symbol and how much geometry should be made. The inclusive `scan` is made exclusive by removing the last element and appending the neutral element in the front. The exclusive sum-scan of the integers can be used as offsets into the resulting array with geometry.

3. *Get offsets for line segments into the result array*: It maps over the offsets from pass 2 and the objects array from pass 1. If the symbol produces geometry then it uses the offset from the scan else the symbol should be removed by using a negative index.

4. *Produce line segments between every consecutive symbols*: It uses a map to compute start and end points for the lines to be drawn.

5. *Discard lines of symbols not producing geometry*: It uses scatter with the indices calculated earlier to remove the line segments which should not be drawn.

The algorithm takes $O(n)$ work and $O(\log(n))$ depth, as the most asymptotically expensive function used is a scan over an array with the length of the string (line 4). The input function to scan takes $O(1)$ work-depth as matrix multiplication with a matrix of known size is asymptotically constant.

3.3.2 Bracketed L-Systems

The interpretation of bracketed L-systems is longer and more complicated so we have chosen to divide the explanation in two. The code snippets together compose the whole main function of the program. It is inspired by section 2.2.3.2 but the work queue is much more complicated to do in Futhark than in CUDA.

3.3.2.1 Parallel Hierarchy Extraction

```
let main [n] (str: [n]i32): [][4]i32 =
  let pass1 = map (\x -> (if x == 91 then 1 else if x == 93 then -1 else 0, objects[x])) str
  let pass2com = scan (\ (a, b) (c, d) -> (a + c, b + d)) (0,0) pass1
  let (pass2in, objsin) = unzip pass2com
  let pass2comex = take n ([(0,0)] ++ pass2com)
  let (pass2, offsets) = unzip pass2comex
  let dmax = reduce (\ x y -> i32.max x y) i32.lowest pass2
  let pass3 = map (\i -> map3 (\ c ed id -> if (c == 91 && id == i + 1) || (c == 93 && ed == i + 1) then 1 else 0) str pass2 pass2in) (iota dmax)
  let pass4 = unflatten dmax n (take (n * dmax) ([0] ++ scan (+) 0 (flatten pass3)))
  let pass5help = map (\ i -> if str[i] == 91 then pass4[pass2[i],i] else if str[i] == 93 then pass4[pass2in[i],i] else -1) (iota n)
  let pass5 = scatter (replicate n (-1)) pass5help (iota n)
```

```
12  let pushpop = scatter (replicate n 0) pass5 (map (\ i -> if i % 2 == 0
      && i + 1 < n then pass5[i + 1] else 0) (iota n))
13  (...)
```

Listing 3.2: Interpretation for bracketed L-systems: hierarchy extraction

Hierarchy extraction has six passes:

1. *Find pushes and pops*: It `maps` over the string and gives an array of tuples. The first tuple value is 1 when there is a push and -1 when there is a pop (91 and 93 are ASCII values for '[' and ']' respectively). Otherwise it is 0. The second value is 1 if it generates geometry (draw command) and 0 if it does not (move or turn).

2. *Get depths*: It does an exclusive `scan` over pass 1 to get the depth of each index and get the offsets for the geometry. It also computes the maximal depth with a `reduce` over the offsets.

3. *Make depth arrays*: For each push and pop with position p and depth d do $arr[d, p] = 1$. All other elements should be initialised to 0.

4. *Get offsets for 1D array*: It `scans` over pass 3 to get the offsets for a 1D bucket array.

5. *Make 1D bucket array*: It `maps` over the input string and uses the offsets from pass 4 and the depths found in pass 2 to get the position for each push and pop in the 1D bucket array. Then it `scatters` the push and pop positions into the 1D bucket array.

6. *Integrate the bucket array into the string*: The bucket array is 'integrated' into the string by making a new array where at the position of each push, we put the position of its corresponding pop. When interpreting the string in the work-queue, it can use this array to lookup the pop position when it encounters a push.

The most asymptotically expensive function in this algorithm is the `scan` on line 3 which has $O(n)$ work and $O(\log(n))$ depth when the string is n long.

3.3.2.2 Work Queue

```
(...)
let (_, pointsin) = loop (tmp, matrices) = ([(0, ints[str[0]])],
 replicate n [[1,0,0],[0,1,0],[0,0,1]]) while length tmp != 0 do
```

```
    let (work, init) = unzip tmp
    let m = length work
    let (lens, ends) = unzip (map (\ x -> let en = loop i = x while i < n && str[i] != 91 && str[i] != 93 do i + 1 in (en - x + if en < n then 1 else 0, en) ) (work :> [m]i32))
    let scanres = scan (+) 0 lens
    let scanresex = take m ([0] ++ scanres)
    let len = last scanres
    let flag = scatter (replicate len (0, false)) scanresex (zip (work :> [m]i32) (replicate m true))
    let iotas = modified_segmented_scan (+) 0 flag (replicate len 1)
    let flag = scatter (replicate len ([[1,0,0],[0,1,0],[0,0,1]],false)) scanresex (zip (init :> [m][3][3]f32) (replicate m true))
    let ps = modified_segmented_scan scanfun [[1,0,0],[0,1,0],[0,0,1]] flag (map (\ i -> ints[str[i]]) iotas)
    let newmatrices = scatter matrices iotas ps
    let newwork_items = map (\ i -> if i >= n || str[i] == 93 then [(-1, [[1,0,0],[0,1,0],[0,0,1]]), (-1,[[1,0,0],[0,1,0],[0,0,1]])] else [(i+1, matmul ints[str[i+1]] newmatrices[i-1]), let ind = pushpop[i] + 1 in (ind, matmul ints[str[ind]] newmatrices[i-1])]) ends
    let newwork = filter (\ (x,_) -> x != -1) (flatten newwork_items)
    in (newwork, newmatrices)

  let pointsex = take n ([[[1,0,0],[0,1,0],[0,0,1]]] ++ pointsin)
  let points = map2 (\ m1 m2 -> [i32.f32 (f32.round m1[2,0]), i32.f32 (f32.round m1[2,1]), i32.f32 (f32.round m2[2,0]), i32.f32 (f32.round m2[2,1])]) pointsex pointsin
  let indices = map2 (\ (_, o) i -> if o == 1 then i else -1) pass1 offsets
  in scatter (replicate (last objsin) [0,0,0,0]) indices points
```

Listing 3.3: Interpretation for bracketed L-systems: work queue

The approach for the work queue presented by Lipp *et al.*, 2009 uses a lot of sequential work which is not possible in Futhark. The work queue is a sequential loop which continues until there is no more work (symbols to interpret). A work item consists of a tuple with the index where it should start interpreting, and a starting turtle state for that index. It continues interpreting symbols sequentially from that work item's index until it finds a push and it branches out.

7. *Get the length of each work*: For each work item find how many symbols should be interpreted sequentially. This is done by a `map` over the work items and for each work item, it sequentially finds the number of symbols until the next push/pop (or end of the string).

8. *Get the total number of symbols to interpret and offsets*: It `scan`s over the lengths of the work items to find the total number of symbols to interpret and the offsets.

9. *Get an array with the indexes of all the symbols which are to be interpreted*: It uses a modified segmented scan which when encountering a true flag does not place the element from the input array but instead places the element which are associated with the flag (the flag is a tuple). The input flag array holds tuples which are the work item index and `true` in the work offsets and `(0,false)` otherwise. This marks the segments of the modified segmented `scan`. It results in an array with the indexes of the symbols to be interpreted. For example the work items $[1, 10]$ with work lengths $[2, 3]$ would generate $[1, 2, 10, 11, 12]$.

10. *Interpret the symbols*: It `map`s over the previous array and gets the matrix for each symbol. Then it `scan`s over the matrix array with matrix multiplication. This is done with the modified segmented `scan` where the flag array is the same as before, except it has the starting turtle state matrices from the work items instead of the indexes. The matrices are `scatter`ed into the matrix array.

11. *Generate the new work items*: It `map`s over the work items and generates two new work items from each element. Some cannot generate new work items in which case it should generate dummy work items. These dummy work items are then `filter`ed away.

12. *Compute line segments*: This is done just like in the non-bracketed version of interpretation.

Figure 3.5 shows an example of the visualisation by this algorithm where the line segments are converted into a picture. Due to the size and volume of the algorithm, it is not feasible and brings little value to dry run and show all the details of the visualisation passes here. The picture is the result of running our implementation on the language example showed in section 3.1.2 after three derivation iterations as in the examples shown for the derivation strategies.

The runtime of the work queue depends on how many work items there are (the number of brackets). The worst case occurs when there are no brackets as then the loop in the `map` on line 5 will go through all elements in the string sequentially. Thus the work-depth is $O(n)$, when the string is n long. Be aware that the worst case is rare as a bracketed L-system only rarely produces strings with no brackets. Our compiler checks if there are any brackets in the L-system, and only uses the bracketed approach for visualisation if the system is bracketed.

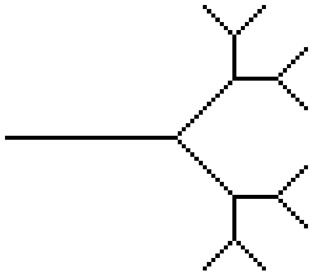


Figure 3.5: Binary tree for 3 iterations computed by our parallel algorithm for bracketed interpretation.

In the article, one thread starts by interpreting sequentially and new work items are distributed to new threads as they appear. The work queue is implemented with a loop since it is not possible to just write new work items into an array and have new threads handle them as they discovered. Futhark also have restrictions on in-place array updates which makes the simple sequential interpretation that the article describes more or less impossible to implement. We discuss other ways the work queue might be implemented in section 5.

3.4 Testing Correctness

We have made a broad sample of tests with different L-systems which are mostly based on grammars from *The Algorithmic Beauty of Plants* (Prusinkiewicz and Lindenmayer, 2004). In addition to those, we have a couple of examples from the Wikipedia page about L-systems (Wikipedia, n.d.) and we have made some grammars ourselves which have names beginning with "test".
We have made a test script that handles all tests for both the interpreter and compiler (script is based on a testing script from course IPS 2019 at UCPH). See appendix 8.1 for how to run the script. The interpreter and the compiler is tested with the same tests which are to be found in the folder `tests` of the project. All of our tests succeed for both derivation and visualisation.

We have implemented an interpreter with the main purpose of making test cases for the compiled programs with respect to interpretation. The interpreter's derivation was tested by a test script that compared its results with our dry runned test cases.
The visualisation was tested by comparing sets of list segments from the interpreter and the compiler. A list segment consists of a starting point and an end point for a line. Comparing segments is much more efficient than comparing a full image file which contains a lot of extra data for satisfying the picture syntax. Furthermore when different

picture formats are used the notation will unavoidably be different and the tests would fail. Also we have not implemented the drawing mechanisms ourselves so it would test more implementations apart from our own.
Running visualisation tests by hand proved to be rather impractical. We compared the interpreter's pictures with pictures of the fractal with the same number of iterations (Prusinkiewicz and Lindenmayer, 2004), to catch any errors perceivable by the eye. It would have been much more precise to compare the set of line segments with the expected set but we prioritised other parts of the testing. The interpreter was used to compute results for the tests of the compiler.
At first we had some difficulties comparing the results of the interpreter with the parallel program as the line segments had an opaque type which meant that it could not be returned (cf. section 2.3.1). Therefore, we changed the type to a 2D array for the testing version. In the version that actually draws the picture, it still uses an array of tuples as the drawing library needs that type.

We have chosen to test only positive cases of the L-system language due to time constraints. It is our assessment that it is more important to test functionality such as derivation, than testing that the DSL fails when the user writes in a wrong syntax. However, it would be important if the language should be used by many people and especially people without much knowledge in computer science. In that case they would probably not understand the errors so it would be more important but that is not the case with this project. Also if there were any serious mistakes in the parser or lexer of the DSL, they would probably have surfaced at the positive test cases.

Evaluation 4

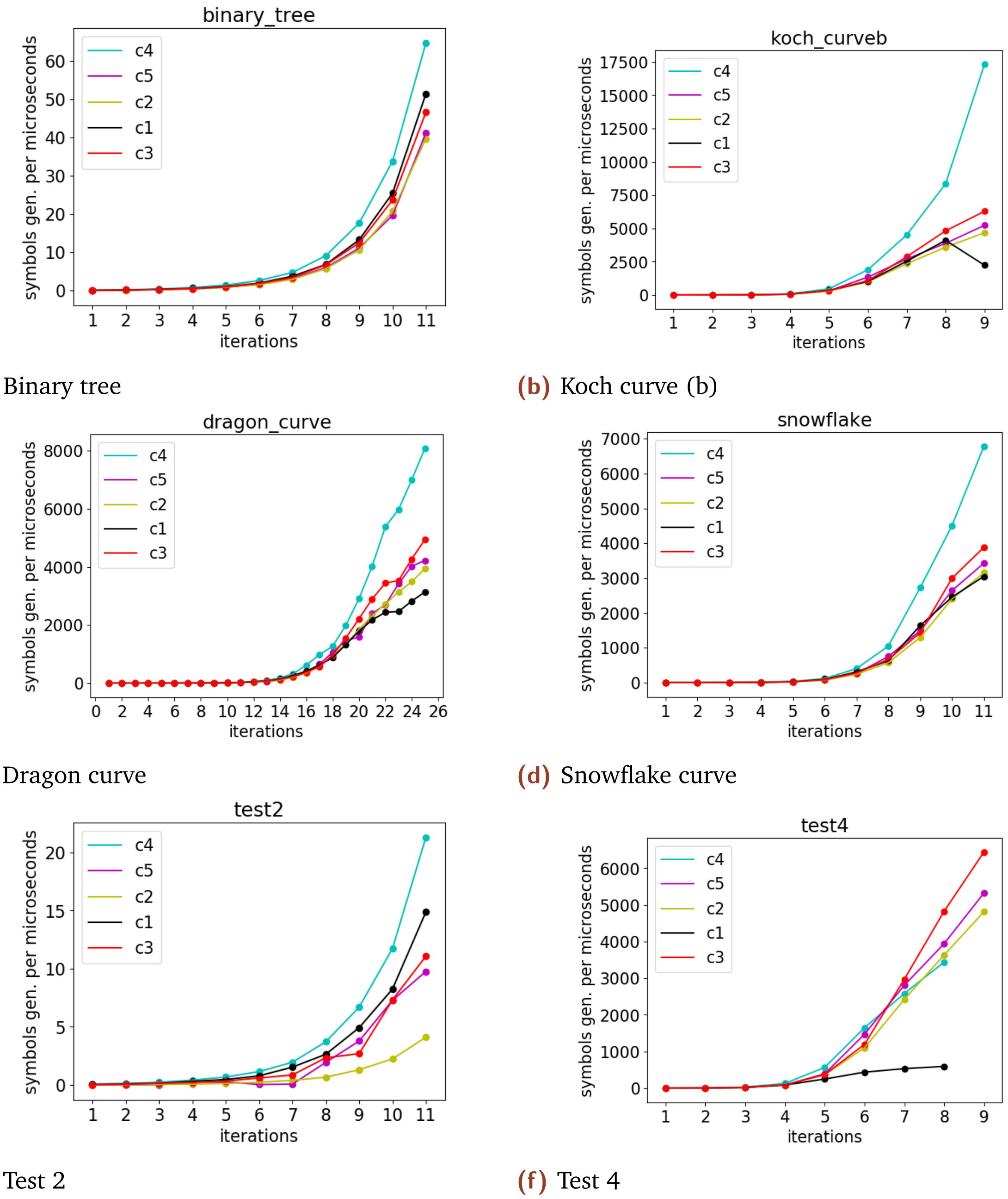
We ran our tests on the GPU GeForce RTX 2080 Ti on a server from the University of Copenhagen. The GPU has 4352 CUDA-cores. We have used `futhark bench` with CUDA backend for benchmarking our programs with 10 runs per input for a program (Elsman *et al.*, n.d.(b), sec. futhark-bench). Raw data from the benchmarks are included in the code repository in the folder `benchmarks`.

4.1 Derivation

4.1.1 General Tendencies

Figure 4.1 shows selected benchmark plots for derivation. All of our benchmark plots can be seen in appendix 8.2, including references to the corresponding grammars, so the results can be reproduced. The performance is better the higher the graph as we would like the number of symbols generated per time to be the highest possible. The number of symbols generated per time is the throughput.

We can see irregularities in some of the graphs. This could be due to variance in the GPU performance or someone else using the server at the same time. We could probably receive more predictable data by benchmarking with more than 10 runs per program but we did not do this due to time constraints.
`futhark bench` gives a relative standard deviation (RSD) for each benchmark. We can use this to find out if an irregularity has a high probability of being caused by an outlier. An example is in one of the Koch curves where iteration 9 with strategy 1 seems to be rather low (Figure 4.1b). However, when looking at the RSD graph in Figure 4.2b, the RSD is relatively high for iteration 9. Thus, it is probable that external influences created a single outlier that is dragging the curve downwards and causing an irregularity in the graph.
Some of the spikes on the RSD graphs do not create irregularities in the benchmark curves. There can be two main reasons for this. Firstly, RSD spikes will appear extremely small on the derivation graph if they occur at small symbol numbers. This is because



(a) Binary tree

(b) Koch curve (b)

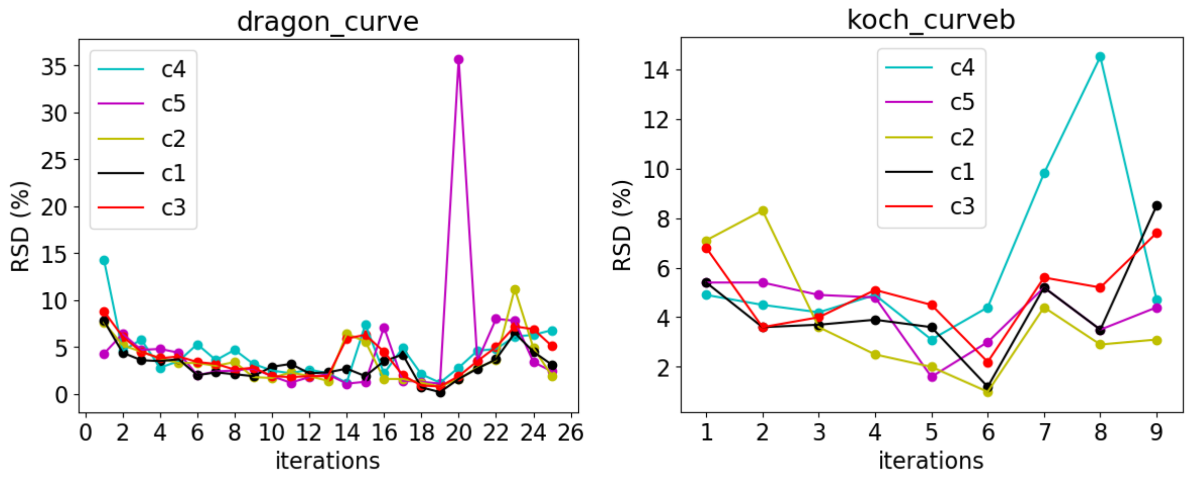
(c) Dragon curve

(d) Snowflake curve

(e) Test 2

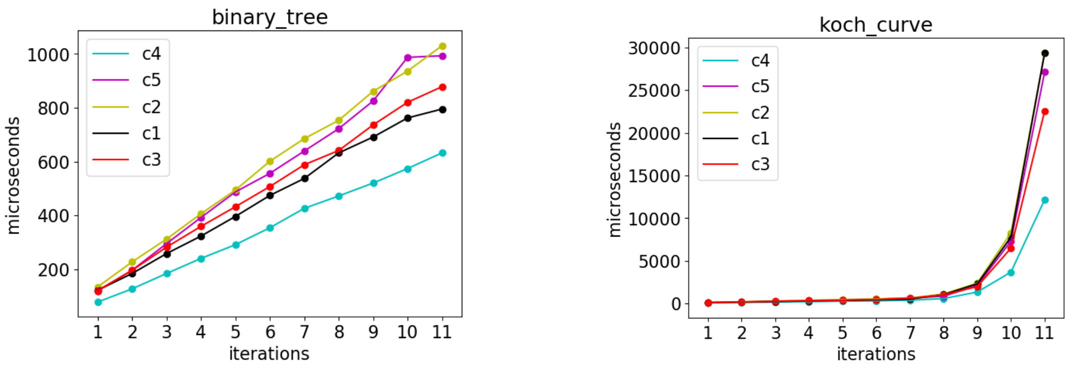
(f) Test 4

Figure 4.1: Benchmark curves for derivation of a selection of L-systems. The line cX uses strategy number X. The rest of the plots for the L-systems can be seen in appendix 8.2.



(a) Dragon curve **(b)** Koch curve (b)

Figure 4.2: Relative standard deviation as a function of iterations. The rest of the plots for the L-systems can be seen in appendix 8.3.



(a) Binary tree **(b)** Dragon curve

Figure 4.3: Runtime as a function of iterations. The line cX uses strategy number X. The rest of the plots for the curves can be seen in appendix 8.4.

the number of symbols generated per microsecond is close to 0 so even very high RSD percentages will be invisible on the graphs. Secondly, it could be that multiple outliers is cancelling each other out so the graph does not show an irregularity. The overall tendencies are still visible through the irregularities.

On an ideal machine with no overheads or processor limitations, we would expect the generated symbols per microseconds (throughput) to be a linear function of the number of iterations for a specific L-system. This is due to the fact that the depth of the algorithms is $O(\log(a) + n\log(l))$ where $\log(a)$ and $\log(l)$ are constants for a specific L-system so only the iteration number n matters. However, the graphs grow rather slowly in the beginning and then increase roughly linearly. Some stagnate their growth a bit at high iterations which we will explain in a bit. The first part where the throughput is almost constant and very low, is probably caused by the overheads which affect the runtime more in the small iterations. On a real machine, we would not expect the graphs to be perfectly linear, because the GPU can run out of threads at which point the throughput would stagnate in its growth. Brent's theorem says that the runtime is bounded as:

$$\frac{W(n)}{P} \leq T \leq \frac{W(n)}{P} + D(n)$$
$$\frac{O(a \cdot l^n)}{4352} \leq Runtime \leq \frac{O(a \cdot l^n)}{4352} + O(\log(a)) + n\log(l)) = O(a \cdot l^n)$$

when we have 4352 GPU threads. When the amount of work is less than the amount that the GPU can handle, the runtime will be bounded by the depth since $W(n)/P$ is below 1. However at some point, the amount of work will be too large for the threads and because the work grows exponentially with n, the runtime will be affected considerably by the work instead of (almost) only by the depth. When the iteration number goes to infinity, the asymptotic runtime is bounded by the work $O(a \cdot l^n)$ and no longer by the depth like when the GPU had enough threads. As the work is larger than the depth, we would expect the throughput to decrease because the symbols per time becomes smaller. Thus the growth of the graph would decrease when the GPU cannot handle the whole string in parallel anymore. The growth would eventually become negative and then converge towards a constant value for the throughput as the runtime becomes closer to the work. None of the benchmark curves for the L-systems begins to fall but we can still see the effect described by Brent's theorem in Figure 4.3. For the slowly growing binary tree, it does not run out of threads so Brent's theorem does not affect the expected linear function. The Koch curve reaches the number of symbols where the amount of work is too big for the GPU, so the runtime begins to grow exponentially which is what Brent's theorem predicts because the work is an exponential function of the iteration number.

4.1.2 Comparing the Strategies

With small iteration numbers, the performance differences between the strategies are very small. We think it is because there are only small numbers of symbols so the overheads count more than the actual strategies.

In most cases, strategy 4 is the fastest strategy by far. This is consistent with expectations since it goes through the data a small number of times and uses a low number of expensive operations. However, our theory was that it would perform badly for L-systems with very long productions. Therefore, we made test 4 with an L-system that had one 90 symbols long production and otherwise only used constants (Figure 4.1f). In test 4, strategy 4 is slower than the other strategies and runs out of memory at 8 iterations, which were also one of the expected disadvantages (section 3.2.5.4). Be aware that

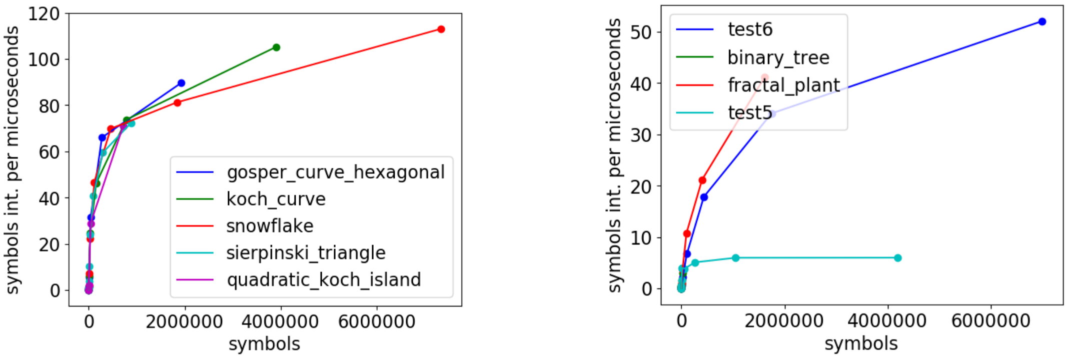
these kinds of L-systems are rare so strategy 4 will be best in most cases.
Strategy 1 performs even worse in test 4. It is probably because it uses the expensive `filter` function with $O(\log(n))$ depth over the string with the dummy values. It runs out of memory after 8 iterations like strategy 4 which is expected as it uses an array of the same length as the array with dummy values in strategy 4. However, generally strategy 1 performs well especially at small symbol numbers as seen in the binary tree (Figure 4.1a) and L-systems with equally long productions (Figure 4.1e). In these cases, using the expensive `filter` operation has proven to be worth the cost compared to e.g. exchanging it for `scans` and `scatters` in strategy 2 and 3.

Strategy 2 is slower than strategy 3 like suspected. Strategy 2 and 3 are very similar with the difference being that 3 uses one segmented scan less so it makes sense that strategy 3 is faster. The graphs c2 and c5 in Figure 4.1 are both strategy 2 where c5 uses the ASCII values instead of unique IDs. c2 was converted by making both the production array and ASCII conversion table 128 long, so we can use ASCII to index instead of unique IDs. c5 was consistently faster than c2 which means that the `maps` between ASCII and IDs were actually affecting the runtime quite a lot. The idea behind the unique IDs was to try to reduce the overhead of uploading the production array to the GPU but it has proven to be faster to use the ASCII values directly. However, the performance difference between c2 and c5 is very small.

4.2 Visual Interpretation

Benchmark curves for interpretation can be seen in Figure 4.4. We expect the same general tendencies as for derivation but be aware that the x-axis is the number of symbols instead of iterations. For non-bracketed systems the depth is $O(\log(k))$ where k is the number of symbols so the throughput as a function of symbols should be logarithmic and Figure 4.4a shows just that. The interpretation of bracketed L-systems uses different amounts of sequential work so it is harder to set a general tendency for the plots. If the sizes of the work items match with a good chunking size, the depth will be logarithmic like for the non-bracketed L-systems (see e.g. test 6). If the sizes of the work items are too big however, it will do a lot of sequential work and the curve will quickly settle on a somewhat constant throughput (see the graph for test 5).

As all non-bracketed systems use the same strategy and the representation of the string is the same for all systems, we would expect them to be equally fast for a given number of symbols. The only operation that could have different runtime for strings of the



(a) Non-bracketed L-systems

(b) Bracketed L-systems

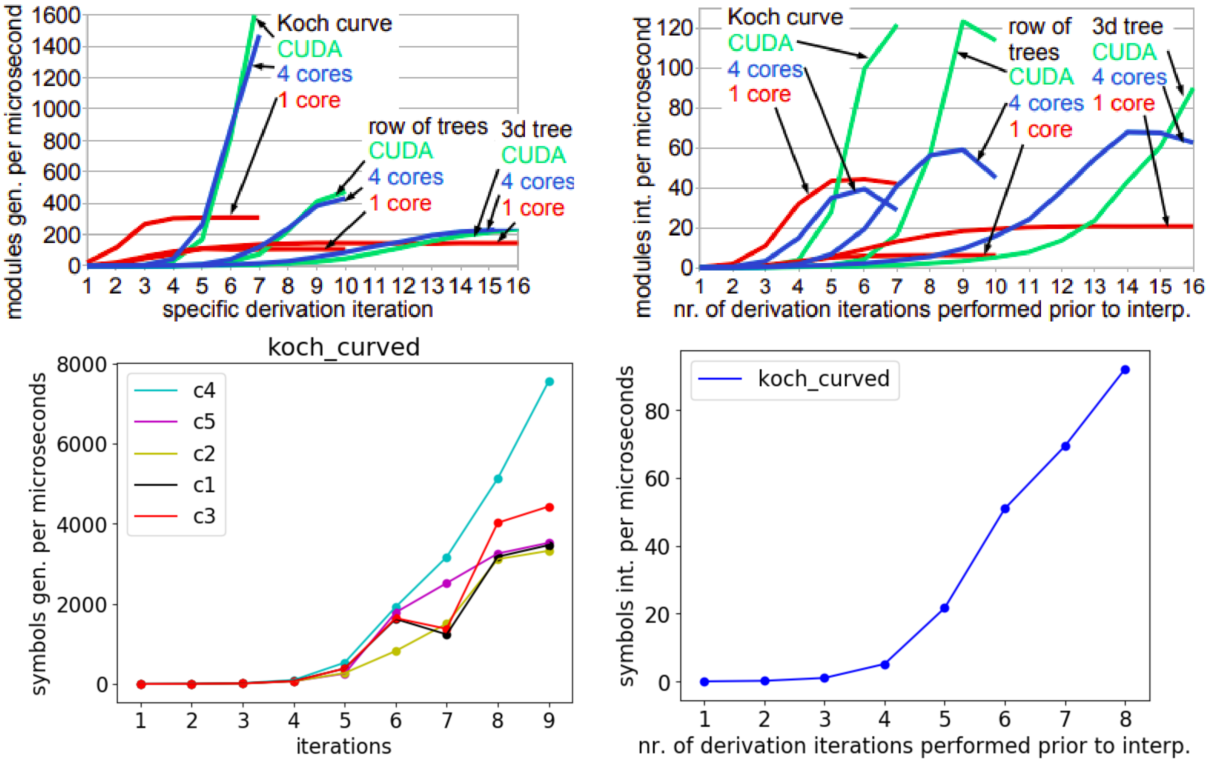
Figure 4.4: Benchmark curves for interpretation of a selection of L-systems. Symbols interpretation per microseconds as a function of symbols in the string.

same length is the `scatter` to place the generated lines because the L-systems can have different numbers of symbols that use the turtle command *draw*. `scatter` has $O(1)$ depth so this should only generate differences between L-systems of the same length if the GPU does not have enough threads to `scatter` the geometry for one L-system and does have enough for another L-system. In the plot, the graphs for non-bracketed systems lay roughly on top of each other which match with the expectation.

For bracketed L-systems however, we would expect L-systems to behave differently depending on the length of work items. Therefore, the ratio between brackets and other symbols in the string affects the runtime. The only production in test 6 is $a \to [aaaa]$ so it has a high number of brackets compared to other symbols. Therefore, the sequentially handled work items are fairly short and it has a high number of symbols interpreted per microsecond. However, it seems that the work item lengths in fractal plant is closer to optimal since its throughput is higher. Additionally, brackets has no visual interpretations so the array with transformation matrices is short and thus the `scan` over the matrices becomes faster in test 6.

Test 5 represents another edge case which is that there is only a single bracket in the L-system. It has only three work items: The symbols before the push, the symbols inside the brackets and the symbols after the pop. It is expected to be very slow as almost all symbols are in one single work item and each work item is handled sequentially. As expected Figure 4.4b shows that test 5 has a much lower throughput than the other bracketed L-systems.

Notice that the bracketed L-systems are much slower than the non-bracketed L-systems. This is also expected as the bracketed visualisation strategy has a great deal more passes and sequential work.



(a) Koch curve derivation

(b) Koch curve interpretation

Figure 4.5: Benchmarks for derivation and interpretation approaches from *Parallel Generation of L-systems* (Lipp *et al.*, 2009, p. 8) and benchmarks for our implementation with the same Koch curve. Be aware that these graphs use generated symbols and runtime for running only the last iteration where the other figures in this report uses all previous iterations as well.

4.3 Comparison with Other Research

We expected that our results would be faster than the results presented in *Parallel Generation of L-Systems* since they use a much slower GPU (Geforce GTX 280). This is also the case for derivation. When we compare derivation, our fastest strategy at 7 iterations Koch curve had about 3173 symbols per microsecond, and they had about 1600 (using the grammar from Prusinkiewicz and Lindenmayer, 2004, p. 10, (d)). This is a 0.5x speed-up from their implementation. However, it is not possible to say if their implementation would be faster on the hardware that we use for benchmarking.

For visual interpretation, Figure 4.5 shows that our implementation is actually slower than their CUDA implementation. E.g. with the 7 iteration Koch curve, they had about 120 symbols per microsecond where we had about 51 symbols per microsecond. This corresponds to a 2.3x speed-up from our implementation.

These speed-ups are rather insecure however, for multiple reasons. Firstly, we only have access to a very limited amount of their data (only the data presented in the article). You should also be aware that we measure the time and generated symbols for the last iteration only by subtracting the time used for the previous iteration. This might affect the precision.

We would also like to note that when reading the data table in Lipp *et al.*, 2009, p. 8, we would assume that absolute time for 1 core divided by the relative speed-up to CUDA would result in the absolute time for the CUDA implementation. However, we get that it takes $3.45/3.2 = 1.078$ milliseconds to derive a 7 iteration Koch curve, but our implementation only took 0.530 milliseconds on average. Furthermore, they claim that the Koch curve has 915,049 symbols after 7 iterations (Lipp *et al.*, 2009, p. 8), where we have only 624,999 symbols after 7 iterations. We thought it might be that the table summed up the generated symbols for all iterations up to the 7^{th} but that only gives 781,233 symbols. The number of symbols for the 2D plant does not match with our number either. When we compute their throughput for the Koch curve derivation with their numbers, it should be $915,049/1078 \approx 849$ symbols per microseconds but their graph says 1600. We have checked our results against a sequential implementation (St-Amour *et al.*, 2007) which gives the same number of symbols as our implementation. Mind that their numbers for interpretation are consistent with their graphs. However, it is still a problem that they have a different number of symbols.

Maybe Lipp *et al.*, 2009 made mistakes in the article or just measure the numbers differently than we expect. In either way this means the comparisons with their benchmarks are dubious at best.

If we assume that their throughput values are correct, it is quite high compared to how much slower their GPU is. We think it might be that our Futhark programs utilise more parallelism than what is optimal for the L-systems and this can be fixed in the CUDA approach where the programmer can adjust the parallelism more directly. For example in their implementation, they copy the symbols of a successor into the resulting string sequentially but we do it in parallel. The length of a successor is normally very short (below 10) and no more than 34 for the L-system grammars we used from Prusinkiewicz and Lindenmayer, 2004. Thus it is reasonable to think that the overhead of making it parallel is more than the gain. Futhark does not allow parallel inline updates in arrays except using `scatter` and this is a weakness in the programming model that prevents us from testing if a less parallel strategy would be faster. Even though the approach of Lipp *et al.*, 2009 has the asymptotic depth $O(\log(a) + nl)$ which is worse than our strategies' depth $O(\log(a) + n\log(l))$, $\log(l)$ normally is not much smaller than l so the overhead of using parallelism to make it $\log(l)$ does not pay off.

It might seem strange that their sequential program runs approximately equally as fast as our parallel implementation for derivation with the 5 iteration Koch curve. However, they say that they have a "highly optimized single-core CPU version"(Lipp *et al.*, 2009, p. 7) so if this means a hand-optimised assembler program, it could be reasonable. After all, Futhark is a high-level programming language so it is hard to compete with that. Although we would expect a sequential implementation to be faster for small numbers of symbols because it does not have the overheads of parallelism but at larger numbers of symbols the parallel implementation should be faster. This is also visible here, as by 7 iterations, our implementation is much faster and it keeps growing where the graph of the sequential implementation seems to be relatively constant after 3 iterations.

5 Future work

This section contains ideas and suggestions for further work and improvements beyond the scope defined by the time limitations of the present project.

5.1 Language Extensions

In the formal notation $+$ and $-$ are used very often along with a specified angle for visualisation which is the amount $+$ and $-$ turn right and left respectively. This angle is also used as the starting angle. In the current language design we decided to support only alphabetical symbols, but other characters can be easily added as our implementation already do an ASCII to ID mapping anyway. If our grammar added more special characters, we would have to consider if the characters cause ambiguity. E.g. if we added the symbols ",:", we would need to escape these as they are used for determining the tokens of our language.
A restriction on the visual interpretation is that the language only allows the turtle commands to be given integers instead of floats. Furthermore, there is no way to specify a starting rotation for the turtle so some fractal pictures will appear rotated compared to the examples from the standard visual representation of those fractals.

Another area of improvement could be the support of more L-system types. As it stands, the language can only describe deterministic, context-free L-system that are either bracketed or non-bracketed. Expanding the language will only affect the derivation part of the implementation as only the bracketed/non-bracketed property of L-system types affects visual interpretation. It should be relatively simple to expand the language itself with more types. For example, we could exchange the *Rule* nonterminal with the following, to implement context-sensitive productions:

$$\begin{aligned}
\mathit{RuleList} \rightarrow \quad & \mathit{RuleType} \mid \mathit{RuleType}\,,\ \mathit{RuleList} \\
\mathit{RuleType} \rightarrow \quad & \mathit{ContextFree} \mid \mathit{ContextSens} \\
\mathit{ContextFree} \rightarrow \quad & \mathbf{char}\ \texttt{->}\ \mathit{Str} \\
\mathit{ContextSens} \rightarrow \quad & \mathbf{char} < \mathbf{char}\ \texttt{->}\ \mathit{Str} \mid \\
& \mathbf{char} > \mathbf{char}\ \texttt{->}\ \mathit{Str} \mid \\
& \mathbf{char} < \mathbf{char} > \mathbf{char}\ \texttt{->}\ \mathit{Str}
\end{aligned}$$

It is relatively straight forward to expand the grammar of the language compared to making the same expansions in the implementation. The header in the productions would be expanded with two elements: One for the unique ID of the preceding context character and one for the unique ID of the succeeding context character. If a production does not contain a preceding or succeeding context, then the header for the missing context will contain a dummy value, e.g. -1. There could be multiple productions for each symbol which are applied in different contexts, so the production array could be an array of arrays, where each of the internal arrays holds every production for a specific predecessor. The representation inside the internal arrays could be either 2D like in strategy 1 and 2 or 1D like strategy 3. However, this representation would potentially include a great deal of dummy values and thus waste a lot of space. Another approach, would be to use a one-dimensional production array where the unique ID of the predecessor is included in the header. The unique ID of a symbol would be the index to the first production for the symbol in the production array and every production for a predecessor should be in a row. There would be no dummy values in the 1D array but it would still have dummy values in the ASCII conversion array.

When applying productions, it would map over the productions to find which production is applicable and then apply it. It depends on the representation exactly how it would map. If more than one production is applicable, we would have to resolve the collision in some way. To do that we would check if there is a consensus for the solution of these collisions.

Be aware that the context of a symbol is not necessarily in the previous or next index of the string in bracketed L-systems. If the previous or next character contains a bracket, the context is determined in the following way:

- Preceding character is...

    1. ... a push '[': The context is the symbol just before the push.

    2. ... a pop ’]’: The context is the symbol just before the corresponding push.

- Succeeding character is...

    1. ... a push ’[’: The context is the symbol just after the push and the symbol just after the corresponding pop. There are two contexts for the predecessor.

    2. ... a pop ’]’: There is no context after.

To implement this, each push and pop would have the position of their corresponding pop or push like in parallel hierarchy extraction.

Stochastic productions would have a representation very similar to the representation of context-sensitive productions. The only change would be to include the probability of the production in the header. When choosing between stochastic productions, we could do like in the previous work and create a new random array each time and use each production index to index into the random array and get a random value for each production. The production with the highest product of probability and random value would be applied.

Parametric L-systems could be implemented in the same way as in the previous work. The parameter lists would be stored in a parameter array where each list has a header containing the length of the list. Each parametric symbol in the string would have the starting index of its parameter list in the parameter array. Lipp *et al.*, 2009 does not include production conditions in their implementation but we could incorporate that as well. As we do not want to assume anything about the length of the condition, you would have to save the condition descriptions in an array. The header of a production would then contain an index to the corresponding condition.

5.2 Work-Queue

The current work queue is really slow and it does a lot of work compared to the one described in the article. One idea that could be tested, would be to increase the amount of sequential work and try to decrease the amount of work done. We do not think it is possible to remove the for-loop since Futhark does not allow us to put threads on stand-by that pick up new work items as they are discovered (a thread pool). Moreover, Futhark does not allow us to do in-place updates for other threads to see which is used in the previous Work. The work queue can therefore only be optimized in the body of the for-loop. One way that might allow the symbols to be interpreted sequentially is to use a map with a sequential loop to determine how many symbols should be interpreted per work item. The sequential loop in the map should work on two arrays: One to keep the results of the interpretation (either the full matrices or just the points) and another to keep the indices of the interpreted symbols. Futhark does not allow irregular arrays so to make sure all the arrays are long enough they would all have to be as long as the highest number of symbols per work item. This might result in big memory waste since there is no guarantee that the symbols per work item is anywhere close to balanced. The result of the map would be an array of tuples (or maybe arrays) where the tuples hold the results of the interpretation and the indices where the results should be scattered into the final interpretation array. The same strategy could be tried for derivation when writing the successors of the applied productions into the final array.

6 Conclusion

This project has 3 notable objectives: to explore how L-system derivations may be parallelized, to define and implement a language for defining and compiling L-systems and lastly to define our own strategies for parallel derivation and interpretation of L-systems written in our language.

The first objective was to explore the theory of L-systems and how they could be parallelised. We based our project on research from theoretical articles and books about the workings of L-systems and how their properties inspire parallelism, as well as theory about massively parallel programming. Notably, we have focused on describing the parallel approach for L-systems used by the article *Parallel Generation of L-systems*(Lipp *et al.*, 2009).

The second was to define and implement a domain specific language for writing L-systems. We have made a language that can describe arbitrary deterministic and context-free L-systems, both bracketed and non-bracketed. We have made a formal grammar for the language and described its limitations and how it could potentially be expanded and improved. For this language, we have written an interpreter and a compiler to parallel programs in Futhark which can both derive and visually interpret the L-systems of our language.

The third notable aspect of the purpose was to construct our own parallel strategies for L-systems, especially for derivation but for visual interpretation as well. We constructed four different derivation strategies and for each of them we made a work-depth analysis and formulated advantages and disadvantages to set expectations for their efficiency. We implemented the algorithms in Futhark and benchmarked them to measure their performance on different L-systems. The benchmarks were used to compare each of the strategies with both our expectations and the performances of the other strategies. Our data matched the expected behavior of each strategy which were based on the included theory of parallelism and our analyses of the algorithms. We found that strategy 4 performed the best in a large majority of the tested L-systems.
For visual interpretation, we made two algorithms: One for non-bracketed L-systems and one for bracketed. Like for derivation, all of our benchmarks on the interpretation

showed the expected tendencies. The algorithms were closely inspired by the approaches described in the previous work but Futhark's limitations proved to set restrictions that made it difficult to translate parts of the bracketed approach to efficient parallel Futhark code. A more low-level programming language with less restrictions might be more suitable to implement an irregular problem like visual interpretation of bracketed L-systems. We also compared our strategies for derivation and interpretation with the benchmarks from the article *Parallel Generation of L-systems*. We found that our derivation achieved better performance, but that our interpretations were slower. We did, however, note that for various reasons, the results are not fully comparable.

Our research discusses and evaluates different kinds of approaches for derivation of L-systems in Futhark. While the specific data for the benchmarks are highly dependent on the used GPU, the comparisons of the approaches can still be used as a guideline to determine the focus for future research in efficient L-system derivation.

Bibliography

7

St-Amour, Vincent, Daniel Feltey, Spencer P. Florence, Shu-Hung You, and Robert Bruce Findler (Sept. 2007). *Herbarium Racketensis: A Stroll through the Woods*. eng. Vol. 1. 1. Northwestern University, USA.

Cormen, Thomas H., Charles Eric Leiserson, Ronald Rivest, and Clifford Stein (2009). *Introduction to algorithms*. eng. Third edition. Cambridge, Massachusetts: MIT Press.

Elsman, Martin, Troels Henriksen, and Cosmin E. Oancea (n.d.[a]). *Futhark Library Documentation: prelude/soacs*. eng. https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html (accessed at 05-05-2020).

Elsman, Martin, Troels Henriksen, and Cosmin E. Oancea (n.d.[b]). *Parallel Programming in Futhark*. eng. https://futhark-book.readthedocs.io/en/latest/index.html (accessed at 24-04-2020).

Henriksen, Troels (Dec. 2017). *Design and Implementation of the Futhark Programming Language (Revised). PhD thesis*. eng. University of Copenhagen.

Henriksen, Troels and Martin Elsman (Nov. 2019). *Cost models and advanced Futhark programming*. eng. Lecture Slides for Parallel Functional Programming. Department of Computer Science, University of Copenhagen.

Ju, Tao, Scott Schaefer, and Ron Goldman (Mar. 2004). *Recursive Turtle Programs and Iterated Affine Transformations*. eng. Department of Computer Science, Rice University, Houston.

Lipp, Markus, Peter Wonka, and Michael Wimmer (2009). *Parallel Generation of L-Systems*. eng. http://peterwonka.net/Publications/pdfs/2009.VMV.Lipp.ParallelGenerationOfLSystems.final.pdf.

Oancea, Cosmin E. (Sept. 2018). *PMPH Lecture Notes for the Software Track*. eng. Vol. 1. 1. Department of Computer Science, University of Copenhagen, Chapter 3.

Prusinkiewicz, Przemyslaw and Aristid Lindenmayer (2004). *The Algorithmic Beauty of Plants*. eng. Digital edition. New York: Springer-Verlag.

Santell, J. (Dec. 2019). *L-systems*. eng. https://jsantell.com/l-systems/ (accessed at 07-03-2020).

Wikipedia, contributors (n.d.). *L-system*. eng. https://en.wikipedia.org/wiki/L-system (accessed at 08-04-2020).

8 Appendixes

8.1 User Guide

We have made an interpreter and three compilers to Futhark with different strategies (strategies covered in section 3.2). Compile the code by the command:

```
$ make
```

You can run the test programs with the interpreter by the command:

```
$ ./runtests.sh
```

To test the derivation with a specific strategy use:

```
$ ./runtests.sh -cx
```

Where x can 1, 2, 3, 4 and 5. The numbers correspond to the strategy with the same number, except 5 which is included for testing reasons. 5 is the same as 2 except it uses the ASCII values for the characters and not the unique IDs. To test the drawing use:

```
$ ./runtests.sh -d -cx
```

Where x again is the strategy used for derivation.

The compiler has the command `linsys` which takes a description of an L-system in an `ls` file. It can be run with a number of flags and flag combinations which are:

- `./linsys -i file`: uses the interpreter to derive the L-system specified in file. It will ask for the number of iterations.

- `./linsys -i -d file`: uses the interpreter to derive and draw the L-system specified in `file`. It will ask for the number of iterations and then generate an `svg` file with the same name as `file` except the file extension is `.svg` instead of `.ls`.

- `./linsys -s file`: same as `-i` but with no instructions for the user in the console.

- `./linsys -s -d file`: same as `-i -d` but with no instructions for the user in the console.

- `./linsys -cx file`: generates a Futhark program which derives the L-system specified in file. The program will take the number of iterations as argument. x specifies the strategy used for derivation.

- `./linsys -cx -p file`: generates a Futhark program which derives the L-system and then calculates all the start and end points of all lines in the geometry. The program takes the number of iterations as argument and x specifies the strategy used for derivation. This is used by the test script.

- `./linsys -cx -d file`: the same as `-cx -p` but instead of just start and end points, it outputs a 2D array that can be turned into a `png` by `data2png.py`[1]. The script only takes binary data so you should use the flag `-b` when running the Futhark program.

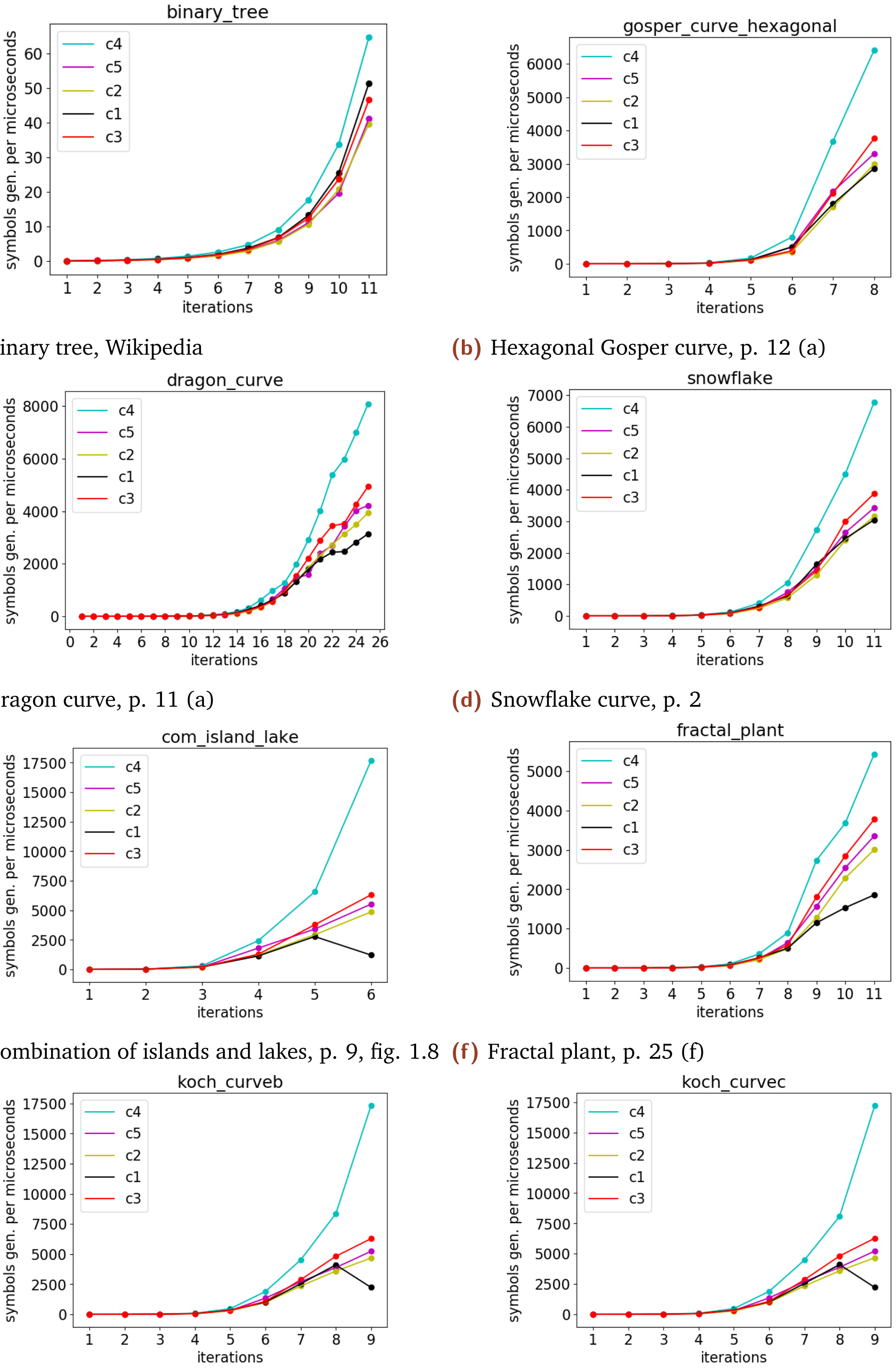
- `./linsys -pd file`: generates a Futhark program which can visualise the L-system in `file`. The Futhark program takes an array of ASCII character integers as argument. Any ASCII characters in the file with no interpretation in the L-system will not generate geometry.

- `./linsys -sp file`: uses the interpreter to derive the L-system and then calculates the start and ends points of all the line in the geometry. It takes the iteration number as input.

- `./linsys -pp file`: same as `-pd` except that it computes start and end points of all lines in the geometry.

[1] https://github.com/diku-dk/futhark/tree/master/tools

8.2 Derivation Runtimes



(a) Binary tree, Wikipedia

(b) Hexagonal Gosper curve, p. 12 (a)

(c) Dragon curve, p. 11 (a)

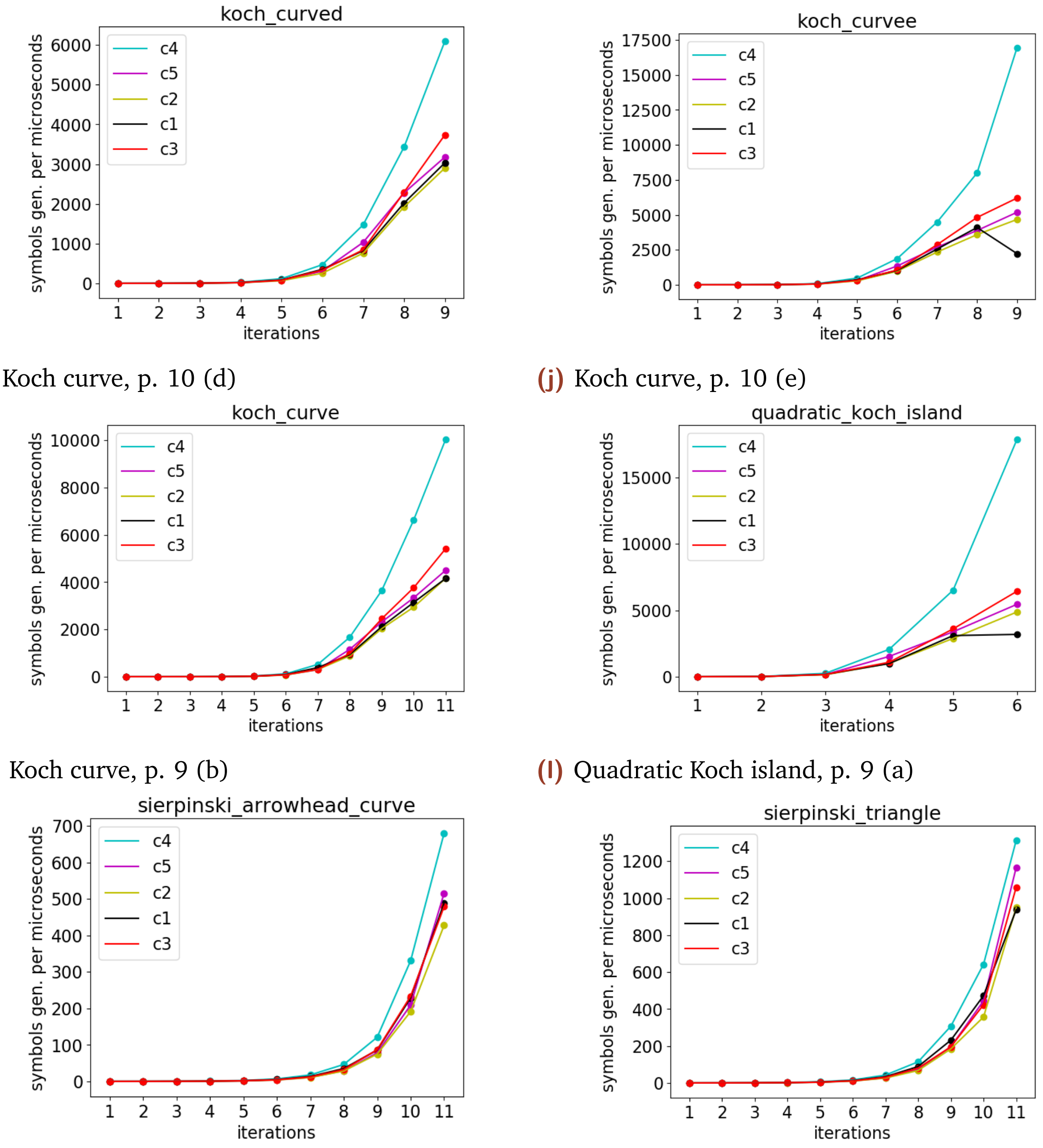
(d) Snowflake curve, p. 2

(e) Combination of islands and lakes, p. 9, fig. 1.8

(f) Fractal plant, p. 25 (f)

(g) Koch curve, p. 10 (b)

(h) Koch curve, p. 10 (c)



(i) Koch curve, p. 10 (d)

(j) Koch curve, p. 10 (e)

(k) Koch curve, p. 9 (b)

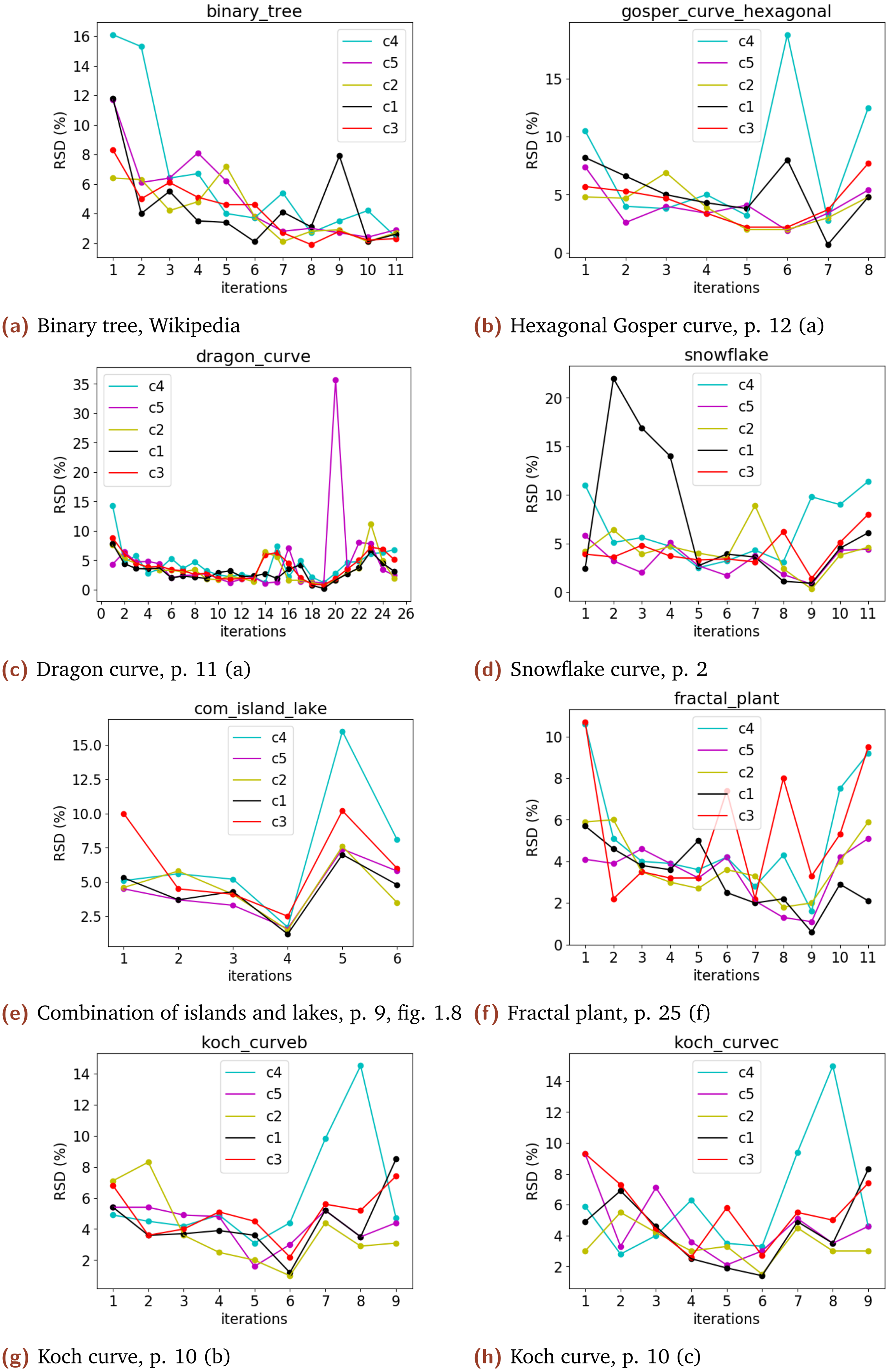
(l) Quadratic Koch island, p. 9 (a)

(m) Sierpinski gasket, p. 11 (b)

(n) Sierpinski triangle, Wikipedia

Figure 8.1: Benchmark curves for derivation of L-systems with references to their descriptions: Wikipedia page about L-systems (Wikipedia, n.d.) or Prusinkiewicz and Lindenmayer, 2004. The graph cX uses strategy number X. Generated symbols per microseconds as a function of iterations.

8.3 RSD on Derivation



(a) Binary tree, Wikipedia

(b) Hexagonal Gosper curve, p. 12 (a)

(c) Dragon curve, p. 11 (a)

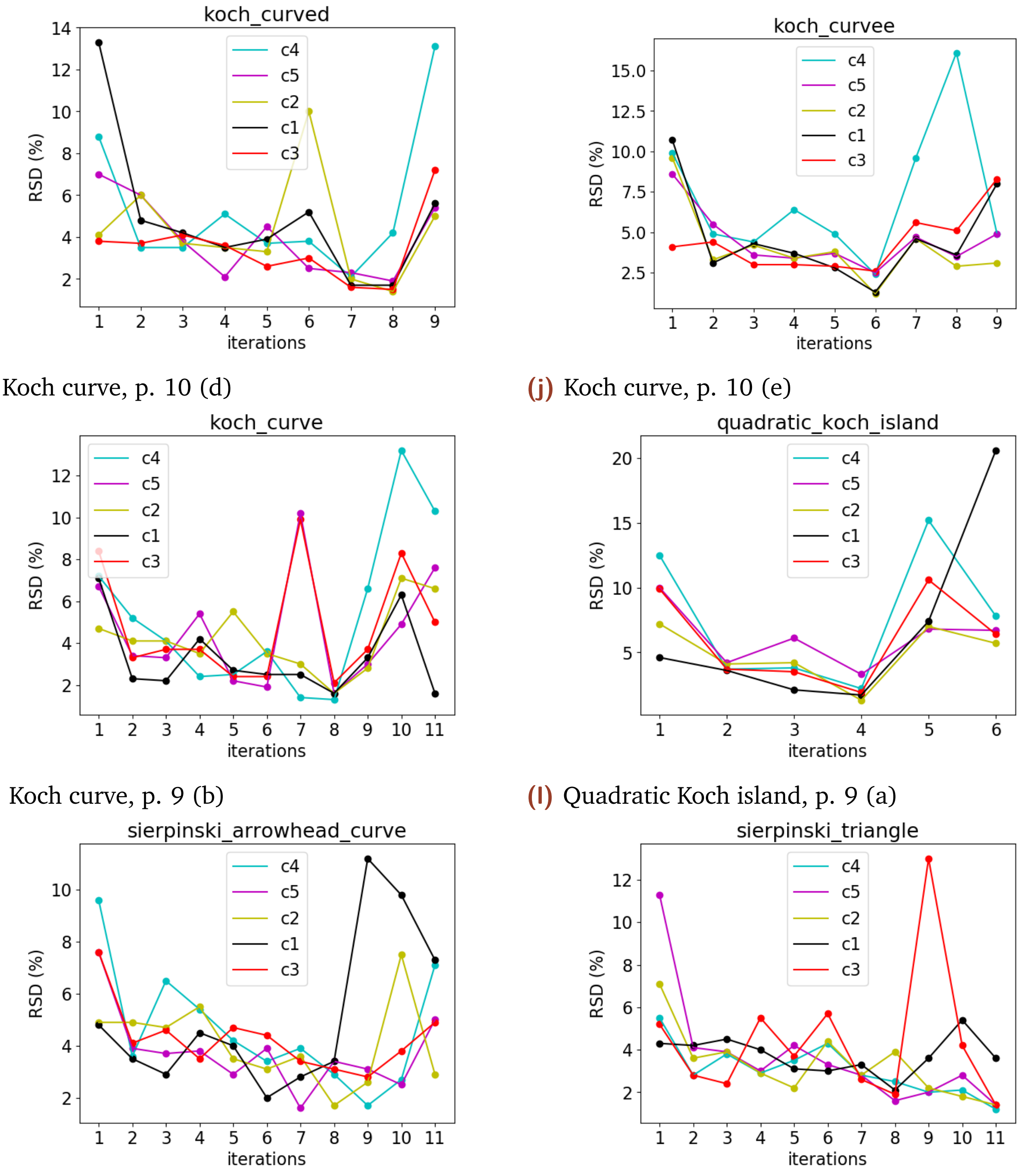
(d) Snowflake curve, p. 2

(e) Combination of islands and lakes, p. 9, fig. 1.8

(f) Fractal plant, p. 25 (f)

(g) Koch curve, p. 10 (b)

(h) Koch curve, p. 10 (c)



(i) Koch curve, p. 10 (d)

(j) Koch curve, p. 10 (e)

(k) Koch curve, p. 9 (b)

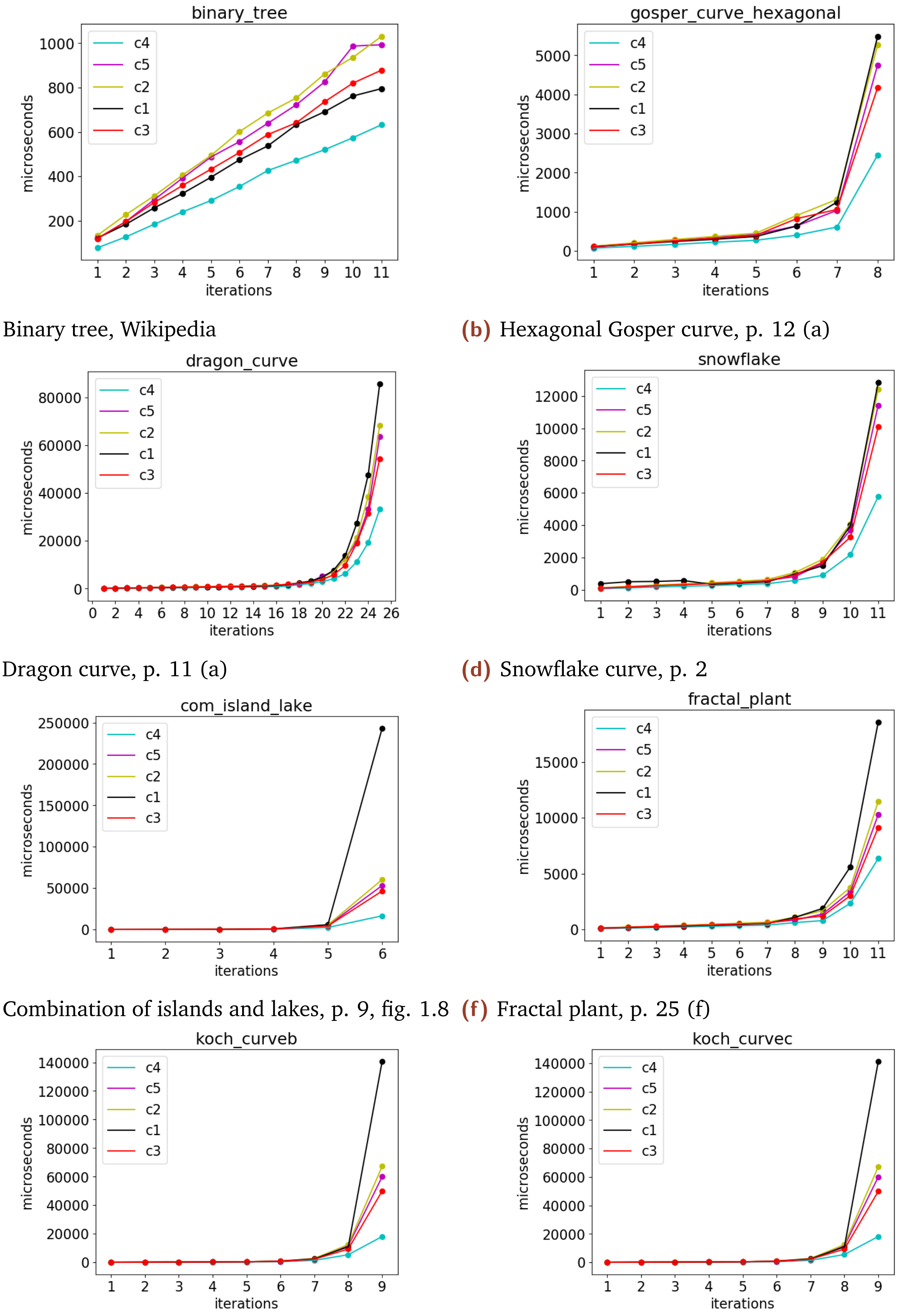
(l) Quadratic Koch island, p. 9 (a)

(m) Sierpinski gasket, p. 11 (b)

(n) Sierpinski triangle, Wikipedia

Figure 8.2: Relative standard deviation on the derivation of L-systems with references to their descriptions: Wikipedia page about L-systems (Wikipedia, n.d.) or Prusinkiewicz and Lindenmayer, 2004. The graph cX uses strategy number X.

8.4 Time to Generate Symbols



(a) Binary tree, Wikipedia

(b) Hexagonal Gosper curve, p. 12 (a)

(c) Dragon curve, p. 11 (a)

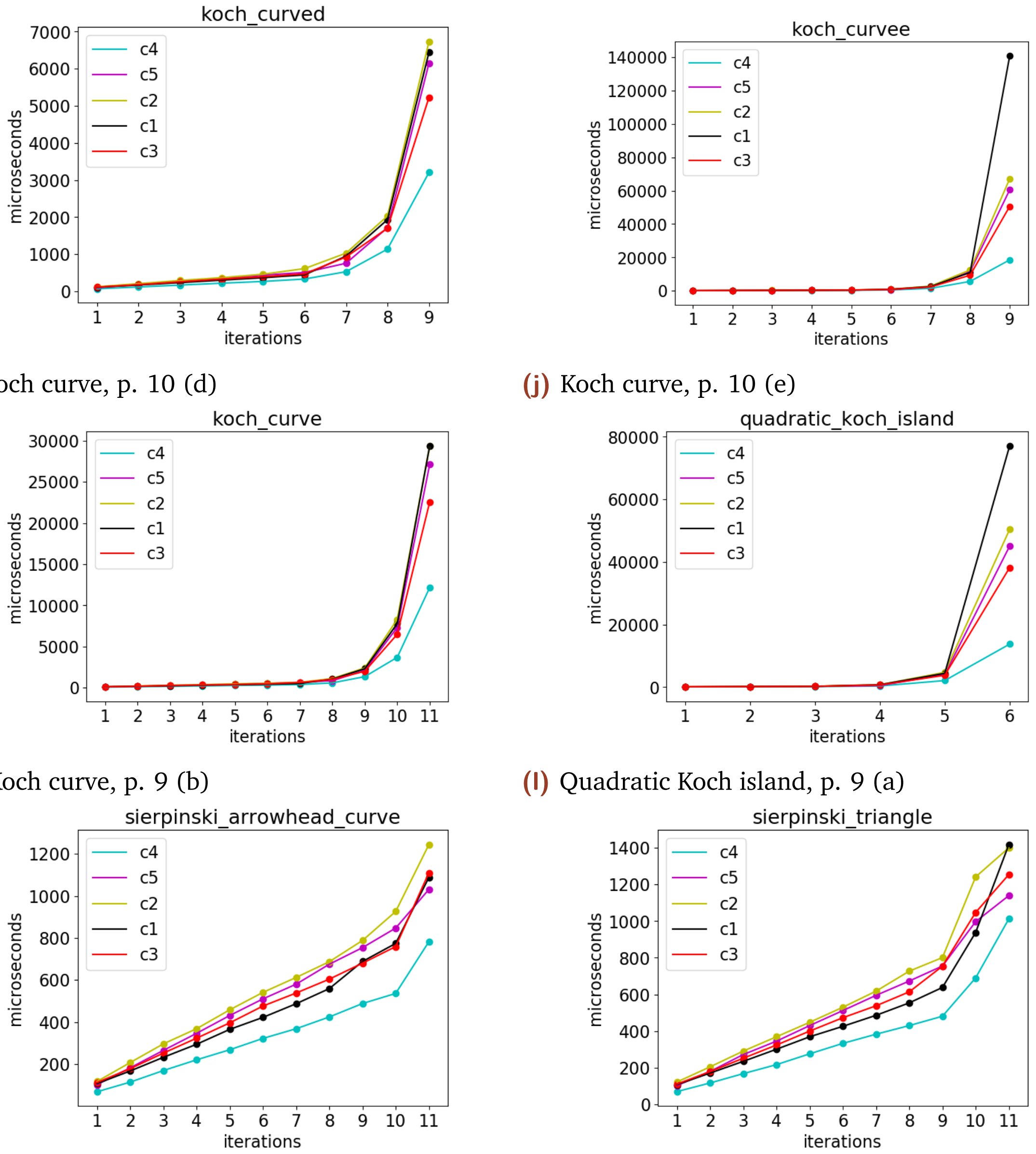
(d) Snowflake curve, p. 2

(e) Combination of islands and lakes, p. 9, fig. 1.8

(f) Fractal plant, p. 25 (f)

(g) Koch curve, p. 10 (b)

(h) Koch curve, p. 10 (c)



(i) Koch curve, p. 10 (d)

(j) Koch curve, p. 10 (e)

(k) Koch curve, p. 9 (b)

(l) Quadratic Koch island, p. 9 (a)

(m) Sierpinski gasket, p. 11 (b)

(n) Sierpinski triangle, Wikipedia

Figure 8.3: Time to generate symbols for different L-systems with references to their descriptions: Wikipedia page about L-systems (Wikipedia, n.d.) or Prusinkiewicz and Lindenmayer, 2004. The graph cX uses strategy number X.