Parallel Generation of Multiple L-Systems

Markus Lipp[a], Peter Wonka[b], Michael Wimmer[a]

[a] *Vienna University of Technology*
[b] *Arizona State University*

Abstract

This paper introduces a solution to compute L-systems on parallel architectures like GPUs and multi-core CPUs. Our solution can split the derivation of the L-system as well as the interpretation and geometry generation into thousands of threads running in parallel. We introduce a highly parallel algorithm for L-system evaluation that works on arbitrary L-systems, including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects. This algorithm is further extended to allow evaluation of multiple independent L-systems in parallel. In contrast to previous work, we directly interpret the productions defined in plain-text, without requiring any compilation or transformation step (e.g., into shaders). Our algorithm is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free.

Keywords: L-systems, graphics hardware, parallel processing, real-time rendering

1. Introduction

Procedural modeling techniques to compute large and detailed 3D models have become very popular in recent years. This leads to the question of how to handle the increasing memory requirements for such models. The current trend is towards data amplification directly on the GPU, for example tesselation of curved surfaces specified by a few control points. This results in low storage costs and allows generating the complex model only when needed (i.e., when it is visible), while also reducing memory transfer overheads. In the same vein, grammars can be viewed not only as a modeling tool, but also as a method for data amplification since a very short grammar description leads to a detailed model.

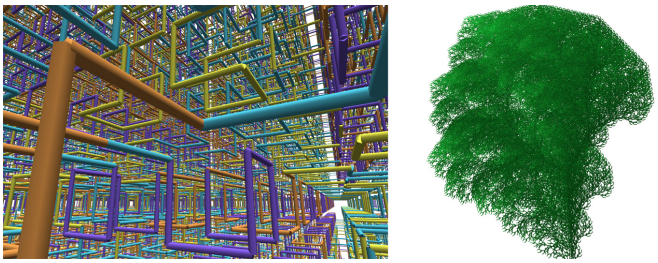

Figure 1: L-systems generated in real-time, at up to 198,000 modules per millisecond: Hilbert 3D space-filling curve and 2D plant.

In this paper we investigate whether it is possible to efficiently evaluate one of the most classical procedural modeling primitives, L-systems, directly on parallel architectures, exemplified by current GPUs and multi-core CPUs. The main motivation is to enable interactive editing of large L-systems (examples are shown in Figure 1) by designers, therefore it is important to speed up the computation of L-systems in order to achieve low response times.

Although L-systems are parallel rewriting systems, derivation through rewriting leads to very uneven workloads. Furthermore, the interpretation of an L-system is an inherently serial process. Thus, L-systems are not straightforwardly amenable to parallel implementation. Previous work has therefore focused on specialized types of L-systems that do not allow side effects in productions, which makes them very similar to scene graphs [3]. In contrast, we deal directly with uneven workloads in L-system derivation, and we have identified two main sources of parallelism in the interpretation of L-systems: (1) the *associativity* of traversal in non-branching L-systems, and (2) the *branching structure* itself in branching L-systems.

The main contribution of this paper is a highly parallel algorithm for L-system evaluation that

- works on arbitrary L-systems , including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects
- works directly on an input string and a plain-text representation of the productions without requiring any compilation or transformation step (e.g., into shaders)
- is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free

Email addresses: lipp@cg.tuwien.ac.at (Markus Lipp), pwonka@gmail.com (Peter Wonka), wimmer@cg.tuwien.ac.at (Michael Wimmer)

- parallelizes both within one L-system as well as among a large number of L-systems

To our knowledge, this is the first L-system algorithm that is highly parallel, i.e. utilizes thousands of threads in an efficient manner. This is achieved by identifying and exploiting the parallelism inherent in L-system derivation using parallel programming primitives like scanning or work-queue management, and a novel algorithm to explicitly resolve the branching structure. We demonstrate that our algorithm outperforms a well optimized single-core CPU implementation on larger L-systems.

This paper is an extended version of [5], adding support for multiple L-systems as described in Section 6.

Overview: First we will provide a background on L-systems and parallel primitives in Section 2. An analysis of the intrinsic parallelism of L-systems is provided in Section 3. Then our system consisting of two major building blocks will be described: (1) The derivation step will start with the axiom and generate a long string of modules (Section 4). (2) The interpretation step takes the string as input and generates the actual geometry (Section 5). An extension to support multiple independent L-systems in parallel is shown in Section 6.

1.1. Previous Work

General L-Systems: Prusinkiewicz and Lindenmayer cover the basic L-system algorithm [9]. Multiple extension to the basic approach were introduced [10, 11, 7].

Parallelizing L-Systems: Lacz and Hart showed how to use manually written vertex and pixel shaders combined with a render-to-texture loop to compute L-systems [3]. This concept was later extended using automatically generated geometry shaders [6]. Both methods require a shader compilation step for the productions. Further a transformation step of every production's successor to a set of successors is needed to allow independent parallel executions in a shader. For example, the production $L \rightarrow aLf[+L]Lf[-L]L$ is transformed to the set $L \rightarrow aL, af+L, afL, aff-L, aff-L, affL$ [3]. This is only valid if the successor of L does not have any effect on the traversal state, which is not generally the case.

An algorithm utilizing multiple processors (the results show up to 8 CPUs) with distributed memory, communicating using the Message Passing Interface (MPI) was introduced [14]. In their algorithm, the derivation of the L-system is performed using two binary trees, a Growth-State Tree (GST) and a Growth-Manner Tree (GMT). To actually render the system, the GST is interpreted as a scene graph. In order to get global scene-graph transformation matrices needed for rendering in the individual threads, the matrices are serially transfered from one process to the next.

Parallel Computation in CUDA: In order to access the parallel computing capabilities of GPUs we employ the NVIDIA CUDA data-parallel programming framework [2]. Recent work shows how to map computations having a highly dynamic nature to CUDA. Most notably, algorithms to efficiently implement workload balancing using a compactation step were introduced in the context of KD-trees [15], Reyes-style subdivision [8] and bounding volume hierarchies construction [4]. Generalized stream compaction was presented by Billeter et al. [1]. In the context of tessellating parametric surfaces, scan operations were used in order to scatter dynamically generated vertices to a VBO [12]. We employ both work-load balancing and vertex scattering in our work.

2. Background

Our work is based on L-systems and parallel processing primitives. Both concepts will be explained in this section.

L-Systems. In our work, we use the formalism of parametric L-systems as introduced by Prusinkiewicz and Lindenmayer [9]. Parametric L-systems operate on *parametric words*, which are strings of *modules* consisting of *letters* with associated *actual parameters*. An L-system consists of a parametric word ω called the *axiom*, and a set of productions describing how the current word is transformed. A production consists of a letter possibly combined with *formal parameters*, called the *predecessor* and a *successor*. The successor consists of a list of letters, where each letter can have multiple *arithmetic expressions* containing formal parameters. Formal parameters can be global or local to one production rule. The real-valued *actual parameters* appearing in the words are calculated from the arithmetic expressions of *formal parameters*. The predecessor can also consist of several letters, in which case the L-system is called context sensitive [9].

In the following example, F, A, and B are the letters defining modules, g_i are global parameters, l is a local parameter, and the arrow separates the predecessor from successor:

$$F(l) \rightarrow A(l * g_1)[B(l + g_2)]$$

To actually generate geometry, two distinct phases are performed: A *derivation* phase generating a string of modules, and an *interpretation* phase in which the string of modules is interpreted in order to generate geometry.

Derivation: The derivation starts from the axiom. For every module contained in the axiom, a *matching* production is searched. A production matches a module m if the letter of the predecessor matches the module letter, and the number of actual parameters in the module equals the number of formal parameters in the production. We then *apply* the matching production to the module: First, for every module in the successor, we calculate the actual real-valued parameters from the arithmetic expression of the formal parameters. Then we *rewrite* the module m with the modules of the successor. One *iteration* consists in rewriting all modules in the string *in parallel* using matching productions [9]. A user-defined amount of

iterations is performed in order to get the final string of modules.

Interpretation: The interpretation is performed *serially* from the start of the string, performing modifications of a *turtle state* based on predefined *turtle commands* associated with specific letters [9]. The turtle state represents the position and orientation of a virtual turtle. This state can be represented with a 4x4 matrix. The turtle commands associated to letters modify the turtle state, for example 'F' moves the turtle forward while drawing a line, or '+' rotates the turtle. Most of these turtle commands can also be expressed by a 4x4 matrix. A notable exception are the commands '[' and ']', which push and pop the turtle state on a stack, allowing the creation of *branching* (also called *bracketed*) L-systems [9].

Parallel Primitives. We extensively use the parallel *scan* primitive in our work. Given an ordered set of values $[a_0, a_1, \ldots, a_n]$ and an associative operator $\circ$ with the identity element I, an exclusive scan operation will result in the ordered set $[I, a_0, a_0 \circ a_1, \ldots, a_0 \circ a_1 \circ \ldots \circ a_{n-1}]$ [13]. If the operator is the addition, this results in a set of values s_i with $s_i = \sum_{j=0}^{i-1} a_j$. The main advantage of the scan primitive is its capability to compute seemingly serial operations very efficiently on highly parallel hardware, since subsequences can be processed independently due to associativity. Unless noted otherwise, we always refer to an exclusive scan on integral values using the addition operator when we use the term scan in our work.

3. Analysis of Parallelism in L-System

3.1. Derivation

As an L-system is by definition a module string rewriting system utilizing *parallel* module replacements, the domain of parallelization is obvious: We simply assign chunks of the modules uniformly to multiple threads and perform the rewriting in parallel. The rewritings themselves are independent and thus do not need inter-thread communication. However, the output strings need to be concatenated again, which creates a dependency between the threads. The major problem here is that the length of these strings can vary greatly: for a chunk containing n modules, the minimum expanded module amount is n. This case occurs when no production can be applied and thus every module is copied unmodified to the output. However, the maximum amount of modules is m^n, when the production with the maximum amount m of modules in the successor gets applied to each module.

Therefore, a parallel implementation has to efficiently cope with highly incoherent output module counts for each chunk. Previous shader-based approaches rely on the graphics pipeline to handle concatenation by load balancing (i.e. different output sizes of the geometry shader), which is not ideal because it can lead to serialization, and only works for special types of L-systems. In chapter 4 we show a native parallel solution to this problem utilizing the scan primitive.

3.2. Interpretation

The interpretation of a derived word is defined in a serial manner: Starting with an initial turtle state from the beginning of the module string, the position in the module string is advanced one by one, while applying a modification to the turtle state as defined by the letter of the current module. Therefore, the turtle state of every module string position is dependent on all previous turtle states. While it may look like there is no parallelism to exploit here, there are two inherent parallel concepts that can be extracted, as shown next.

Associative Operations: As mentioned before, most turtle commands and the turtle state can be represented as 4x4 matrices, except the push and pop commands. Further, as 4x4 matrix transformations can be combined, we can represent the turtle state up to a specific module string position using one matrix. The key point to parallelize the interpretation is to exploit the *associativity* of those matrix multiplications by accumulating matrices in each parallel chunk locally and combining them in a separate pass using a scan operation, as described in Section 5.1.

Inherent Branch Hierarchy: Since push and pop commands cannot be represented as matrices, the matrix approach cannot be applied for branching L-systems. Fortunately, the push/pop commands create another type of implicit parallelism that we can exploit: Every time a module representing a push command is encountered, two independent interpretation branches are possible: the module string directly following the push command, and the module string following the corresponding pop command. Thus we can split the work at this point into two threads, as shown in Section 5.2.

3.3. Multiple L-Systems

Another possibility for parallelism is deriving and interpreting multiple independent L-systems in parallel, for example interpreting multiple trees in a forest. A trivial approach to achieve this is to launch independent threads for every L-system. However, there are two problems with this approach: First, CUDA does not allow running different programs (also called kernels) in parallel. Every thread has to use the same program with possibly different input data. Therefore, in order to interpret different L-Systems in parallel, we need one flexible program that is able to handle all input L-systems. Second, we also want to achieve work-load balancing between the different L-system threads, so there has to be some communication between them. In Section 6 we show how to solve those problems in a data-parallel fashion.

4. Parallel Derivation

First we show how productions and module strings are efficiently represented on the GPU. Then we introduce the

algorithm to perform one iteration of the derivation.

4.1. Efficient L-system Representations

$$F(l) \rightarrow A(l*g_1)[B(l+g_2)]$$

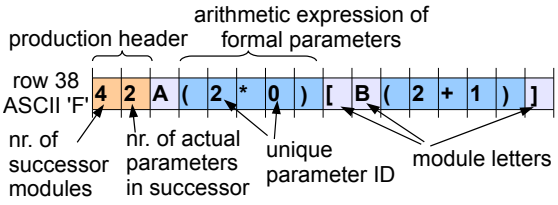

Figure 2: An example production of a parametric L-system packed in a texture.

In order to allow fast and efficient access to the productions, we store them in a 2D texture in the GPU version. The global parameters are also stored in a texture. For the multi-core CPU version we use a 2D array.

The successor is stored in the row indexed by the ASCII-value of the predecessor's letter. To resolve collisions of two productions that have the same predecessor letter, we create collision chains similar to hash tables.

We perform two simple optimizations: First, we count the number of modules and parameters occurring in the successor for later reference. Those values are stored in a designated header area. Second, in order to allow $O(1)$ parameter value lookup during the derivation, we translate every parameter to a unique numerical ID. To differentiate between local and global parameters, we define that all IDs over a certain threshold l_t identify local variables. This process is visualized for parametric L-systems in Figure 2. In order to store stochastic or context sensitive productions, we extend the header area accordingly, by adding the production probability or respectively the left and right context letters.

Representing the Module String: A module string contains n module letters. As every module may have an arbitrary amount of parameters assigned, we use an additional array of size n containing indices to an array of actual parameters. One advantage of this separation is that we can simply ignore the parameter array when we have an unparameterized L-system, thus removing the overhead of parameter storage. Figure 3 visualizes one module string.

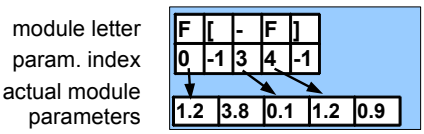

Figure 3: A module string represents a specific state during derivation. We store it as an array of module letters, a parameter index and the actual parameters.

4.2. Derivation

First, we prepare the axiom module string on the CPU side. In the GPU version we then upload it to the GPU. This step is extremely fast, as the axiom usually consists of just a few modules. For the desired iteration amount, we perform one iteration after the other on the GPU or the multi-core CPU. One iteration of the derivation takes the current module string as input and creates an expanded output module string. Between the iterations we swap the pointer to the input module string with the output module string.

The method to compute one iteration in parallel consists of three passes (or "kernels" of the parallel programming language) (see Figure 4):

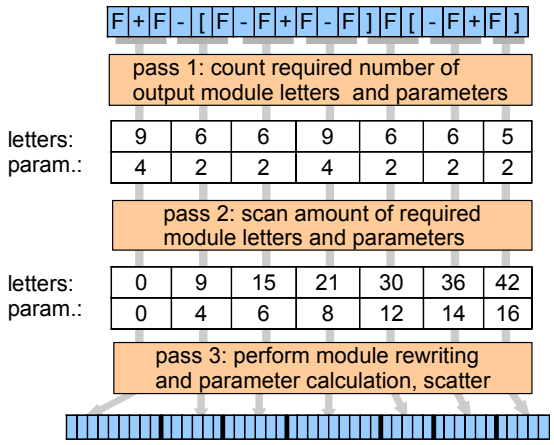

Figure 4: Three passes are performed during each iteration, transforming an input module string to an output module string. For better readability, we show only the letters of the input modules, and omit the parameters.

1. Count. We launch a kernel with n threads. $m = inputSize/n$ subsequent modules from the input module string are assigned to each thread. Each thread visits all m assigned modules, and fetches the amounts of required output module letters and parameters from the header section of the corresponding production. Those amounts are accumulated for all assigned modules, and finally written to an array in global memory.

2. Scan. We perform a sum-scan operation on this array, in order to calculate offset positions for the scattering of the result.

3. Rewrite. Again, m threads are launched, but this time the module replacement and parameter calculation is actually performed. This is done by fetching for each assigned module the matching production. If a production is *context-sensitive*, we also compare the left and right module letters of a module with the letters stored in the production header in order to determine if the production is applicable. There is no problem in performing a context search across chunk boundaries, as the whole input module string is stored in global GPU memory, and is thus visible to all threads. For *bracketed context sensitive* L-systems the context search is more involved, as we need to

take the push and pop commands into account. Therefore, for those systems, we perform a parallel hierarchy extraction step as explained in Section 5.2 before we start one iteration. For *stochastic* productions, we determine a random value for every applicable production, using a texture containing random values indexed by the position in the module string. This value is multiplied with the probability stored in the rule header. We then choose the rule with the highest result of the multiplication.

After having decided which production to use, we evaluate the parameters for every module in the production's successor, and insert the resulting successor modules into the result module string. The parameter evaluation based on the arithmetic expression of formal parameters is conducted by a simple mathematical expression parser in the kernel. When no production is defined for a module, we simply copy it unmodified to the output. As we have the offset values to index the module string, every thread can write its resulting modules without interference from the other threads.

Alternative Method: Our algorithm requires three passes for each iteration. An alternative approach is to implement the module string as a linked-list of modules, which is modified with atomic operations. This requires only one pass. The amount of atomic operations can be reduced by using batched linked lists, where each element contains multiple modules. However, when we implemented this alternative approach it turned out to be considerably slower than the three-pass approach, probably caused by the implicit serializations occurring on concurrent atomic operations.

Possible Enhancement: Our algorithm does not use shared memory between threads on the same multiprocessor for communication. Maybe applying the shared-memory aware compaction model presented by Billeter et al. [1] could further improve performance.

5. Parallel Interpretation

The result of a derivation is a module string. This needs to be converted into a geometric representation. There are two cases allowing two different parallel algorithms: non-branching and branching L-systems.

5.1. Non-Branching Module Strings

As explained in Section 3, most modules can be represented as associative matrix transformations. We can exploit this efficiently to interpret non-branching L-systems by interpreting chunks independently. We present a three pass algorithm (see Figure 5):

1. Matrix accumulation The string is split into m chunks, each chunk is assigned to an independent thread. In each chunk, we combine the matrices corresponding to the modules in the chunk, resulting in one local transformation matrix. Further, we count the amount of geometry generated in the modules. Both values are stored in an global array.

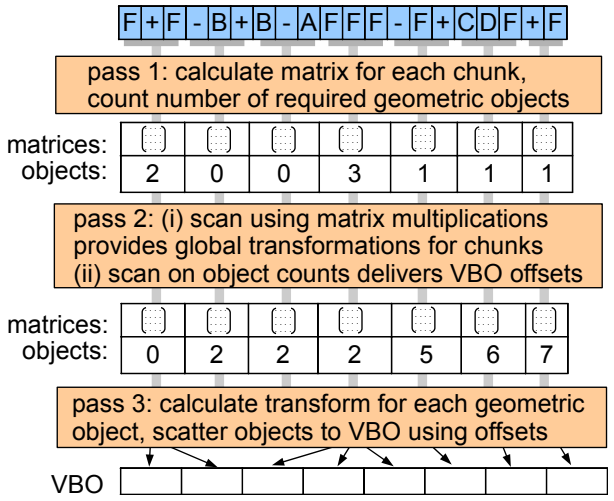

Figure 5: By exploiting the associativity of modules representable as matrix operations, we can efficiently interpret non-branching L-systems with this algorithm.

2. Matrix scan. A parallel scan operation is performed on the matrices, using 4x4 matrix multiplication as the operator. The resulting array contains matrices representing a global transformation of the turtle state to the start of each chunk. Additionally, a scan using integer additions on the objects array is performed to calculate offsets for the geometry to be created.

3. Geometry generation. Finally, to get the global positions of each geometry object, we again operate on m chunks in parallel as in 1. and accumulate module matrices. But this time we do not start with the identity matrix but with the matrix determined during the scan. Furthermore, every time we encounter a geometry generation module, we calculate the global position of the object and insert it into a vertex buffer object (VBO), using the offsets determined during the object scan.

The idea to use a scan to determine VBO offsets for tessellating parametric surfaces was introduced by Schwarz and Stamminger [12].

5.2. Branching Module Strings

For branching L-systems, parallelization is achieved by exploiting the branch hierarchy. Whenever a push command opening a new branch is encountered, two independent new work items are generated: one for the branch and one for the remaining string following the corresponding pop command. The main problem is to find the pop command in an efficient (i.e., parallel) way. This information is also necessary for fast context search in bracketed context sensitive systems. We therefore first present a novel parallel algorithm to extract the hierarchy, and then show how the work items can be efficiently managed.

5.2.1. Parallel Hierarchy Extraction

One critical observation is that when looking only at a particular hierarchy depth in the branch hierarchy, corresponding push/pop pairs follow each other directly. The

main idea is therefore to extract the push and pop commands from the module string and sort their positions into buckets according to their depths. Each bucket will then contain the positions of corresponding push/pop pairs. These can then be easily traversed to store with each push the position of the corresponding pop.

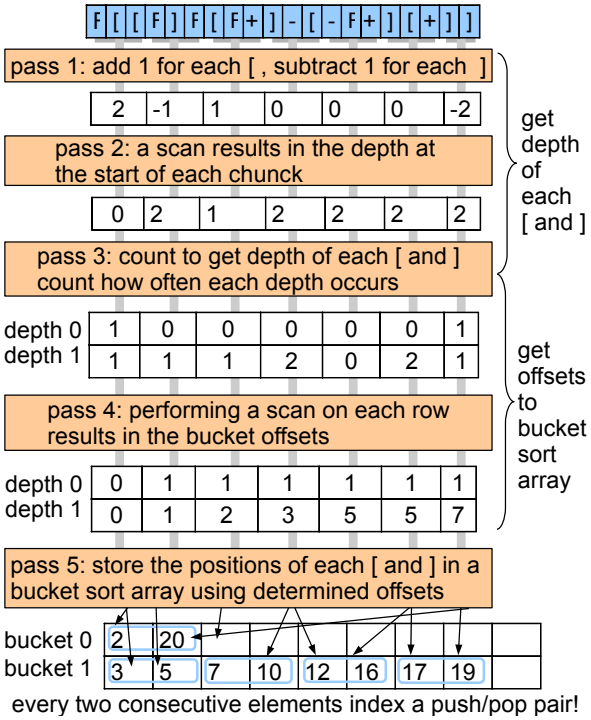

Figure 6: This algorithm allows efficient and parallel searching for corresponding push and pop pairs.

We introduce an efficient parallel algorithm based on this idea that does not require direct communication between the blocks. We assume that we know the maximum depth of d_{max} of the branching hierarchy and allocate a two-dimensional bucket sort array with d_{max} rows. The complete algorithm is visualized in Figure 6, and consists of 5 passes operating on uniform chunks in parallel:

1. Chunk depth calculation. Starting at zero, we add 1 for every push, and subtract 1 for every pop occurring in a chunk. This results in the depth of the chunk end relative to the chunk start.

2. Depth scan. Performing a scan of those values results in the absolute depths of the start of each chunk.

3. Depth-based push/pop count. Now, by starting at the calculated absolute depth of the chunk start, we can determine the absolute depth of every push and pop occurring in the chunk. We use this to to determine the offsets for the bucket sort array by counting the amount of push and pop commands m_i in each depth j in the following way: $c_j = \sum m_i \mid d_i = j$. We store the values for $c_j \mid 0 < j < d_{max}$ in a global array.

4. Scan push/pop counts. The scan of the c_j arrays results in the bucket offsets o_j each thread has to use in order to allow conflict-free writing to buckets.

5. Write push/pop locations. Again, we visit every push and pop command m_i of every chunk. But this time we write the absolute input module string position of the module m_i in the bucket d_i using the offset positions determined previously. This ultimately leads to a bucket sort array where every two consecutive elements in a row correspond to a push and pop pair.

Analogously to the matrix interpretation algorithm, we also calculate VBO offsets needed to scatter the geometry. In our implementation, this process is combined with pass 1, and the VBO offsets are stored as a parameter of push commands.

Memory Footprint: In order to reduce the memory footprint of the bucket sort array, we actually use a one-dimensional array instead of two dimensions. This allows us to pack the bucket arrays for the individual depths without empty spaces tightly together. In the worst case, when every module in the module string is a push or pop module, the number of required memory elements is then equal to the amount of modules. The offsets needed for the 2D to 1D packing can simply be calculated form the values obtained in iteration 4: $c_j + o_j$ of the last chunk equals the total amount of elements in a specific bucket. When we perform a scan operation of those values for each bucket we get the offsets to map the 2D bucket arrays into a 1D array.

Integration into Module String: As a last step, we use the information bucket arrays to write the position of corresponding push and pop modules directly into the module string to allow fast access during interpretation. This is a simple parallel algorithm: We evenly assign the 1D bucket array to multiple threads. Every even element in this array contains the position of a push command, every odd element references a pop command. Thus we need to write the position stored at every odd element as a parameter to the push module referenced by the preceding even element.

5.2.2. Work Queue-based Interpretation

As a result of the previously explained algorithm, every push module has a parameter indexing the position of the corresponding pop module, as well as a VBO offset parameter. With this information, we use a parallel work-queue approach [15, 4, 8]:

First, we define a work item as a tuple (M, i) where M is a 4x4 matrix representing the turtle state, and i is the array index of a module in the module string. The interpretation is started with the work-queue item $(I, 0)$, where I represents the identity matrix. Then, one thread starts serially interpreting the module string using this work item. When a push-module is encountered, the thread creates two work items (M, i_1) and (M, i_2), where M is the current turtle state, i_1 is the current module index and i_2 is the index of the corresponding pop command. Then it puts (M, i_1) on a thread local work stack and continues with (M, i_2). Directly continuing with (M, i_2) is important

for good performance, as it induces a breadth-first traversal (with respect to the branch hierarchy) which enables faster spreading of the work items to multiple threads.

To actually distribute the work items between threads, we use parallel work queue management [15] [4] [8] in the following adapted way: When the local work stack of a thread is full, we write the contained items to a work item array in global memory. Each thread is assigned a unique offset to this array. When all threads are finished, we create a compact array of indices to the work item array using scan operations. These indices are then evenly distributed to threads, which execute the tasks as described previously. The process is iterated until no more work items are left.

Alternative approach: In our approach each thread has an independent local work item queue. During kernel execution, no intra-block distribution of work is performed using the shared memory. We have also implemented and tested a version utilizing shared block local memory to distribute threads. However, this method turned out to be slower, probably the overhead of the multiple required intra-block synchronization points is higher than the possible gain achieved through faster work distribution.

6. Multiple L-systems

The algorithms presented in the previous sections always work on one L-system at a time. However, it would be beneficial to derive multiple L-systems with different parameter sets in parallel at the same time. An example application for this is designing procedural forests using L-systems, as shown in Figure 7. Therefore we introduce extensions to the previously introduced algorithms, allowing the derivation of multiple L-systems in parallel while still retaining work-load balancing between threads.

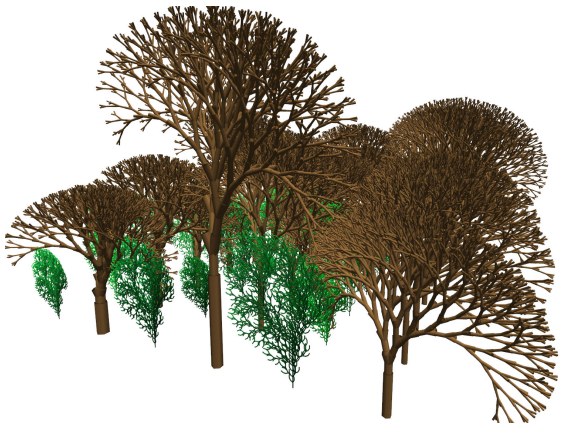

Figure 7: Here multiple different L-systems with unique parameters are shown. The derivation and interpretation can still be performed interactively.

6.1. Representation of Multiple L-systems

For every L-system L_i with the unique identifier i, we need to define a unique set of productions, global parameters and an initial turtle state T_i. In Section 4.1 we showed how one L-system production and global parameter set is stored in a texture. In order to store multiple L-systems, a unique offset O_i into this texture is assigned to every L-system. The offsets can be trivially calculated by summing the amounts of productions and parameters of every L-system, and ensuring that the values stored in the texture do not overlap.

In order to access this representation during derivation, a lookup table from L_i to O_i and T_i is created and stored in global memory.

6.2. Derivation of Multiple L-systems

Before the derivation algorithm is started, we connect the axioms of every L-system to one combined axiom. At the start of every axiom a delimiter symbol (we have arbitrarily chosen 'M') is inserted. In order to keep track of which letter belongs to which L-system during derivation, the following algorithm is performed before every derivation iteration: First we assign the letters uniformly to threads. Then every thread counts the amount of delimiter symbols in its assigned chunk. In a second pass, an additive scan is performed on those values, resulting in the L-system index of the first letter for each chunk. This algorithm is outlined in Figure 8.

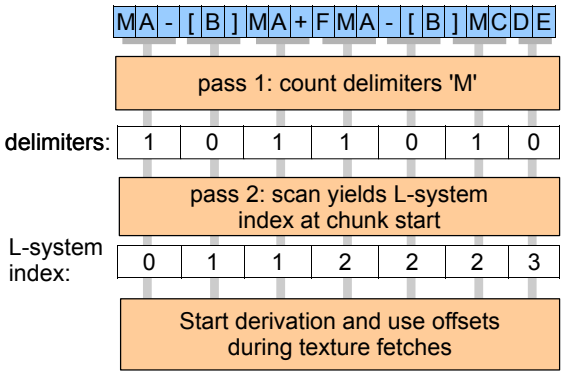

Figure 8: In order to determine which L-system a letter belongs to, this algorithm is performed before every derivation iteration.

Afterwards the derivation is performed. Every time a rule or parameter needs to be fetched, we determine the texture offset from the look up table using the L-system index. When a delimiter symbol is encountered, the index is increased by one. In the last pass of the derivation, we also write the positions of the delimiter symbols in the module string to a global array.

Different Iteration Amounts:. In order to support different numbers of iterations for the L-systems, the number of iterations to be performed is also stored in the look up table. This value is compared with the current iteration number during the derivation. When it is higher, no rule replacement is performed for this L-system.

6.3. Interpretation of Multiple L-systems

For non-branching L-systems, the algorithm in Section 5.1 needs to be modified. Instead of a scan of all matrices, a segmented scan of the matrices needs to be performed. The segments are defined by the delimiter symbols. The matrix of the first letter after a delimiter symbol is set to the initial turtle state stored in the lookup table.

As we know the string positions of each delimiter symbol after the derivation, the adaptation of the branching interpretation algorithm described in Section 5.2.2 is straightforward. We simply create a work item for every delimiter symbol. In every work item the L-system index is stored, and the turtle state is set to the one in the look up table. Then the interpretation algorithm is performed as described in Section 5.2.2.

7. Results

We implemented our parallel algorithms for GPUs utilizing CUDA and for multi-core CPUs using `POSIX` threads. In CUDA, up to 1920 threads are utilized (60 blocks of 32 threads), the multi-core CPU version uses 4 threads. We compare those implementations against a highly optimized single-core CPU version, created with the help of performance profilers to detect and remove bottlenecks. This version has the advantage that no kernel or thread launch overheads occur, and that no scan or multi-pass operations are necessary. The main low-level optimizations include the complete avoidance of advanced C++ features like virtual functions and dynamic memory allocations during runtime. Those optimizations resulted in a speedup of multiple orders of magnitude compared to our initial single-core CPU implementation. The test platform was an Intel Core 2 Quad Q6600 2.4GHZ PC with a Geforce GTX 280 graphics card.

L-system	bracketed	parametric	stochastic	context sens.
Hilbert 3D, pg. 20				
Koch curve, pg. 10 (d)				
row of trees, pg. 48		✓		
2D plant, pg. 25 (c)	✓			
3D tree, pg. 60 (b)	✓	✓		
plant stochastic, pg. 28	✓		✓	
p. ctx. sens., pg. 35 (b)	✓			✓

Table 1: Property matrix of the L-systems shown in our results. The page numbers refer to the L-system definitions by Prusinkiewicz and Lindenmayer [9].

Test Scenes: We used seven test scenes to demonstrate several aspects of our system. In order to ensure repeatability and comparability of our results, all our L-system productions are directly taken from Prusinkiewicz and Lindenmayer [9] for our performance measurements. In Table 1 we classify the test scenes according to the properties of the used production set.

Rendering: Our implementation creates a VBO containing lines. For our figures we use a geometry shader during rendering, creating cylinders from the lines. All our performance measurements do *not* contain the rendering times, as the rendering times are the same both for the CUDA and the CPU versions. Neither do the measurements contain the CPU-GPU memory transfer times required by the CPU versions, which we measured in the range from 20–40ms, making the CPU versions very hard to use in a real-time rendering setting.

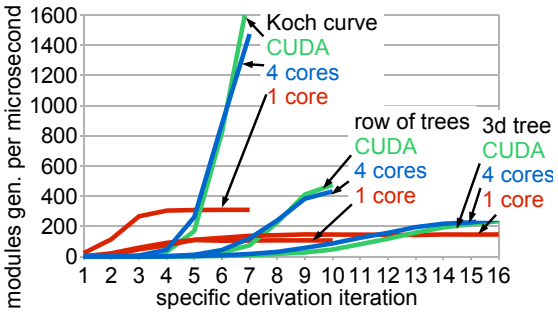

Figure 9: Scalability analysis of the derivation step. For every iteration, we calculate the number of modules generated per microsecond.

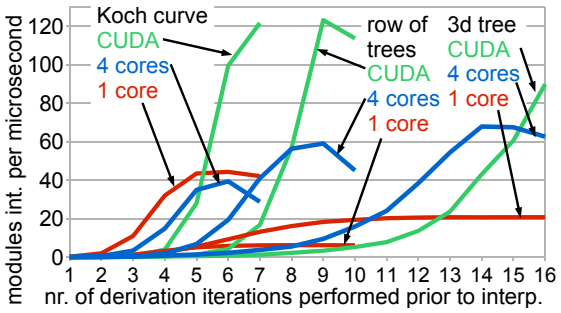

Figure 10: Scalabilty analysis of the interpretation step. We performed a specific amount of derivation steps before the interpretation was performed.

Scalability: We evaluated how our derivation and interpretation scale with the number of iterations. For all our test scenes, we measured how long one specific iteration i of the derivation takes, and calculated the number of modules generated per microsecond during each iteration i. The results for three L-systems are shown in Figure 9. For the interpretation, a specific amount of derivation iterations was performed prior to the interpretation, the interpretation time was measured, and the amount of modules interpreted per microsecond was calculated. The results are shown in Figure 10. For readability only three L-systems are shown, but all results show a similar pattern: As expected the initial iterations incur some overhead in the parallel implementation on the GPU and the multi-core CPU, because the amount of parallelism is

		total derivation times			total interpretation times					total speedup deriv.+interpr.	
		ms	rel. speedup		ms	rel. speedup					
L-system, i	modules	1 core	4 cores	CUDA	1 core	4 cores	CUDA	d_f	i_f	4 cores	CUDA
Hilbert 3D, 6	1,266,864	4.70	3.76×	2.92×	31.50	1.23×	6.61×	5	5	1.35×	5.68×
Koch curve, 7	915,049	3.45	3.26×	3.20×	22.56	0.70×	2.99×	6	6	0.78×	3.02×
row of trees, 10	815,545	10.21	3.29×	3.15×	77.04	4.25×	10.84×	8	7	4.11×	8.43×
2D plant, 7	813,169	3.31	3.04×	3.15×	22.45	1.27×	1.21×	6	7	1.37×	1.31×
3D tree, 16	622,334	8.53	1.40×	1.17×	31.44	2.78×	3.87×	13	13	2.30×	2.59×
plant stoch., 11	835,481	6.45	1.75×	3.23×	14.70	0.09×	0.24×	9	?	0.13×	0.33×
p. ctx. sens., 30	25,174	0.73	0.39×	0.03×	0.11	0.43×	0.11×	?	?	0.39×	0.03×
multi L-sys., 10	2,751,022	30.51	1.48×	1.31×	29.12	1.65×	2.22×	4	4	1.56×	1.64×

Table 2: Performance measurements. i shows the amount of iterations performed. The single-core CPU times are absolute values in milliseconds, the multi-core CPU and CUDA values are relative speedups compared to the single-core CPU values. d_f denotes the first iteration where CUDA is faster compared to the single-core CPU version during derivation, i_f is analogous for interpretation.

limited, and the overhead of launching CUDA-kernels or `POSIX` threads is a significant factor. This makes parallel versions slower on the first few iterations. For the later iterations the parallel implementations are several times faster, because a high amount of threads can be utilized. For all L-systems, we list the first derivation iteration d_f where CUDA is faster compared to the single-core CPU version, as well as the first interpretation of a string generated with i_f iterations where CUDA is faster in Table 2. The total performance including the cases where CUDA is slower will be discussed in the next two paragraphs.

Total Derivation Performance: The CUDA and the multi-core CPU version are very similar in performance and are significantly faster than the single-core CPU version in most cases. There are two notable exceptions: First, the 3D tree is only marginally faster. Second, the context sensitive plant is considerably slower. Both cases can be attributed to the following observation: The corresponding L-systems are growing rather slowly, compared to the other test cases. For the 3D tree 622,334 modules are created in 16 iterations, while the plant creates only 25,174 modules in 30 iterations. All other tested L-systems create more modules with a lower iteration count. Therefore the other L-systems have less relative thread launch overhead. To sum it up, during the derivation the parallel implementations are significantly faster when the L-systems grow relatively fast. Another important thing to note is that the results for the stochastic system vary with the random seed, our measurements were in a range of about +/- 20% for different seeds.

Total Interpretation Performance: The three tested non-bracketed (serial) L-systems are significantly faster in CUDA compared to the other versions, probably because the parallel matrix interpretation makes good use of the high arithmetic density of the GPU. On the other hand, the multi-core CPU version performs rather bad on those L-systems (with the exception of the row of trees L-system), probably the matrix multiplications and the memory bandwidth are the limiting factors here.

The results for the branching L-systems vary. The first thing to note is that the five-pass hierarchy extraction step requires considerably less time than the actual interpretation. For reference, a hierarchy extraction takes 3.3ms on one CPU core and 1.97ms in CUDA for the 2D plant. Our interpretation of the varying results is that the L-systems have different branching structures, which directly affect how effective our work-queue interpretation is: The 3D tree (Figure 11) has very regular branching, and is considerably faster to interpret with the parallel versions, while the 2D plant (Figure 1) exhibits more irregular branching, resulting in only a small speedup. The stochastic plant contains only a few long branches with many small ones attached (Figure 11), making it hard to spread the work to multiple threads. The context sensitive plant is even harder for the parallel algorithms to interpret, as the amount of modules is very low compared to the other cases. In summary, the non-bracketed L-systems are considerably faster in CUDA, while the bracketed L-systems create varied results based on the branching structure.

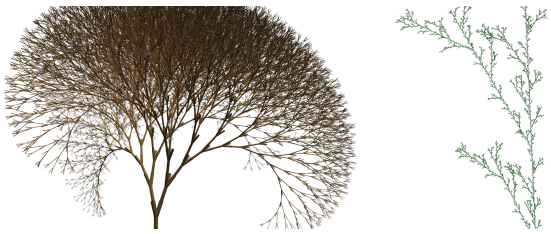

Figure 11: L-systems generated in real-time: 3D tree and stochastic plant.

Total Speedups: The combined derivation and interpretation speedups are shown in the last two columns of Table 2. In all but the context sensitive and multiple L-system case, the interpretation time is much higher than the derivation time, therefore the combined speedups are mainly dependent on the interpretation speedups.

Multiple L-Systems: The performance of multiple L-systems is tested using a scene consisting of 38 2D plants with 4 iterations and 12 3D trees with 10 iterations, similar as shown in Figure 7. The results are shown in the last row of Table 2. While the multi thread versions consistently

show a speedup of up to 2.22 times, it is not as high as we had expected. It is important to note that we did not optimize the multiple L-system implementation, while the single-core version is highly optimized. We believe that there is still room for improvement.

8. Discussion

Comparison to Previous Work: The main advantage over the previous GPU-based methods [3, 6] is that we make explicit use of parallel primitives and do not rely on the graphics pipeline to deal with data amplification and other issues. We fully support productions having side-effects and thus do not need to rely on the specific side effect-free turtle commands presented by Lacz and Hart [3]. Furthermore, we can directly use the productions without requiring a compilation or transformation step. Compared to the multi-CPU based method proposed by Yang et al. [14] our algorithm does not need an intermediate scene-graph representation of the module string. Furthermore our algorithm can utilize thousands of threads, which is significantly higher than what was shown in the multi-CPU version.

Memory Transfer to Graphics Hardware: One important advantage of our CUDA version is that the resulting geometry already resides in GPU memory, so there is no need for a copy operation. The CPU versions, on the other hand, needs to perform a copy from the main system memory to the GPU. We measured copy times of about 20–40ms for the tested L-systems – this is very high compared to the generation times, increasing the total speedup of CUDA significantly, and showing that a GPU implementation is highly desirable. All our results do *not* include those transfer times.

Intra-Block Thread Divergence: In CUDA, when different execution paths are taken within a sub-block of threads (called warp), those execution paths are serialized, reducing the utilization of the intrinsic SIMD capabilities. In our algorithm, the following situations lead to divergence in the code: (1) If two productions produce a successor of different length during derivation. This divergence is directly caused by the varying data amplification, and can probably not be avoided. (2) During interpretation, the matrix notation helps in maintaining thread coherence, as we can perform the multiplications coherently after each thread decides which matrix to use based on the associated commands. However, when some threads either have no command to perform or have a geometry generation command, SIMD can not be fully utilized. (3) In the work-queue algorithm, the number of elements in a local work queue can vary, leading to divergence. As pointed out, in an alternative approach we removed this divergence by enabling intra-block work item sharing, which turned out to be slower than having divergence. (4) The length of one work item can vary, leading to divergence. Unless we would further split work items into sub work items, we can probably not avoid this divergence.

Limitations: The varying results of the work-queue approach indicate that there may be future work necessary in creating more consistent speedups, maybe a more elaborate work-queue management can achieve this. As for the tested context sensitive L-system, the high iteration counts result in a low performance of the CUDA approach, making the use of the CPU version more appropriate in this case. Stochastic productions can be used to simulate biological plant-internal signals [9], maybe there is a way to abstract this simulation that is more amenable to parallel interpretation.

9. Conclusion

In this paper we introduced a solution to generate L-systems on a parallel architecture. We make two major contributions. First, we show how parallel primitives can be employed to handle the varying data amplification during derivation. Second, we introduce an algorithm to match the push and pop stack operations to obtain a parallel implementation of L-system interpretation. The system can work with a broad set of rules, including parametric rules, stochastic rules, and context-sensitive rules. It can parallelize a single L-system as well as multiple independent L-systems. We have demonstrated that our parallel L-system outperforms a highly optimized single-core CPU implementation in many test cases, while there are some cases where the single-core version is faster. The advantage of our GPU version gets more pronounced when taking into account CPU-GPU memory transfer times required by the CPU versions.

Future Work: We would like to integrate the parallel derivation of L-systems in a rendering engine to render large-scale environments. We plan to combine the derivation of L-systems with occlusion queries and memory management algorithms so that we can render environments several times the size of graphics card memory in real time. Also, it would be interesting to extend the work to procedurally generated architecture, and more complex L-system concepts.

Acknowledgements: This research was supported by the Austrian FIT-IT Visual Computing initiative, project GAMEWORLD (no. 813387), and by the NSF, contract nos. IIS 0612269, CCF 0643822, and IIS 0757623.

References

[1] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, NY, USA, 2009. ACM.

[2] NVIDIA CORPORATION. Cuda: Compute unified device architecture. http://developer.nvidia.com/. 2007.

[3] P. Lacz and J.C. Hart. Procedural geometry synthesis on the gpu. In *Workshop on General Purpose Computing on Graphics Processors*, pages 23–23, NY, USA, 2004. ACM.

[4] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[5] M. Lipp, P. Wonka, and M. Wimmer. Parallel generation of l-systems. In *Proceedings of the Vision, Modeling, and Visualization Workshop 2009*, pages 205–214, 2009.

[6] M. Magdics. Real-time generation of l-system scene models for rendering and interaction. In *Spring Conf. on Computer Graphics*, pages 77–84. Comenius Univ., 2009.

[7] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01*, pages 301–308, NY, USA, 2001. ACM.

[8] A. Patney and J.D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):1–8, 2008.

[9] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, Inc., NY, USA, 1990.

[10] P. Prusinkiewicz, M.J., and Radomír Mêch. Synthetic topiary. In *SIGGRAPH '94*, pages 351–358, NY, USA, 1994. ACM.

[11] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. In *SIGGRAPH '01*, pages 289–300, NY, USA, 2001. ACM.

[12] M. Schwarz and M. Stamminger. Fast gpu-based adaptive tessellation with cuda. *Computer Graphics Forum*, 28(2):365–374, 2009.

[13] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, pages 97–106, NY, USA, 2007. ACM.

[14] T. Yang, Z. Huang, X. Lin, J. Chen, and J. Ni. A parallel algorithm for binary-tree-based string rewriting in l-system. In *Proc. of the Second International Multi-symposiums of Computer and Computational Sciences*, pages 245–252, Los Alamitos, California, 2007. IEEE Computer Society Press.

[15] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.