Deriving Shape Grammars on the GPU

Mark Dokter*
Supervised by: Markus Steinberger and Michael Kenzel†

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

Due to growing demand for computer generated graphical content, procedural modeling has become an important topic in the gaming and movie industry. Creating vast amounts of content by hand requires excessive amounts of manual labor. Using a procedural rule set, entire worlds can be generated by a computer. However, the traditional CPU-based derivation of a large city can take multiple hours, making rapid design iterations impossible. In this paper, we investigate different strategies to execute procedural modeling on graphics processors using CUDA. We compare a persistent threads megakernel approach to simple kernel calls and different rule queuing strategies. Along these lines, we explore the trade-off between precompiling an entire rule set and interpreting a rule set online.

Keywords: GPGPU, megakernel, procedural modeling, rule derivaton, shape grammar

1 Introduction

Generating graphical content in an automated fashion has become increasingly important during the last decade. Many recent computer games offer vast open virtual worlds, where the player can freely explore the environment. In the latest version of Grand Theft Auto for example, the area is not confined to a single city, but also includes its suburbs, where one can walk, drive or fly. Movies like Lord of the Rings show huge battlefield scenery with thousands of warriors fighting on wide open plains.

Those are examples of extensive use of digitally created content, which requires vast amounts of manual labor to produce. By automating content creation as much as possible, artists can spend more of their time on elements relevant to narrative and gameplay, rather than on creating peripheral scenery.

The task of creating reoccurring, parameterizable objects like houses or trees is well suited to procedural modeling. Rules for model production can be defined in a shape grammar. Starting from an initial set of shapes, these rules iteratively add detail to the scene. After fully evaluating such a rule set, the geometry which describes the entire scene is ready for rendering. However, grammar derivation for thousands of buildings can take many hours on a conventional CPU, even with several cores and a high clock rate.

One way to increase performance is parallelization. Parallelizing tasks and algorithms has gained much popularity with the introduction of general purpose GPU computing. With many cores on a single chip, the performance of a GPU is unmatched by any CPU, assuming a suitable parallelizable task. Procedural geometry generation is such a task, which can be, if done carefully, parallelized and computed efficiently on a GPU.

As will be discussed in the section on related work, various attempts of mapping this challenging task to a graphics processor have already been made. This work focuses on exploring the benefits and drawbacks of deriving precompiled rule sets versus interpreting them at runtime. Furthermore, various methods of controlling the GPU rule evaluation process will be subject to testing. This includes launching several successive kernels as well as deriving the entire scene using a single kernel launch, using a persistent threads megakernel approach.

2 Related Work

Currently, the most widely used grammar for procedural architecture modeling is CGA [12]. CGA is based on Stiny's work on *shape-grammars* [16] and *set-grammars* [18]. Furthermore, it uses split operations for facade modeling as proposed by Wonka et al. [19] and transformation operations similar to *L-systems* [13]. Approaches augmenting the functionality and usefulness of shape grammars exist on more general non-terminal symbols [3] and mesh refinement [2]. Apart from grammar based approaches to the procedural generation of geometry, other methods can be used to obtain high quality models [6, 7, 11].

Parallel grammar derivation has been investigated in various approaches which differ greatly in their strategy. Deriving L-systems on CPU clusters has been done by [20]. Considering the inherent parallelism of the algorithm, CPU clusters seem to be a good idea. However, when using a GPU, the results are already in the memory of the graphics card which is obviously more convenient for rendering.

*dokter@icg.tugraz.at
†steinberger@icg.tugraz.at, kenzel@icg.tugraz.at

A recent L-system generator for the GPU has been proposed by Lipp et al. [8]. In their work, they used multiple kernel launches to implement iterative rewriting of L-systems. Using a single thread per symbol without sorting the symbol stream has some drawbacks. First, memory accesses can become problematic if symbol sizes are not coherent. Second, thread divergence, which is the effect of threads taking distinct execution paths, results in different times the threads need to finish their work. On a GPU, where it is desirable to have as many threads occupied as possible at any point during run time, this effect causes some threads to wait on others which might take longer. This drastically impacts performance. And third, the management overhead for keeping track of where to store symbols quickly becomes a dominant factor. Thus, for context sensitive grammars, the derivation process was even slower on a GPU than on a CPU.

Shader based derivation of split grammars has been proposed and investigated by Lacz and Hart [4], Magdics et al. [9] and Marvie et al. [10]. The method by Lacz and Hart uses a render-to-texture loop and imposes the main workload of the algorithm on sorting intermediate symbols—similar to the overhead found in L-system generator by Lipp et al. The method Magdics et al. also requires several rendering passes. It tries to prevent divergence by using a different shader for each output symbol. In our evaluation, we incorporate an approach inspired by their work, efficiently grouping output symbols and launching individual kernels for each symbol type.

The approach by Magdics et al. avoids multi-pass rendering by using a fixed size stack. Using a fixed size stack has multiple drawbacks. First, recursion depth is limited. Second, stack elements might be spilled to slow global GPU memory. Third, parallelism is limited to the number of axioms. And fourth, divergence can play a crucial role, if objects do not have identical structure.

An approach focusing on parallelizing grammar derivation for procedural modeling of architecture has been published by Steinberger et al. [15]. The *PGA* grammar is based on CGA[12] and uses a software scheduling GPU framework [14]. To avoid divergence, their approach groups shapes, which are to be processed by the same rule. Additionally, they draw parallelism from the rule itself. PGA compiles the entire rule set to achieve high performance rule derivation. The work reported in this paper is a direct extension of PGA.

3 GPU Split Grammars

Split grammars, introduced by Wonka et al. [19], are specialized set grammars, which impose restrictions on the allowed shapes and operations to make the grammar simple enough for automated derivation, but sufficiently expressive to allow the modeling of many different objects.

A split grammar builds on the notion of shapes and set grammars. A *shape* can be defined as follows [17]:

Definition 3.1 *A shape is a limited arrangement of straight lines in three-dimensional Euclidean space.*

Split grammars operate on a set of *basic shapes*, which can have attributes, can be parameterized and labeled. These basic shapes form the core buildings blocks of split grammars. Examples for the geometry represented with basic shapes are boxes, spheres, cylinders, rectangles, etc. The parameters of these basic shapes define their extent, their position, etc. The label associated with the shape is often called symbol. This symbol can either be a terminal symbol $\in T$ or a non-terminal symbol $\in N$.

A *grammar* can be defined as a set of production rules R on a set of symbols U, using the following definition similar to the one given by Wonka et al. [19]:

Definition 3.2 *A grammar $G = (N,T,R,I)$ consists of the non-terminal symbols $N \subseteq U$, the terminal symbols $T \subseteq U$, a set of initial symbols (axioms) $I \subseteq N$ and a set of rewriting rules (productions) $R \subseteq U \times U^*$.*

A *rule* $a \longrightarrow B$ in a grammar is applicable to a non-terminal symbol $a \in N$, replacing it with B, whereas B can be any combination of non-terminals $\in N$ and terminals $\in T$.

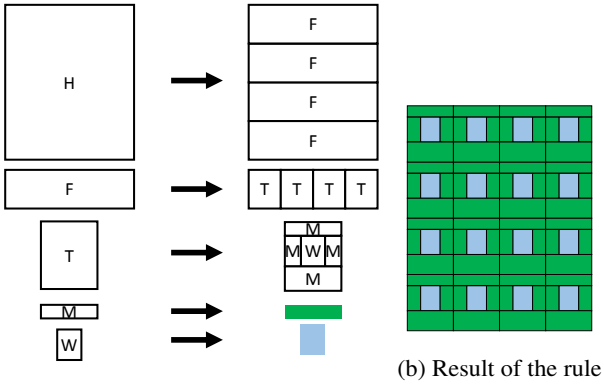
In a set grammar, the production process works on an active set of symbols. Initially, the active set consists of all axioms. During production, any non-terminal symbol from the active set of symbols is chosen and a fitting rule is executed on this symbol. The symbols generated by that rule are put back into the active set of symbols. This process continues, until there are only terminal symbols left in the active set.

In the case of split grammars, the production process works on shapes. Rules thus describe geometry operations on the input shape, generating any number of new shapes. For a grammar to be a split grammar, only two kinds of rules are allowed [19]:

- Split rules: A split rules splits a shape into multiple shapes, covering the exactly same volume as the input shape.
- Conversion rules: A conversion rules replaces a shape by zero to multiple shapes, where the generated shapes must be contained in the volume of the input shape.

These restrictions allow for a simple grammar derivation, as rules can only influence a constrained volume, as shown in Figure 1. Furthermore, every shape can be treated independently of the other shapes in the active set. This allows for a fully parallel production process. CGA, and consequently our grammar as well, do not have these restrictions and shapes can also increase in size, be moved or extruded. Furthermore, to simplify things, our implementation does not support control grammars lie CGA does.

To ease the process of writing rules, rules are usually composed of *operators*. Operators can be seen as basic geometric transformations executed in sequence to form a rule. Our grammar supports the transform-only operators



(a) A Simple Split Grammar

(b) Result of the rule set (a)

Figure 1: A split grammar works a set of shapes, each associated with a symbol. Rules (a) replace one shape by a group of other shapes. Using split grammars, more complex objects can be generated from very simple rules.

Translate, Rotate, and Scale and generative transformations that produce more shapes than existed before the transformation. Those operators are Repeat and Subdivide. Furthermore, we support two operations that change the dimension of a shape: the Extrude operator, applied to a quad, generates a box the ComponentSplit operator, applied to a box, generates quads, representing the six faces of the box. Finally, we support the GenerateTerminal and the DiscardTerminal operator. The former calculates the geometry or simply copies the scaled model matrix (details in the implementation section 4). For instance, the third rule in Figure 1(a) splitting shape T into five shapes can be modeled as a combination of two Subdivide operators.

We use C++ template code to write define operators in the rule sets. Three example operators are described below with listings 1, 2 and 3 to illustrate the syntax:

- **Repeat** takes two parameters, a successive symbol and a shape and produces as many new shapes with the width specified by the second parameter as fit into the original shape. The operator can work on either of the other two dimensions.

Listing 1: Repeat Operator

```
1 repeat<X, 2, CallRule<Successor>>
```

applied to a box with width 8 will output four new boxes with a width of two (and the remaining extents according to the input shape), which all have the symbol ”Successor” as its successive symbol.

- **Subdivide** takes a varying amount of parameters and successive symbols plus the input shape. The first parameter is again the axis, the operation is applied to. The remaining parameters specify the relative width/height/depth for the newly generated shapes and their successive symbol, which can be different for each shape. The symbol can also be the same for every output shape, but has to be specified as many times as there are output shapes.

Listing 2: Subdivide Operator

```
1 subdivide<Y,
2           SubdivParam<500, CallRule<Successor1>,
3           SubdivParam<500, CallRule<Successor2>>>
```

Applied to a box with the height of four, Subdivide will produce two boxes with the height of two (and the remaining extents according to the input shape). The successive symbols of the two resulting boxes will be ”Successor1” and ”Successor2”, respectively.

- **ComponentSplit** takes an input shape and generates as many new shapes of lower dimension as are needed to represent the faces of the original shapes. Our implementation supports only the splitting of a box into six quads. The operator needs to be provided only with the six successive symbols (which may be all the same symbol, but in this case have to be specified six times).

Listing 3: Component Split Operator

```
1 Compsplit<CSP<CallRule<Bottom>,
2           CSP<CallRule<Top>,
3           CSP<CallRule<Right>,
4           CSP<CallRule<Left>,
5           CSP<CallRule<Back>,
6           CSP<CallRule<Front>>>>>>>
```

Using CUDA, a single thread can be launched for every shape in the active set, applying the rule associated with the shape’s symbol. Despite the great potential for parallel execution in split grammars, traditional GPU stream processing approaches are not well suited to fulfill the derivation process efficiently because work loads are highly irregular in split grammars, leading to thread divergence. Since our grammar descends from split grammars, this problem needs to be considered.

To avoid thread divergence, a scheduling system based on rule queuing can be set up to keep up the occupancy of a GPU [15]. The results of this rule scheduling paradigm are promising, as this system allows to generate whole cities in real time. However, this work only focuses on a single strategy to schedule rules: The entire GPU is occupied with a persistent threads approach [1]. Symbols of equal type are collected in queues, while workers draw elements from these queues. All rules have to be available at compile time, requiring a full recompile when altering the rule set. In this work, we investigate the alternative methods to schedule shape grammars on the GPU. On the one hand, we investigate the benefits and downsides of scheduling whole rules versus scheduling work for each operator individually. On the other hand, we investigate the trade-off between compiling entire rule sets and interpreting the provided rule set during runtime.

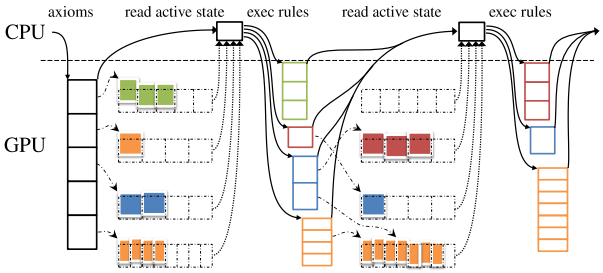


Figure 2: Using an individual queue for each rule, we provide an iterative shape rewriting algorithm, which does not suffer from divergence. At first all axioms are being placed in the queues. Then, we read the queue fill rate back to the CPU before launching just enough threads to process all queued shapes. We continue this process until there are only terminal shapes left.

3.1 Iterative Production

The most straight forward way to tackle shape grammar evaluation is starting a single thread for each symbol in the current active set. This is similar to the approach by Lipp et al. [8]. As mentioned before, this method has the drawback of extensive thread divergence. Inspired by the approach by Laine et al. [5], we avoid thread divergence, providing individual queues for each rule. Before running the rule evaluation, we allocate a queue for each rule on the GPU. We then insert the axioms into the per-rule queues. After querying the queue fill rates, we start an individual kernel for each queue, launching just as many threads as there are elements in each queue. Each thread then fetches one element from the queue and executes the rule associated with it. During rule evaluation, new shapes are generated, which are again inserted into the respective queues. Terminal shapes are placed into a separate set of arrays, for which no rule evaluation is taking place. These arrays are later used for rendering. After all kernel launches are completed, we read the queue fill rates from the GPU and again launch kernels to evaluate rules for all shapes currently being held by all queues. We continue this process until all non-terminal shapes have been processed, *i. e.*, all queues reach an empty state. Shapes currently being queued represent the current active set. This process is visualized in Figure 2.

Using this approach, all threads within one kernel evaluate the same rule, executing the same set of instructions. Thus, no thread divergence occurs and execution is efficient on the GPU hardware. While this approach is set up easily, deriving a whole rule set requires many kernel launches. Additionally, the queue fill rates need to be read back from the GPU before a new set of kernels can be launched. This step cannot be avoided, as the number of threads to be launched needs to be known.

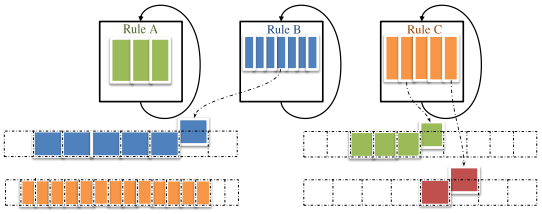


Figure 3: As alternative rule derivation algorithm, we use a persistent megakernel setup. Worker blocks are running in an endless loop. At the beginning of each loop iteration, they draw a new setup of shapes from one of the queues and evaluate the associated rules, before inserting the generated shapes back into the queues. The kernel is kept alive until all non-terminal shapes have been processed.

3.2 Persistent Megakernel Production

As alternative way to tackle shape grammar evaluation on the GPU, we use a persistent threads approach [1]. We again use a single queue per rule. But instead of launching a new kernel for every rule production, we run threads in an endless loop. In every loop iteration, each thread draws a shape from one of the queues and executes its associated rule. If new shapes are being generated, we add them back into the respective queues. As shapes are being drawn from the queues and inserted into the queues concurrently, we use a flag per queue element to avoid errors due to read-before-write dependencies [14]. All threads continue in their loop, until all queues are empty and no thread is still evaluating a rule. To avoid thread divergence, we force all threads within a block to draw shapes from the same queue in every iteration. This setup is outlined in Figure 3.

A persistent megakernel setup avoids synchronization with the CPU and does not have any kernel launch overhead. On the downside, all rules must be compiled into the same kernel. As kernels are optimized as a whole, the characteristics of the most resource-hungry rule determines the efficiency of all others. Furthermore, the persistent setup requires a more complex queuing strategy to avoid errors due to read-before-write dependencies. As our grammar does not introduce any priority among rules, shapes can be drawn from any queue at the beginning of each iteration. To avoid idle threads, we circle through all queues in a round robin fashion and only draw shapes from a queue if there are enough shapes in the queue to provide all threads in the block with work.

3.3 Precompiled Rules

The goal of the precompiled rule set approach is to leave as many decisions as possible to the compiler. For this approach, we require the complete rule set to be specified beforehand. This includes all rules, their parameters, and outputs. The only information not required in advance are the axioms. All computations and branch decisions that are not input dependent need only be done once, so we perform

them at compile time. Thus, during runtime, all operators can be executed without any additional information. No lookups to rule tables or symbol translations are needed.

The anticipated result is that this method achieves the best possible performance when compared to approaches that can adjust their behavior to different rule sets during runtime. Precompiled rules lose the flexibility of changing the rule set at run time and need significantly longer compile time. Usually, the performance gain from precompiling a rule set would be leveraged in production systems, such as games, once the design phase is finished and no interactivity is needed anymore.

Precompiled rule sets are evaluated in a ”one rule at a time” fashion by our software. This means that several operators can be chained together in a rule which forms the procedure to be called by the scheduler. While this approach has low scheduling overhead, it may not exploit all options for parallelism. The same operators are likely to be used in different rules and could be executed efficiently in parallel. However, the scheduler only knows about rules, thus it treats all rules as different. Moreover, the complexity of such a precompiled rule set can increase quickly. This circumstance not only imposes high requirements on the quality of the compiler, but also, if not implemented carefully, results in very high compile times, which may only be tolerable for production use.

3.4 Interpreted Rules

Interpreting rules at runtime gives flexibility when designing new objects at the cost of performance. With this approach, rule sets can be imported from file or created interactively, possibly with a rule editing tool—ideally with a graphical user interface.

In the interpreted mode, our solution evaluates rule sets in a ”one operator at a time” fashion. This means that every rule is broken apart into its operators and intermediate shapes are generated. These shapes are handed over to the scheduler. To determine how operators are strung together to rules for the currently used rule sets, we generate a dispatch table. This table holds for each (intermediate) symbol the operator to be executed, the parameters for each operator and all generated output symbols. During runtime, whenever a thread starts the evaluation for a certain shape, it fetches all parameters from the dispatch table and executes the requested operations. To avoid divergence, we keep one queue per operator. When a new shape is generated, we look up which operator should be called for it next and insert it into the respective queue.

The major advantage of interpreted rule sets is the ability to alter the rule set during run time, allowing for efficient prototyping and immediate feedback. Another advantage of interpreted rules is that the scheduler is now exposed to all available parallelism: shapes to be executed by the same operator can be grouped, even if they are used in completely unrelated rules.

4 Implementation

Our implementation is written in CUDA and C++ and makes heavy use of templates. Rendering is done in two different ways using OpenGL. The two variants are instanced and non-instanced rendering. In the non-instanced method, vertex, normal and instance data is generated by the terminal operator. The instanced method renders basic shapes (boxes, quads, etc) and only needs to apply the calculated model matrix to put the shape into its place in the final rendering.

Using instanced rendering has three advantages: First, during terminal evaluation, less data has to be written to slow global GPU memory, as only the matrices need to be copied. Second, less storage is required between generation and rendering. And third, during rendering, less data needs to be read, saving memory bandwidth. However, the number of vertices of the basic shapes is too low for an efficient usage of instanced rendering. Thus, rendering is actually slower using instancing.

To implement a shape, we store its type, size and the model matrix (and the symbol ID in the interpreted method). All operators, except the GenerateTerminal operator, only alter these attributes, which is all the information needed to produce the geometry data. Using the non-instanced rendering method, GenerateTerminal calculates, according to the type of the shape, the vertex attributes and stores them in an OpenGL buffer, which is mapped to CUDA before the generation process starts. If we use instanced rendering, all that is left to do for the GenerateTerminal operator, is to store the model matrix in the OpenGL buffer.

The precompiled method is implemented using the template meta-programming paradigm. All rule sequence decisions are made by the compiler according to the rule definitions, which creates instances of rules and operator chains at compile time. The rule definitions are written in C++ template code as shown in listing 4. We use the template code not only to generate the operator chains for the precompiled method, but also to fill the dispatch tables for the interpreted case. However, the compile process in the interpreted case does not involve the generation of GPU code, only the CPU code generating the dispatch table. Thus, a full runtime adjustment of the rule set could easily be achieved using a custom parser.

Listing 4: Sample Rule Set

```
struct RuleB : RuleT<Box, IfSizeLess<X, 200,
           DiscardTerminal, GenerateTerminal> > {};

struct RuleA : RuleT<Box,
        translate<0, 567, 0, GenerateTerminal > > {};

struct StartRule : RuleT<Box,
          rotate<45000, 45000, 0, subdivide<X,
          SubdivParam<270, CallRule<RuleA>,
          SubdivParam<160, CallRule<RuleB>,
          SubdivParam<300, CallRule<RuleA>,
          SubdivParam<270, CallRule<RuleB>
          > > > > > > > {};

typedef RuleSet<RS<StartRule, RS<RuleA,
         RS<RuleB> > > > TheRules;
```

	Houses	S. Sierpinski	M. Sierpinski
Rules	9	5	15
Operators	11	4	4
Terminals	332.8k	3.20M	1.35M
Vertices	7.99M	75.8M	32.5M
Indices	11.99M	115.2M	48.7M

Table 1: Test scene statistics

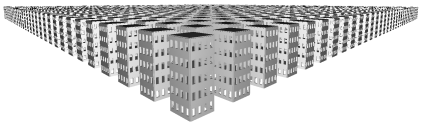


Figure 4: The *Houses* testcase shows a scene of 32×32 simple house models.

5 Results

As this paper focuses on the rule derivation process, only rudimentary shaders have been applied to visualize the produced geometry. The absence of visually appealing rendering, as well as other features not relevant for this paper, like optimizing the number of objects that need to be generated or ignoring geometry that need not be regenerated for every frame is the topic of future work. The results of the tests presented in this section were carried out on a system with an Intel Core i7-3770 CPU at 3.4 GHz, 16 GB of main memory and a NVIDIA Geforce GTX TITAN with 6 GB VRAM.

We compare eight different configurations, which are variations of interpretation and precompilation, instanced and non-instanced rendering, as well as iterative production and persistent megakernel production. We applied these variation to three different rule sets: *Houses*, *Single Sierpinski*, and *Multi Sierpinski*. The statistics for these three rule sets are summarized in Table 1. Example views for all scenes are shown in Figure 4-6.

The evaluation results are shown in Table 2-4. In all examples, the generation time in the instanced variant was between three to five times lower than the non-instanced variant. The highest difference was achieved in the Multi Sierpinski test case, which generates a vast amount of geometry with relatively few rule evaluations per terminal. This fact is clearly visible when looking at the number of

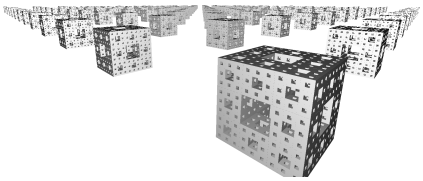


Figure 5: *Multi Sierpinski* consists of 13×13 Sierpinski Cubes at recursion depth 3.

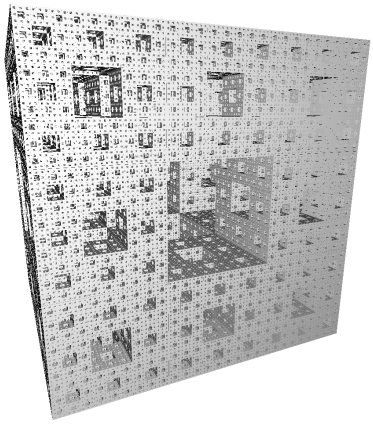


Figure 6: *Single Sierpinski* shows one deep Sierpinski Cube at recursion depth 5.

			t_g	t_r	load	store
int	n-inst	IP	16.6	2.4	1.55	28.51
		PMK	15.3	2.3	1.50	26.46
	inst	IP	4.5	5.7	1.33	7.72
		PMK	6.6	5.7	1.30	8.97
pre	n-inst	IP	21.4	2.4	1.11	32.93
		PMK	11.9	2.4	0.75	22.56
	inst	IP	2.8	5.7	0.75	4.72
		PMK	2.5	5.8	0.64	5.40

Table 2: Evaluation results for Houses, including interpreted (int) and precompiled (pre) rule sets; non-instanced (n-inst) and instanced (inst) rendering, as well as iterative production (IP) and persistent megakernel production (PMK). Generation time (t_g in ms) corresponds to the time needed for rule evaluation, t_r is the time spent in OpenGL rendering (ms), and load/store correspond to DRAM load and store requests to global GPU memory during grammar derivation.

			t_g	t_r	load	store
int	n-inst	IP	144.5	22.3	14.9	239.4
		PMK	103.3	22.1	11.7	188.1
	inst	IP	33.4	55.3	13.1	51.9
		PMK	32.3	56.0	11.6	52.0
pre	n-inst	IP	196.4	22.5	9.0	306.3
		PMK	105.7	22.0	5.8	190.4
	inst	IP	17.9	55.0	6.2	33.3
		PMK	21.8	55.3	5.8	34.7

Table 3: Evaluation results for Single Sierpinski

			t_g	t_r	load	store
int	n-inst	IP	57.6	9.2	4.6	98.2
		PMK	45.8	9.1	3.9	84.6
	inst	IP	10.7	23.2	3.8	18.6
		PMK	9.4	23.5	3.7	17.7
pre	n-inst	IP	84.5	9.3	2.3	131.1
		PMK	52.0	9.2	2.7	95.5
	inst	IP	5.5	23.3	1.1	11.3
		PMK	6.6	23.2	1.2	13.5

Table 4: Evaluation results for Multi Sierpinski

DRAM stores. The rendering time itself was hardly influenced by instancing, confirming that rendering really simple shapes using instancing is not more efficient than rendering the uncompressed geometry.

Surprisingly, in the non-instanced variant, interpreted rule evaluation was faster than precompiled in half of the cases, achieving an overall faster average rule evaluation by 12%. In the instanced variant the relationships are reversed, with precompiled outperforming interpreted by 84% on average. In all instanced tests, percompiled could generate the geometry faster. These are very interesting results, as precompiled is only significantly faster, when there is less memory traffic involved, due to the use of instancing. We can only assume that the interpreted evaluation can catch up in the non-instanced variant because all terminal operators are collected in the same queue. Thus, the terminal generation itself is highly efficient in comparison to the precompiled rule derivation, where the terminal generation is mixed with other operations. Thus, the interpreted version generates more homogeneous memory access patterns and overall runs faster. This is also reflected by the lower number of DRAM stores in the interpreted non-instanced versions when comparing interpreted to percompiled. In the instances variants, these numbers are reversed. Most of the memory access of the interpreted evaluation is due to dispatch table lookups and intermediate symbol generation, thus slowing down the generation process.

When comparing our iterative production implementation against the persistent megakernel approach, one can observe that the persistent megakernel implementation is on average 20% faster than the iterative production. In nine of the twelve cases, persistent megakernels were faster. Interestingly, there is no generalizable pattern visible, as to when iterative production works better. In the Houses test case, iterative production gives the best results for interpreted+instanced, for Single and Multi Sierpinski it achieves the best performance for precompiled+instanced.

Overall, we can observe that persistent megakernel production seems to work faster on average than iterative production. Instancing always increases performance. If instancing is used, precompiled rule sets are better than interpreted rule sets. If instancing is not used, terminal generation dominates performance, for which the interpreted rule sets are slightly faster, as they are able to merge the terminal operators.

When looking at the raw generation times, we can observe that the fastest method can generate 135 million terminals per second (MTPS) in the Houses test case, 178 MTPS in the Single Sierpinski test case and 247 MTPS for the multi Sierpinski rule set.

6 Future Work

Since the focus of this paper is the evaluation of different rule scheduling strategies, we omitted the implementation of features which only affect appearance and not performance. These features include textured rendering, auxiliary scenery like roads, water, vegetation and varying elevation of the ground. Also the support of imported offline generated models would make the scene more lively. Furthermore, a randomization of input parameters, so the generated shapes do not all look alike, would be essential for producing realistic scenes. For the testing setup of our implementation, the use of boxes and quads was sufficient. To build more realistic housing procedurally, many more shapes could be implemented, like cylinders, cones and wedges. We plan to add these features in the future.

A rule editor to specify interpreted rules at run time would be beneficial to the usability of our solution, as writing rules off-line is not very intuitive, especially for generating complex models. Such an editor would ideally support writing rules in an already established shape grammar and could even support using a rule database, so users can import and export model descriptions like it is already done for conventional 3D models.

An interesting feature to implement is proper use of instanced rendering. While in our case it was enough to render instances of basic shapes to prove that instanced rendering is desirable when constantly generating geometry every frame, to save bandwidth, the vertex data for basic shapes is far to low to justify the instancing overhead for the rendering alone. Rendering instances of fully generated objects with a reasonable amount of vertex data would use the full potential of instanced rendering.

Last but not least, an important aspect of CGA is context sensitivity. In our implementation this was deliberately left for future investigation, since the complex matter of rule interdependency is out of scope of this work.

7 Conclusion

We have shown in this work that scheduling of rule derivation work load on a GPU in the context of grammar based procedural modeling has several aspects influencing performance that have to be considered carefully. First, decision making can be offloaded to the compilation stage in order to avoid expensive branching at run time.

Second, the proper utilization of GPU programming paradigms, while being partly platform dependent, as we focus primarily on NVIDIA CUDA technology, is essential in order to avoid wasting precious resources, slow memory accesses and thread divergence.

Third, the amount of data being moved when generating geometry on the fly ought not to be underestimated, which is why the use of instanced rendering is the preferred method to render massive amounts of procedurally generated models.

Furthermore, when applying excessive template programming, the quality of a decent compiler is not to be underestimated, as is the consideration how code generation (especially its memory consumption) will respond to chaining templates together recursively.

References

[1] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High Performance Graphics*, pages 145–149. ACM, 2009.

[2] Sven Havemann. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2005.

[3] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum*, 29:2291–2303, 2011.

[4] Patrick Lacz and John C. Hart. Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors*, pages 23–23, 2004.

[5] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 137–143, New York, NY, USA, 2013. ACM.

[6] Sylvain Lefebvre, Samuel Hornus, and Anass Lasram. By-example synthesis of architectural textures. *ACM Trans. Graph.*, 29:A84, 2010.

[7] Jinjie Lin, Daniel Cohen-Or, Hao Zhang, Cheng Liang, Andrei Sharf, Oliver Deussen, and Baoquan Chen. Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph.*, 30(6):A183, December 2011.

[8] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel Generation of Multiple L-systems. *Computer and Graphics*, 34(5), 2010.

[9] Milán Magdics. Real-time generation of l-system scene models for rendering and interaction. In *Proceedings of the 25th Spring Conference on Computer Graphics*, SCCG '09, pages 67–74, New York, NY, USA, 2009. ACM.

[10] Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Gaël Sourimant. GPU Shape Grammars. *Comp. Graph. Forum*, 31(7-1):2087–2095, 2012.

[11] Paul Merrell and Dinesh Manocha. Model Synthesis: A General Procedural Modeling Algorithm. *Visualization and Computer Graphics, IEEE Trans.*, 17(6):715–728, 2011.

[12] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[13] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990.

[14] Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph.*, 31(6):A161, 2012.

[15] Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, and Dieter Schmalstieg. Parallel generation of architecture on the GPU. *Comp. Graph. Forum*, 33, 2014.

[16] George Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel, 1975.

[17] George Stiny. Introduction to shape and shape grammars. *Environment and planning B*, 7(3):343–351, 1980.

[18] George Stiny. Spatial Relations and Grammars. *Environment and Planning B*, 9:313–314, 1982.

[19] Peter Wonka, Michael Wimmer, François X. Sillion, and William Ribarsky. Instant Architecture. *ACM Trans. Graph.*, 22(3):669–677, 2003.

[20] Tingjun Yang, Zhengge Huang, Xingsheng Lin, Jianjun Chen, and andJun Ni. A parallel algorithm for binary-tree-based string re-writing in the l-system. In *Proceedings of the Second International Multi-Symposiums on Computer and Computational Sciences*, IMSCCS '07, pages 245–252, Washington, DC, USA, 2007. IEEE Computer Society.