Representing and Scheduling Procedural Generation using Operator Graphs

Pedro Boechat*, Mark Dokter†, Michael Kenzel*, Hans-Peter Seidel†, Dieter Schmalstieg*, Markus Steinberger†
*Graz University of Technology, Austria
†Max Planck Institute for Informatics, Germany

Figure 1: *Using the operator graph, we optimize the procedural generation of an entire city containing* 50 000 *buildings and* 10 *million triangles. The original shape grammar takes more than five minutes to generate on the CPU. Previous work on the GPU requires* 630 ms. *Using our optimization framework, the generation time reduces to* 109 ms *by only changing the scheduling of operations on the GPU.*

Abstract

In this paper, we present the concept of operator graph scheduling for high performance procedural generation on the graphics processing unit (GPU). The operator graph forms an intermediate representation that describes all possible operations and objects that can arise during a specific procedural generation. While previous methods have focused on parallelizing a specific procedural approach, the operator graph is applicable to all procedural generation methods that can be described by a graph, such as L-systems, shape grammars, or stack based generation methods. Using the operator graph, we show that all partitions of the graph correspond to possible ways of scheduling a procedural generation on the GPU, including the scheduling strategies of previous work. As the space of possible partitions is very large, we describe three search heuristics, aiding an optimizer in finding the fastest valid schedule for any given operator graph. The best partitions found by our optimizer increase performance of 8 to 30x over the previous state of the art in GPU shape grammar and L-system generation.

Keywords: Procedural Generation, Operator Graph, GPU, Autotuner, Dynamic Scheduling

Concepts: **•Computing methodologies → Graphics processors;** *Massively parallel algorithms;*

1 Introduction

In recent years, content creation for virtual worlds has become increasingly limited by human effort, rather than technology. Manually crafting every detail of vast virtual worlds for games and feature films is tedious and time-consuming. Thus, it is not surprising that procedural generation is becoming widely adopted in the digital content creation industry, shifting part of the labor from designers to an automated system. Using a procedural approach, complex models are created from small procedural programs or rule sets. A simple program written by an expert can generate a large number of plausible variants of a model type, *e.g.*, buildings for an entire city.

Procedural generation methods are present in all phases of a content authoring pipeline. In the design phase, tools like automatic object placement and style transfer [Guerrero et al. 2015] may evaluate procedural programs hundreds of times to match high level modeling goals. Similarly, inverse procedural modeling techniques [Talton et al. 2011] may execute thousands of parameter sets to steer a model towards a target function. In the deployment phase of an open world video game, a program might be evaluated millions of times, as the user moves through a procedural world. Analogously, during the rendering of a movie, procedural models might be evaluated on-the-fly for every frame, because reevaluation is more cost-efficient than keeping models around. While the aforementioned applications use different procedural generation methods, they all re-evaluate a specific program or grammar thousands to millions of times, with speed being key for usability, frame rates or production time.

In this paper, we consider the challenge of mapping a procedural generation to massively parallel architectures like the *graphics processing unit* (GPU), with the goal of achieving the lowest possible generation time. While prior work has proposed ad hoc strategies specific to particular generation methods [Lipp et al. 2010; Marvie et al. 2012; Steinberger et al. 2014b], we introduce a generalized, systematic approach to scheduling a class of procedural generation methods. We introduce the *operator graph*, an intermediate representation that serves as a unified formal model for a variety of procedural systems and encodes a space of ways to map the generation onto the GPU. Specifically, we make the following contributions:

*{boechat,kenzel,schmalstieg}@icg.tugraz.at
†{mdokter,hpseidel,msteinbe}@mpi-inf.mpg.de

SA '16 Technical Papers, December 05 - 08, 2016, Macao
ISBN: 978-1-4503-4514-9/16/12
DOI: http://dx.doi.org/10.1145/2980179.2980227

- We introduce the *operator graph* as a representation for a program or grammar written in a variety of procedural languages. It is independent of a particular way a designer might write the program or describe the generation process.
- We establish the operator graph as an *intermediate representation* for a domain-specific programming system for procedural generation, in which the operator graph not only describes the generation, but its partitions also encode different ways of scheduling the execution of the program on the GPU. Existing GPU approaches can be reduced to a specific partitioning scheme.
- We show that the scheduling for a given program or grammar can be optimized using an *auto-tuner* based on heuristic-supported search. With an upfront time investment for optimizing the schedule, the auto-tuned generation outperforms the previous state-of-the-art and hand-tuned schedules.

To validate our results and show that the operator graph can be used to parallelize different types of procedural generation systems, we analyze graphs of varying sizes and different characteristics. We fully analyze the space of possible partitions for a number of small graphs verifying the proposed heuristics. For large graphs, we show that the scheduling has considerable influence on the execution time, leading to speed ups of up to 30× over previous GPU approaches. Discussing how the operator graph partitioning covers previous GPU evaluation strategies we compare different scheduling strategies within the same framework, showing that an optimized schedule is up to 14× faster than hand crafted solution. We conclude that optimizing the scheduling based on the operator graph leads to the currently fastest procedural generation evaluation on the GPU.

2 Related work

Computer aided procedural modeling has a long history. Stiny's original *shape grammars* [1975] were one of the first approaches to apply a sequence of procedural operations on a given set of shapes. Later on, Stiny refined this work to define *set grammars* [Stiny 1982]. While shape grammars consider operations on shapes, similar procedural generation processes can be defined on character strings. These approaches, called *L-systems*, are inspired by plant growth patterns [Prusinkiewicz and Lindenmayer 1990]. Similarly, Wonka et al. [2003] found that facades can effectively be described by *split grammars*, a special case of shape grammars, which restrict the available operations to space subdivisions. Combining concepts from the original shape grammar work, L-systems, and split grammars, *CGA shape* [Müller et al. 2006] is able to describe complex buildings and cities. While the work on shape grammars and L-systems spans three decades, the idea behind productions generation remains unchanged. Given a shape or symbol, an operation is applied to that entity to yield new shapes/symbols.

One of the most important extensions to simple procedural generation is considering external influences, such as the environment [Prusinkiewicz et al. 1994], external guidance [Beneš et al. 2011], or vector fields [Li et al. 2011]. In contrast to external influences, interactions between generated objects can be modeled, such as interconnecting different structures [Krecklau and Kobbelt 2011], resolving intersections of generated geometry [Parish and Müller 2001], or querying neighboring shapes [Müller et al. 2006]. Treating shapes as first class citizens allows a variety of queries between generated shapes and even temporary objects [Schwarz and Müller 2015]. Procedural modeling can also be extended to support more general terminal symbols [Krecklau et al. 2011] or more basic mesh editing functions [Havemann 2005]. While these extensions introduce additional dependencies into the generation, the very basic principle of procedural generation remains unchanged.

Parallel Evaluation of Grammars is the basis for computation on the GPU. L-systems and shape grammars, which form the foundation of many procedural systems, provide a high degree of parallelism. In L-systems, every symbol can be worked on independently. Shape grammar evaluation follows a tree-like derivation, where different nodes in the tree can be worked on in parallel. Mapping this parallel workload to the GPU, one can distinguish two classes of approaches. The first class encodes the procedural evaluation within the GPU graphics pipeline; the second uses the GPU compute mode to execute procedural programs written in CUDA or OpenCL. For the first class, a large number of approaches tailored to specific generation systems exist. Split grammars have been evaluated in vertex and pixel shaders [Lacz and Hart 2004], L-system have been derived using multi-pass rendering [Magdics 2009], and full shape grammar evaluation has been spread across geometry and pixel shaders [Marvie et al. 2012]. Instead of generating output geometry, facade grammars can also be executed for every screen pixel [Haegler et al. 2010; Marvie et al. 2011]. To achieve good performance, the aforementioned approaches need to handle irregular parallelism introduced by the nature of procedural generation. This is mostly done with sorting and reshuffling data between launches and, as found by most authors leads, to a significant overhead.

The second class of approaches also include specific solutions for L-systems and shape grammars. Lipp et al. [2010] proposed an approach similar to the traditional multi-pass rendering, launching successive kernels. Between launches, prefix sums and sorting are required for context sensitivity and to reduce execution divergence. Using a work queue, Steinberger et al. [2014b; 2014c] perform an entire shape grammar derivation within a single kernel launch, dynamically handling the challenges of irregular parallelism. Considering both classes of algorithms, there are simply a lot of options for mapping procedural systems to the GPU. While each of the aforementioned approaches provides an efficient solution for one specific class of procedural systems, there is no approach that can be applied to different domains or considers different options for scheduling. Striving for the most efficient generation, considering automation for making scheduling decisions seems to be the logical next step. To enable automation for different classes of procedural systems, one requires a common representation for all problems. With the operator graph, we provide this representation and show how it can be used to automate scheduling decisions for efficient procedural generation on the GPU.

Scheduling of domain specific workloads has received increased research interest, as mapping the execution of complex problems to parallel hardware can be a tedious task. While specialized frameworks have been proposed that assist application programmers in different domains, no such tool exists for procedural generation. For example, OptiX [Parker et al. 2010] dynamically schedules raytracing application on the GPU, the GRAMPS scheduler [Sugerman et al. 2009] can schedule graphics pipelines on CPU architectures, Halide [Ragan-Kelley et al. 2012; Mullapudi et al. 2016] assists and automates the scheduling of image processing algorithms for the CPU and GPU, and Piko [Patney et al. 2015] schedules graphics pipelines on the GPU. Most approaches use a domain-specific intermediate representation in the process. He et al. [2015] even encode possible schedules in their representation. Similarly, our operator graph can be seen as an intermediate representation for procedural systems that also encodes schedules. However, none of the aforementioned approaches is suitable for procedural generation with its typically large number of relatively efficient operations that often only vary in terms of parameters. With our approach, we fill this gap, providing a common representation for a variety of procedural generation approaches, a domain-specific programming design for procedural systems, and an auto-tuning scheduler for the GPU.

3 Operator graph representation

While a wide variety of procedural generation methods for different application domains has been proposed, their execution model can most often be represented with a rather simple graph. Consider the following methods: L-systems [Prusinkiewicz and Lindenmayer 1990], which were developed to model plant growth, define expansions on symbol strings. Shape grammars, like CGA shape [Müller et al. 2006], which are well suited to model buildings, define spatial relationships between shapes. Stack-based generation languages, such as GML [Havemann 2005], which are well suited to model detailed man-made objects, define a list of fine-grained modeling operations on polyhedra.

The common factor among them is that they all describe sequences of operations applied to objects. This observation creates a link to dataflow programming [Wadge and Ashcroft 1985] or the stream processing abstraction [Sussman et al. 1983], in which programs are defined as directed graphs. Graph-based abstractions have been used for procedural generation before. For example, commercial products like Houdini by Side Effects Software or Autodesk Maya use data-flow networks. Directed acyclic graphs have been used for simple shape grammars [Patow 2012]. However, these graphs do not support more complex generation methods or offer the information needed for efficient scheduling. To this aim, we introduce the *operator graph* as an intermediate representation that captures the requirements a procedural approach imposes on a scheduler. In this section, we describe the basics of the operator graph. A detailed account how it can be used to represent the aforementioned procedural modeling methods is given in Appendix A.

Graph definition We model the procedural generation as a directed multigraph $G = (V, E, D)$, whereas $V = \{v_k\}$ describes the vertices of the graph, $E = \{e_j\}$ corresponds to directed edges modeling the flow of objects in the graph, and $D = \{d_j\}$ is an additional set of directed edges introducing dependencies in the graph. Similar to dataflow programming and stream processing, the nodes in the graph describe operations that are applied to objects that travel along the edges E of the graph. Dependencies D introduce restrictions on the order of the operations. Alongside the multigraph, sets of supported operations R and object types O are needed to describe the procedural generation. Both R and O may differ strongly between generation methods and implementations of those methods.

An **edge** $e = (v_s, v_d, O_e, m_e)$ is an ordered quadruple, connecting a source vertex $v_s \in V$ to a destination vertex $v_d \in V$, and defines a path an object can take through the graph, *i.e.*, there is a possibility for v_s to output an object that will be input to v_d. $O_e \in O$ defines the set of objects that can travel along the edge, *i.e.*, objects that may be input to v_d. m_e is called the multiplicity of the edge, describing the number of objects that will move over an edge concurrently after a single invocation of the operation associated with node v_s. A multiplicity set to a static number indicates that a specific number of objects will always be generated by the node; / indicates that either 0 or 1 object will be generated, and * indicates that an arbitrary number of objects can be generated (including 0). The source vertex for an edge shall be given by $s(e) = v_s$, the destination vertex, by $d(e) = v_d$, and its multiplicity, by $m(e) = m_e$.

A **vertex** $v = (r, (p_1, \cdots, p_n))$ is described by an operation $r \in R$ and a list of parameters $(p_1, \cdots, p_n)$, which influence the operation. Whenever an object moves over an incoming edge to v, the operation r is invoked on that object with the node's parameters; we say the object has been consumed by the node. Every invocation might output any number of objects; we say the objects are produced by the node. The types of objects consumable by an operation $r \in R$ and the types of producible objects can be defined by a left-total, binary relation $IO_r \subseteq O \times O$. The operation defines the number of outgoing edges of the node; for each possible output, an edge must exist in the graph.

Examples In a shape grammar, a *translation* operation in $\mathbb{R}^3$ moves an object along a direction given by three parameters. It consumes and produces a single object in $\mathbb{R}^3$. A translation node has one incoming edge and one outgoing edge, each being limited to the same set of objects. In a *string rewriting* system, an operation rewrites characters defined in the alphabet Σ according to matching rules R. It consumes a character and outputs as many characters as there are parameters. Both incoming and outgoing edges are limited to single characters. The outgoing edge multiplicity is equal to the number of parameters. A *triangulation* operation consumes any flat polygon, takes no parameters and triangulates the interior of the polygon. The associated node has a single input edge, which is limited to polygons, and a single outgoing edge with multiplicity *, which allows for triangles only. A *random path* operator choses randomly between different outgoing edges with the parameters describing the likelihood for each option. A random path node consumes any object type and produces objects of the same type. For every likelihood given as parameter, an outgoing edge with $m = /$ exists.

A **dependence edge** $d = (v_s, v_d)$ is an ordered pair, connecting a source vertex $v_s \in V$ to a destination vertex $v_d \in V$, and defines a secondary graph on top of V. Dependencies model side effects in the operator graph that cannot be captured by E. Such side effects describe influences the operation from the source vertex v_s can have on the operation of v_d. For example, the generation of a wall (v_s) may limit an operation v_d the controls the growth of a tree. Dependency edges can be seen as additional parameters to the operation of v_d, which are set up dynamically, as other objects move through the graph. If a dependency edge exists between nodes, the system executing the operator graph has to make sure that there is no possibility that any objects being present somewhere in the graph might still go through v_s, before executing objects waiting at v_d. In the example above, this means that the generation of trees has to wait, until all walls have been generated.

For a vertex v, the incoming edges shall be given by $in(v) = \{e \in E | d(e) = v\}$ and the outgoing edges by $out(v) = \{e \in E | s(e) = v\}$. A vertex with $in(v) = \varnothing$ is called source node, and a vertex with $out(v) = \varnothing$ is called terminal node. Source nodes correspond to operations that start the procedural generation by introducing initial objects into the graph. Terminal nodes correspond to operations that end the generation process, by discarding objects or outputting objects as part of the generated model.

Fractal Example Consider the recursive generation of the Menger sponge, shown in Figure 2, with the operator graph in Figure 3. The generation can be completed by using axis-aligned boxes with a recursion counter: The required objects can be given by the tuple $o = (\mathbf{s}, \mathbf{t}, c)$, where $\mathbf{s} \in \mathbb{R}^3$ is the dimension of the box, $\mathbf{t} \in \mathbb{R}^3$, its position, and $c \in \mathbb{N}$, the recursion counter. The generation uses six operations: one split operation for each dimension, splitting a box into three boxes whose relative sizes are given as parameters, an operation increasing the recursion counter, a conditional operation comparing the recursion counter to the parameter and deciding between two different generation paths, an operation that discards an object, and an operation that adds the object to the generated model. The generation starts at the blue source node with a single outgoing edge with a multiplicity $*$, indicating that any number of initial boxes might come from the source node. Each box is split along the X, Y, and Z-axis, creating three sub-boxes each, leading to a grid of 3×3 equally sized boxes. Seven of these boxes are

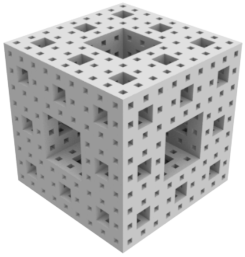


Figure 2: *The Menger Sponge generated by a recursive generation program stopped at level 3.*

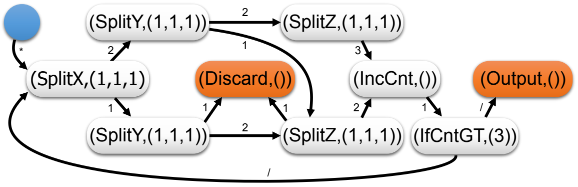


Figure 3: *The operator graph for the Menger Sponge. The blue node is the source; orange nodes are terminals. Each node is annotated with its operation and parameters. For each edge, its multiplicity is given:* * *indicates a variable number of objects can be produced,* / *stands for* 0 *or* 1 *produced object. No dependencies are present.*

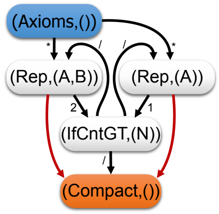
discarded, and the remaining twenty boxes are split anew, unless a fixed number of recursions have been carried out. The recursive nature of the generation is captured by a cycle in the graph. The graph does not contain additional dependencies.

L-System Example An L-system is defined over an alphabet S, axioms $a \in S$ and production rules P, which take a character $s \in S$ and map it to a string $(s_1, \ldots, s_n)$ of characters $s_i \in S$, see Lindenmayers's original algae example in Figure 4(a). The result of an L-system evaluation is a string of characters after a certain number of iterations. A system for L-system evaluation can be described as follows. An object is modeled by a character, its position on the string and a rewrite counter: $O = S \times \mathbb{N}^2$. Only a single parameterized rewrite operation is needed, which produces one object (character) for each parameter. It also increases the recursion count and updates the string position. As the string position depends on the operations that are carried out for all characters to the left, we compute a conservative string positions and leave empty spaces in the string, *i.e.*, the operator multiplies the position of the input with the maximum number of characters that are produced by any operator. After the desired number of rewrites, a compaction operator removes the empty spaces and forms the output string. To stop the generation, a conditional operation is required, see Figure 4(b).

Every production rule, the recursion check, and the compaction step translate into a node in the operator graph. As the compaction requires all other objects to exist first, a dependency edge is introduced in the graph. This dependency is a typical example of a side effect. The operator graph does not only contain the original L-system, but describes a full system that enables the interpretation of the L-system. For example, if one would want to describe the parallel L-system derivation by Lipp et al. [2010], one would introduce a prefix sum node (replacing our compaction step) within the recursive cycle. Note that the axioms themselves are not part of the operator graph and thus it describes all possible algae generations from any combinations of starting symbols and not only a static scene. Further examples can be found in Appendix A.

```
Alphabet: A, B
Axiom: A
Production rules:
1: (A -> AB)
2: (B -> A)
```

(a) *L-system*



(b) *Operator Graph*

Figure 4: *Simple algae L-system and its translation to an operator graph. Red edges indicate dependency edges.*

4 Operator graph scheduling

Previous approaches proposed very specific ways of running procedural generation on the GPU, trying to include different aspects of best practices in GPU programming. Foremost, this includes (1) supplying enough parallelism to fully utilize the GPU by load balancing between the execution cores, (2) avoiding thread divergence, and (3) avoiding costly memory transactions. While previous approaches try to achieve these goals in very different ways, we generalize these techniques using the concept of the operator graph.

From an operator graph view, a system for procedural generation on the GPU provides a user with the definitions of supported operations R and objects O. The user specifies the generation process in a textual or graphical way. This step corresponds to setting up the operator graph and handing it to the generation system. The system must be able to manage objects that are produced throughout the generation process, storing them in one of the memory spaces available on the GPU. For every operation supported by the system, a GPU implementation must exist. Additionally, the runtime system must provide the parameters to the operations and consider the dependencies between graph nodes. The decisions about which objects should end up in which memory space and which operations should be executed when and on which processing cores can have a significant influence on the performance. Inspired by the naming in Halide [Ragan-Kelley et al. 2012], we call the sum of these decisions the *schedule* of the generation.

4.1 Scheduling strategies

In terms of the operator graph, the procedural generation is completed, when all objects have moved through the entire graph. The time spent on the generation can be divided into time spent on individual nodes and edges of the graph. The cost of a node corresponds to the time spent on the execution of operations. Operations are executed quickly, if (1) a sufficient number of them can be executed in parallel, (2) no divergence occurs, *i.e.*, all GPU cores are active, and (3) operations executing on the same SIMD cores require the same low-level instructions. The cost of an edge can be seen as time needed for scheduling, including assigning cores to operations as well as loading and storing of objects. Ideally, scheduling decisions take as little time as possible and lead to divergence-free parallel execution.

Dynamic scheduling However, in real systems, these are opposing goals. To create divergence-free parallel execution, a scheduler needs to collect objects that are to be executed by the same operation. As new objects can be generated during any other operation, on any multiprocessor, there is no way around global, device-wide communication. Such communication is only possible via slow global GPU memory. A global sorting or grouping mechanism for objects that are to be processed by the same operation is ultimately necessary. Thus, objects must be transferred to and from global memory between the execution of operations. In the worst case, the sorting or grouping mechanisms involve device-wide synchronization. The combination of these steps can be very costly in comparison to the execution of a single operation, possibly increasing the cost of edges way beyond the cost of nodes. However, the cost of nodes will be low, as operations are executed divergence-free, and load balancing occurs over the entire device. As these scheduling decisions determine when and where operations are executed during runtime, we call them *dynamic*.

Static scheduling The most time-efficient way of scheduling is to avoid any decision making during runtime. The only possibility to achieve such a *static* scheduling decision on current hardware is to continue using the resources already allocated for a previous operation, fusing an operation with its successor, *i.e.* an object produced by a thread is stored within the registers allocated to this thread and the subsequent operation is executed by the same thread. If a node with multiple outgoing edges should be processed by a single thread and scheduling is completely static, the execution of all child nodes can only be serialized. This does not allow for load balancing and reduces the amount of available parallelism. At the same time, the chances of divergence rise, if the number of produced objects depends on the input object. However, there is essentially no overhead associated with static scheduling. Note that other scheduling systems, especially, if they work on entire kernels, call this kind of static scheduling *kernel fusion*.

Static and dynamic scheduling decisions represent extremes along a continuum of schedules. For example, instead of going to global memory for a dynamic scheduling decision, one can also involve local shared memory, combining information from all threads running on the same multiprocessor. This will generate a faster dynamic scheduling decision with less overhead and fewer options in terms of load balancing and divergence avoidance. However, there is usually a pivotal point on this continuum that separates dynamic scheduling decisions, involving decisions made during runtime, from static scheduling decisions, involving only resource reuse and a predefined execution order. In the interest of brevity, we distinguish only dynamic and static scheduling decisions in the following discussions.

4.2 Graph partitioning

While scheduling decisions can either be dynamic or static, there is no need to make scheduling decisions uniformly throughout the generation process. It is rather possible to decide for each operator graph edge if it should involve a dynamic or static scheduling decision, adjusting the schedule according to a desired execution pattern. Given that the commands executed for individual operations do not change, the schedule is actually (for a given GPU) the only factor influencing the execution time of a production. Distinguishing between dynamic and static scheduling decisions, it is possible to describe a schedule as a partitioning of the operator graph. Edges within a single component of the partition involve static scheduling decisions only. Edges between different components denote dynamic scheduling decisions. We define a *schedule* S as a partitioning of V into non-empty subsets. We call each component $S_i \in S$ an *execution group*. Note that, by definition, S itself is also an execution group. Due to the way scheduling decision can be made, certain conditions must hold for each execution group for the schedule to be valid.

Condition 1 There can only be a single node $v_r \in S_i$, for which incoming edges have source nodes outside S_i. This condition makes sure that each component has a single node v_r from which the execution of the component starts. All other nodes in the component can be executed in a statically defined order.

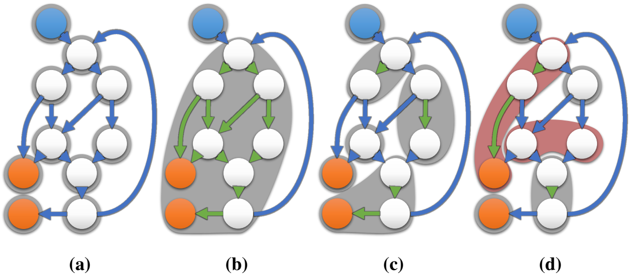


Figure 5: *Simplified Menger Sponge operator graph partitions. Blue edges are dynamic, green edges static, execution groups are outlined in grey. (a) Turning every node into an execution group, yields completely dynamic scheduling. (b) A large execution group with mostly static edges. (c) Rules turned into execution groups. (d) Invalid partitioning (red) having multiple nodes with dynamic incoming edges or disconnected nodes.*

Condition 2 There must be a path from the starting node v_r to each other node $v_n \in S_i$ of the execution group that only contains nodes of S_i. This condition guarantees that a static schedule involving all nodes in S_i can be defined. All edges that are used in the above definition shall involve static scheduling decision and shall be called *static edges*.

Condition 3 An execution group is not allowed to contain cycles, besides cycles that include v_r. This conditions prohibits recursions within a component, which might lead to unpredictable time and resource requirements within statically scheduled nodes. If a cycle is formed with v_r, the incoming edge at v_r is made dynamic, and the cycle can be supported via dynamic scheduling.

Condition 4 Dependency edges are only allowed between different execution groups and not within an execution group. This condition is necessary, as context sensitivity is usually only supported between dynamic scheduling decision, *i.e.*, the scheduler must be able to check if all sources of the dependency have been executed, before the dependent node can be executed.

Examples We revisit the Menger sponge, as shown in Figure 5. If all edges are made dynamic (a), a large number of small execution groups is generated, possibly leading to large scheduling overhead. As the graph contains a single cycle, nearly the entire graph can be turned into a single execution group (b). The edge creating the cycle becomes a dynamic edge. The first node in a large execution group is the only one with incoming edges from outside of the execution group. A schedule like this only draws parallelism from the recursion and the source node. A mixture of dynamic and static edges creates execution groups of different sizes (c). A setup like this might be a good compromise between dynamic and static scheduling. Not all possible partitions are valid; unconnected nodes or multiple nodes with incoming edges outside of the execution group yield an invalid schedule (d).

Operator graph scheduling allows modeling previous work as different partitioning schemes, independent of the low level details of the implementation: Sequential rewriting and shape grammar evaluation algorithms, such as the approaches by Lacz et al. [2004] and Lipp et al. [2010], yield execution groups $S_i \in S$ of single nodes, $|S_i| = 1$. Thus, they make scheduling decisions after every single operation. While they get good load balancing and a high degree of available parallelism, their scheduling overhead is large. GPU shape grammars [Marvie et al. 2012] put the entire graph into a single execution group $S = V$. Thus, they can only draw parallelism from the axioms and face problems with divergence, when generating

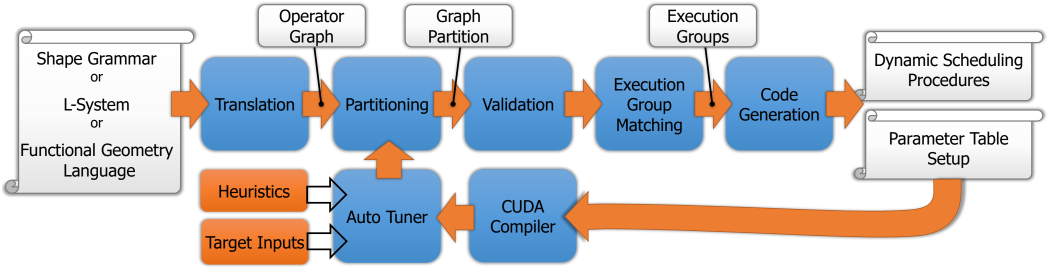


Figure 6: *Using the operator graph as central concept, our compile pipeline consists of seven major steps: A procedural generation written in one of the supported languages is translated into an operator graph; after selecting one partition, it is validated, and execution groups are formed, while taking operator homomorphism into account. Finally, code for a dynamic scheduler is generated alongside the parameter table. After compilation to GPU code, an auto-tuner records statistics for possible input parameters and steers the selection of different partitions.*

different buildings. PGA [Steinberger et al. 2014b] partitions the graph according to the rules written by the designers. Thus, their performance heavily depends on the way the rule sets have been written and can suffer the same problems as the other approaches.

4.3 Execution group matching

There is another factor influencing performance that can be described using the operator graph. Dynamic scheduling combines objects that are to be executed by the same execution group to generate divergence-free parallel execution. If the number of objects for one execution group is too low, the execution will suffer underutilization or divergence. To mitigate this issue, we propose execution group matching in combination with dynamically loading parameters.

To describe execution group matching, we define *operator homomorphism*, which applies to operator graphs that are compatible in terms of structure and operations and, thus, can be described by the same parameterized execution group: Let $G_1 = (V_1, E_1, D_1)$ and $G_2 = (V_2, E_2, D_2)$ be two different operator graphs with $e_1 \in E_1$, $e_2 \in E_2$ and $d_1 \in D_1$, $d_2 \in D_2$. These operator graphs are *operator homomorphic*, iff there is a bijective homomorphism $f_h : V_1 \rightarrow V_2$, which fulfills the following conditions:

$$\forall e_1 \, \exists e_2 : f_h(s(e_1)) = s(e_2) \wedge f_h(d(e_1)) = d(e_2) \qquad (1)$$

$$\exists e_1 \, \forall e_2 : f_h(s(e_1)) = s(e_2) \wedge f_h(d(e_1)) = d(e_2) \qquad (2)$$

$$\forall d_1 \, \exists d_2 : f_h(s(d_1)) = s(d_2) \wedge f_h(d(d_1)) = d(d_2) \qquad (3)$$

$$\exists d_1 \, \forall d_2 : f_h(s(d_1)) = s(d_2) \wedge f_h(d(d_1)) = d(d_2) \qquad (4)$$

$$f_h\left((r_1, p_1)\right) = (r_2, p_2) \longrightarrow r_1 = r_2, \qquad (5)$$

i.e., if there are matching operations in all nodes in both graphs and edges that connect those nodes in the same way.

If the graphs of two execution groups are operator homomorphic, they can be described by a single parameterized piece of code, constructed from the operations used in either graph. This means that all possible ways through two operator homomorphic graphs can be described by the same code. This concept is not only applicable to two execution groups, but can be extended to multiple execution groups, combining all of them. The implication is that more objects can be combined for this parameterized execution group and thus executed more efficiently. However, as the parameters of the individual operations might differ, the scheduling system must provide the parameters dynamically to the executing objects.

In addition to easing the process of finding a sufficient number of objects for efficient execution during dynamic scheduling, a homomorphic execution group is represented using a single GPU function instead of multiple. This might have an additional positive effect on the cost of dynamic scheduling. Depending on the implementation of dynamic scheduling mechanism, fewer execution groups might reduce the number of grouping structures or queues and thus reduce the time spend on searching through this structure. Therefore, execution group matching can potentially reduce the overhead of dynamic scheduling.

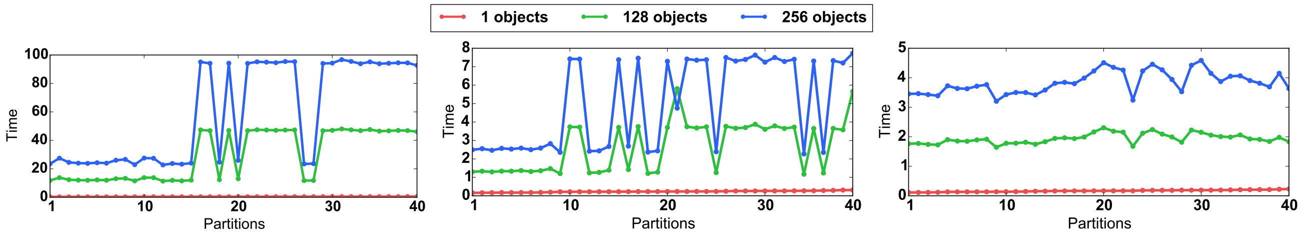
5 Compiler Pipeline

To complete our approach, we describe a state-of-the-art procedural generation runtime system and an auto-tuner for the selection of schedules. These components close the loop of our compile pipeline, see Figure 6. As dynamic scheduler, we use the task scheduling framework *Whippletree* [Steinberger et al. 2014a]. Whippletree works around the definition of procedures and tasks. Procedures are function-like entities that take tasks as input. Tasks are collected in queues in global GPU memory. When there are enough tasks available for parallel execution on a multiprocessor, Whippletree executes them together, increasing the chances for a divergence-free execution. In our terminology, tasks correspond to objects moving through the operator graph and procedures are operations or execution groups. Whippletree makes dynamic scheduling decisions as to when and on which multiprocessor these execution groups should be executed. Note that Whippletree has been used for shape grammar scheduling before. However, we do not use Whippletree's shape grammar implementation, but rather generate a schedule directly from an operator graph and our own operation definitions.

5.1 Procedural generation system

We implemented a limited number of operations, which are similar to the shape grammar operations available in CGA shape, as well as operations for L-systems and functional languages for procedural generation. To use Whippletree for dynamic scheduling, we define a Whippletree procedure for every execution group in the operator graph. The procedure is built from operations used in the execution group, together with the parameters specified. These parameters can either be predefined values or drawn from random distributions. Within an execution group, we serialize the execution of all contributing operations, implementing static scheduling decision. Serializing the graph in a depth first manner, we make sure that the number of temporary objects that need to be kept in registers is low. Whenever a dynamic edge is reached, we hand the object over to the Whippletree scheduler, which transfers it into its own grouping mechanism and load balances its execution among the entire GPU.

We require a way to supply dynamic parameters to operator homomorphic execution groups. For this purpose, we use a parameter table stored in GPU memory. This table is used to look up the parameters of the involved operations based on a unique identifier that we



(a) *GTX 630* **(b)** *GTX 780 Ti* **(c)** *GTX 980*

Figure 7: *Performance results in ms for all 40 possible schedules for the Menger Sponge test case with different object counts (sorted by performance on the GTX 630). Even for such a small rule set, the performance difference between schedules can vary from a factor of two to five. Note how the relative performance of the schedules changes between object counts and GPU.*

store alongside each object. During execution, each thread looks up the parameters associated with its object. Using the texture cache for parameter loads reduces the overhead of these additional memory transactions. As a subset of the parameters might be identical for all operator homomorphic execution groups, we use the look-up table only for those parameters which actually differ. The remaining parameters are statically included during compilation.

Our implementation also supports context sensitivity, *i.e.*, dependency edges. Dependencies always involve two operations. The first operation (source) adds an object into a spatial data structure and adds a customizable id to that object. The second operation (destination) allows to query an object against the objects stored in the spatial data structure. If there is an overlap between the query and the objects of a given id in the data structure, a different outgoing edge is chosen than if there is no overlap. Scheduling needs to make sure that all source operations are completed, before the destination nodes are executed. Whippletree does not allow to set up these dependencies directly. However, we can separate the execution in multiple phases with global synchronization barriers in between by executing a sequence of Whippletree programs, one for each phase. Whenever there are dependencies in an operator graph, we make sure that source and destination nodes end up in different phases, while keeping the number of phases minimal.

5.2 Scheduling optimizer

The fastest schedule for a given operator graph might depend on the GPU architecture and the number of objects generated by the source node. Depending on the number of cores, a GPU requires more or less parallel workload to work efficiently. A larger number of initial objects provides more parallelism, and, thus, less dynamic scheduling is necessary to create enough parallel workload. When trying to find the best schedule, these factors must be considered.

The number of different graph partitions (and different schedules) for an operator graph with $|E|$ edges is $2^{|E|}$ (every edge can either be dynamic or static). While a large number of these schedules might achieve similar performance, the difference between a good schedule and a bad schedule can—according to our experiments—be up to two orders of magnitude for large operator graphs. Even for small graphs, the best performing schedule varies between object counts and GPU type, as shown in the example in Figure 7. This large variance shows that for modern GPUs, it is difficult to predict what makes a schedule good or bad. Thus, it is essential to provide means to find the best schedule for any given operator graph, object count and GPU.

With the goal of finding the most efficient procedural generation system, we search for the schedule that minimizes execution time. To this aim, we implemented an auto-tuner that searches for the best schedule for a given operator graph. It is intended to compute good schedules for use cases such as video games, movie production or inverse procedural modeling. As a baseline, it uses an exhaustive search algorithm, generating and evaluating all valid schedules for a given set of source objects.

Our compiler pipeline starts with a rule set written in a syntax similar to CGA shape or an L-system and internally translates it into an operator graph. To perform this translation, the compiler requires a definition of supported object types, operations, and input-output relations (cf. O, R, and IO in section 3). We use C++ class specifications for object types and C++ source code for the operations and input-output relations. Internally, our compiler represents all possible partitions as a bit sequence (one bit per edge). After a partition is selected, it is checked for validity according to the conditions given in section 4.2. If it is valid, we search for operator homomorphic execution groups and identify common parameters. Finally, the compiler generates Whippletree procedure code for all execution groups, including the CUDA/C++ code for the involved operations. It also generates the parameter table for the procedures and inserts the corresponding load instructions into the CUDA/C++ code. The generated code is compiled, and the auto-tuner evaluates its performance for a given set of target input objects.

Even after restricting the search space to valid partitions, it is still not practical to perform an exhaustive search through the remaining number of possible schedules for larger graphs. Thus, we introduce a set of heuristics based on parallel programming principles translated to choices in the operator graph. These heuristics work by setting edges as static or dynamic, according to certain local characteristics of the operator graph. Naturally, fixing an edge (to either static or dynamic) reduces the search space in half. Although heuristics significantly reduce the search space, there is no guarantee that the best schedule is found. However, we have collected strong evidence that the presented heuristics work well for a wide variety of cases.

5.3 Sequence fusion heuristic

Nodes with a single outgoing edge of multiplicity $m = 1$ do not introduce any parallelism. If no parallelism is introduced, there is no gain for load balancing, and a dynamic edge would only increase scheduling overhead. Thus, the sequence fusion heuristic H1 sets all edges in $E_{h,1}$ to static:

$$E_{h,1} = \{e|m(e) = 1 \wedge |out(src(e))| = 1\}.$$

For example, consider the sequence of a *Translate*, *Rotate* and *Scale* operation, as shown in Figure 8. For simplicity, assume that all operations as well as a dynamic scheduling decisions take equally long. The operations carried out on different SIMD units and cores are represented as cells of the table. Light cells indicate available cores, and dark cells indicate that the resource is used. Using static edges

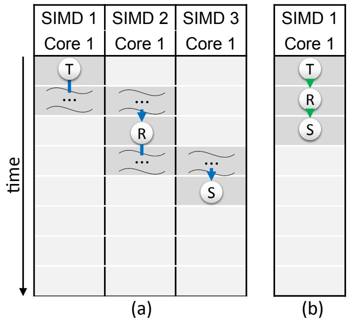


Figure 8: *H1 (a) → (b) removes dynamic scheduling if there is only a single object generated.*

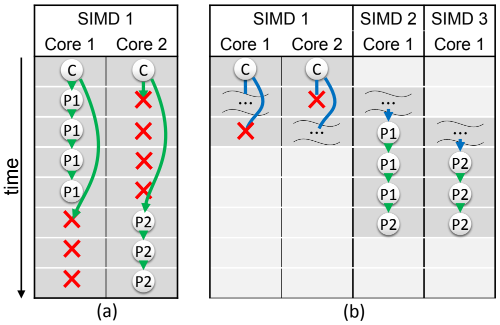


Figure 9: *H2 (a) → (b) removes divergence (red crosses) by adding dynamic scheduling under conditional nodes.*

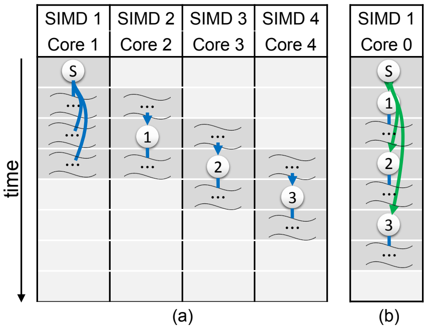


Figure 10: *H3 (a) → (b) makes sure small execution groups are not split apart and thus reduces dynamic scheduling overhead.*

between those operations will result in a single thread executing all three operations, with virtually no cost for scheduling. Dynamic edges, in this case, might result in the operations to be executed with different threads. However, no more than a single thread will be active with those operations at any point in time. Additionally, the overhead for two dynamic scheduling decisions increases the overall execution time. Resource usage (darker cells) is reduced from seven to three, and the execution time is reduced from five to three by applying the heuristic. Intuitively, static edges should lead to a better performance in this case.

Note that applying this heuristic might link two dependent operations to each other with a sequence of static edges. For example, imagine a dependency edge between the *Translate* and *Scale* operation in the previous example. One of the static edges needs to be turned into a dynamic edge to yield a valid schedule. In our implementation, this is taken care of by the optimizer, which validates each application of heuristics. It would reject the static edge, when the heuristic is applied to the last edge in the sequence, as it would connect the dependencies to each other.

5.4 Divergence avoidance heuristic

Edges with multiplicity $m = /$ have a high change of introducing divergence. Consider an *If* operation or an operation that chooses one of the outgoing edges at random. For those operations, different objects have a high chance of choosing different edges for execution. If these edges are made static, thread divergence will occur. Thus, the divergence avoidance heuristic H2 sets all edges in $E_{h,2}$ to dynamic:

$$E_{h,2} = \{e | m(e) = /\}.$$

For example, consider a conditional operation executed on two cores of the same SIMD unit, as shown in Figure 9. Each conditional branch is followed by a sequence of operations. If static scheduling is used, and objects choose different edges, half of the cores will not run code for the entire execution. Using dynamic edges, however, will only introduce a small amount of divergence, when starting the dynamic scheduling. The execution of the following operations can then be carried out in parallel on different SIMD units, avoiding divergence and speeding up the execution (eight time units vs six time units). Also, the overall resource usage (darker cells) is reduced from 16 to 15. Intuitively, dynamic edges will allow to reduce divergence, introduce parallelism and speed up the execution in this case.

5.5 Execution group size heuristic

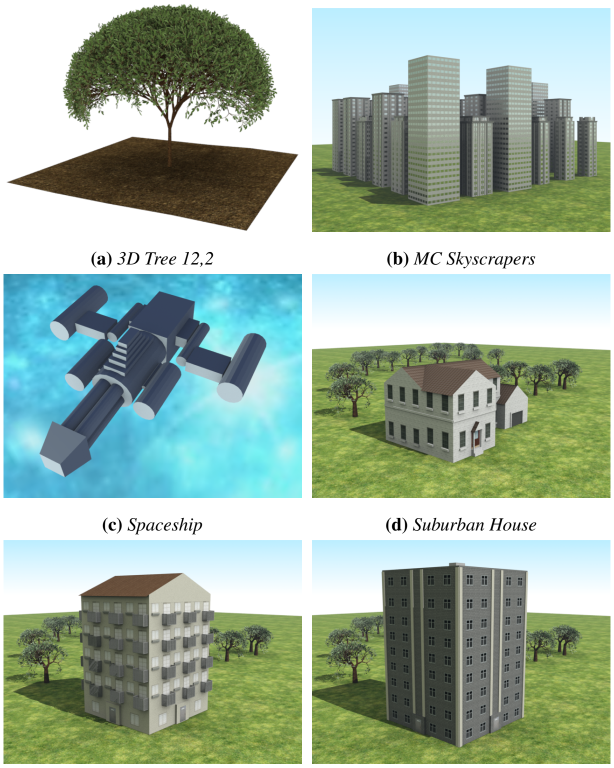
In contrast to edges that might or might not be taken, there are those nodes that have edges with a fixed multiplicity, *i.e.*, there will always be a fixed number of objects created by an operation. Examples include *Subdivide* or *ComponentSplit*. If the outgoing edges of such a node are set to static, combining a node with its children into the same execution group will prevent divergence, but potential parallelism will be lost. At first sight, it may appear that these outgoing edges are good candidates for dynamic edges to increase parallelism. However, one has to consider the overhead of dynamic scheduling. This is particularly true, when the edges following the successors are dynamic, too, creating sequences of dynamic scheduling decisions. In this case, the scheduling overhead will outweigh the gains of parallel execution. Thus, the execution group size heuristic H3 enforces a minimum execution group size by greedily setting edges with a constant multiplicity to static until a certain execution group size t is reached.

$$\forall S_i \subseteq S : |S_i| \geq t \qquad (6)$$

For example, consider a *Subdivide* operation creating three objects, each followed by a single node and a dynamic scheduling operation, as shown in Figure 10. If the edges are set to dynamic, three dynamic scheduling operations are carried out, and the following nodes are distributed among different cores. In this way, parallelism is generated, but with a high amount of scheduling overhead. However, if the edges are set to static, all nodes are executed on the same core, and the scheduling overhead is greatly reduced. While the overall execution is taking slightly longer (one time unit), the consumed resources (darker cells) are nearly halved from thirteen to seven. As long as there is a sufficient amount of parallelism available, the reduction in resource usage will lead to a faster generation process. In our experiments, we used a minimum execution group size t of 5.

6 Results

To evaluate our approach, we used test cases from shape grammars, L-systems, and Monte Carlo procedural modeling approaches as shown in Figure 11 and Figure 1. The test cases include a variety of sizes and include results for different numbers of initial object counts. As test system, we used an Intel i7 4820K with 16GB RAM and an NVIDIA GTX 780Ti.



(a) *3D Tree 12,2* **(b)** *MC Skyscrapers* **(c)** *Spaceship* **(d)** *Suburban House* **(e)** *Balcony* **(f)** *Commercial*

Figure 11: *The evaluation test cases include L-systems (a), Monte Carlo procedural modeling (b and c), and shape grammars (d-f).*

6.1 Evaluation of the heuristics

To evaluate the heuristics on test data sets of a non-trivial size, we use the following approach: First, we identify all edges in a given operator graph for which a heuristic does not apply and randomly set each of those edges to be either dynamic or static. Second, we apply the heuristic to the other edges and run the schedule for a given set of input objects, recording their performance. Third, we invert those edges to the opposite of the heuristic and run the schedule again on the same input set. We repeat the entire process, until we arrive at a predefined number of samples. For the pairs of samples, we run a paired Student's t-test to check if there is a difference between the performance of the schedules with activated heuristic and inverted heuristic. To determine the number of samples for achieving a 5% error margin and a confidence interval of 95%, we created 384 pairs, following the sample size guidelines [Krejcie and Morgan 1970].

In each of the test cases, we make sure that the pattern occurs on which the heuristic is based. We used *Suburban House* for H1, as it is rather small and offers multiple edges that match the heuristic, *Commercial* for H2, as it chooses randomly from different window styles, and *Balcony* for H3, as it has many nodes with only few outgoing edges. The t-test results for all three evaluations are shown in Table 1. As can be seen, all heuristics had a statistically significant influence on performance for all axiom counts. Altering 19.7%, 4%, and 41% of edges boosted performance up to 30%, 12%, 100%, on average, for H1, H2, and H3, respectively, indicating that the chosen edges actually have a large impact on performance. The heuristics seem to work especially well for larger object counts. Overall, we argue that all three heuristics seem to be a good starting point for optimization, reducing the search space significantly.

Table 1: *The heuristics affected 15, 6 and 43 edges out of the 76, 150 and 104 edges in the respective operator graphs. All three heuristics had statistically significant influences on the performance (p value and t for the paired Student's t-test). H is the mean generation time (with standard deviation) in ms with heuristic on, while $\neg H$ shows the same results with heuristics off. For small object counts (Obj), up to 89% (perc) of pairs benefited from the heuristic. For larger object counts, the heuristics work even better.*

Test	Obj	t	p	H	$\neg H$	perc
	1	-8.36	<.001	0.58 (0.65)	0.61 (0.68)	.81
H1	64	-51.38	<.001	1.13 (0.72)	1.36 (0.72)	.99
15/76 edges	128	-74.36	<.001	1.73 (0.70)	2.16 (0.69)	.99
	256	-94.54	<.001	2.95 (0.73)	3.81 (0.72)	1.00
	1	-18.20	<.001	0.53 (0.09)	0.58 (0.08)	.87
H2	64	-30.76	<.001	0.84 (0.16)	0.94 (0.15)	.94
6/150 edges	128	-34.85	<.001	1.16 (0.29)	1.29 (0.28)	.95
	256	-35.86	<.001	1.79 (0.56)	2.01 (0.53)	.96
	1	-2.87	.004	0.51 (0.16)	0.54 (0.04)	.89
H3	64	-27.87	<.001	0.65 (0.17)	0.90 (0.07)	.98
43/104 edges	128	-55.96	<.001	0.78 (0.16)	1.33 (0.13)	1.00
	256	-94.96	<.001	1.09 (0.16)	2.19 (0.25)	1.00

6.2 Runtime performance

To relate our approach to previous work, we ran the *Tree 4,3* test case from GPU Shape Grammars [Marvie et al. 2012], the *Tree 8,3*, *Overview* and *Skyscrapers* test cases from PGA [Steinberger et al. 2014b] as well as the *MC Spaceship* and *MC Skyscrapers* from Stochastically-Ordered Sequential Monte Carlo [Ritchie et al. 2015]. We include their performance numbers for CPU and GPU approaches (adjusted for hardware differences). Additionally, we compare how the scheduling strategies followed by previous work introduced in our scheduler perform compared to the best schedule found by our auto-tuner. Rewriting approaches (*RW*), like the ones by Lacz et al. [2004] and Lipp et al. [2010], correspond to execution groups of size 1 and all parameters being stored in the parameter table. GPU Shape Grammars [Marvie et al. 2012] are represented by a schedule of a single execution group with all parameters being static (*SGL*). To represent PGA [Steinberger et al. 2014b], we used execution groups according to the rules specified in the shape grammar rule set and again provide all parameters as statics. For unbiased evaluation, we let an external expert on shape grammars write those rule sets for us (*DGN*). We ran our optimizer in four different setups: random search (OPT_S), search with heuristics (OPT_H), and both variants with execution group matching ($OPT_{\cdot,M}$).

The performance results are shown in Table 2. SGL performs well for small operator graphs and large object counts, as there is a sufficient parallelism available, and scheduling overhead is reduced to a minimum. Additionally, there is a low chance of divergence for such small operator graphs. RW introduces the most parallelism, but also has the highest scheduling overhead. Thus, it performs better, when there are few initial objects for which parallelism must be generated quickly. DGN leads to more balance between scheduling overhead and the ability to generate parallelism. It works well for test cases which are above a certain size, outperforming both other approaches. In case of smaller test cases, DGN is, however, outperformed by SGL. Looking at all three approaches, we can see that there is no single strategy that always performs best. For the smaller test cases, the auto-tuner can search the entire space. Thus, OPT_S slightly outperforms OPT_H as it tests every single schedule. For the larger test cases, a fully random search is less likely to pick good schedules, thus the heuristics guided search most often

Table 2: *Evaluation results in ms for test cases with different number of edges in the operator graph (Edges), objects counts (Obj), and number of terminals (Term) for various partitioning schemes. Previous implementations are included in terms of CPU and GPU generation time (performance extrapolated from original work using GPU FLOPS ratio). While a single execution group (SGL) performs well for small graphs, it is not well suited for large graphs. The rewriting approach (RW) behaves in the opposite way. The approach using the rules as execution groups (DGN) forms a trade-off between the two. Our optimizer using a random search (OPT$_S$), heuristic search (OPT$_H$), and with execution group matching (OPT$_{S,M}$, OPT$_{H,M}$) always find a better or equally good schedule. Time used for auto-tuning is additionally given; for the first three test cases all possible schedules can be tested. For the following test cases we sampled a maximum of 1000 schedules.*

Scene	Edges	Obj	Term	CPU	GPU	SGL	RW	DGN	OPT$_S$		OPT$_{S,M}$		OPT$_H$		OPT$_{H,M}$	
Overview[1]	9	38000	0.91M	996	48.24	**1.56**	5.24	1.75	1.62	1m	**1.56**	1m	1.76	1m	1.74	1m
Simple House	11	4096	1.48M			3.23	6.36	3.59	2.86	3m	**2.81**	3m	2.93	1m	2.91	1m
Menger Sponge 3	17	256	2.05M			7.28	2.87	2.87	2.28	10m	**2.27**	10m	2.30	2m	2.31	2m
3D Tree 12,2	35	64	0.65M			1.29	3.89	1.41	3.27	5h	2.99	5h	**1.26**	2h	1.28	2h
3D Tree 4,3[2]	35	1	283		10.06	0.10	0.50	0.14	0.21	4h	0.21	4h	0.09	57m	**0.09**	50m
3D Tree 8,3[1]	35	1	23K		3.71	0.20	1.34	0.36	0.32	4h	0.35	4h	0.19	1h	**0.19**	1h
MC Spaceship[3]	51	16384	0.40M	4996		**1.48**	5.47	2.77	1.96	6h	1.91	5h	1.55	4h	**1.48**	4h
Skyscrapers[1]	61	529	1.84M	516	22.51	12.10	13.99	6.81	**4.15**	7h	4.23	7h	4.64	7h	4.52	7h
MC Skyscrapers[3]	63	512	1.02M	818		65.6	6.11	5.61	**4.00**	7h	4.11	7h	4.80	7h	4.57	7h
Suburban House	76	256	0.60M			14.54	5.75	4.08	2.74	11h	2.45	10h	1.95	8h	**1.87**	8h
Balcony	104	256	0.18M			4.99	3.57	2.16	1.18	7h	1.16	7h	**0.75**	9h	0.76	8h
Commercial	150	256	0.14M			7.58	3.20	2.68	1.06	7h	1.06	7h	1.04	6h	**0.96**	6h

[1] from [Steinberger et al. 2014b], [2] from [Marvie et al. 2012], [3] from [Ritchie et al. 2015]

achieves better results in less time. Also, execution group matching usually achieves slightly better results while at the same time reduces compile time. Overall, OPT always picks the best option, boosting performance by up to 14.4× compared to SGL, 7.1× compared to RW, and 2.8× compared to DGN. On average, OPT is better by 3.9×, 3.6×, and 1.8×, respectively. The larger the operator graph gets, the larger the difference becomes. This points towards the fact that efficient scheduling for complex problems cannot easily be done by hand. Also, as procedural worlds are growing in size, the gains of an auto scheduler grows.

As the operator graph can be applied to different procedural approaches, we used it for L-system generation, shape grammars and Monte Carlo procedural modelling. GPU Shape Grammar's 3D Tree 4,3 is constructed by their approach in 40 ms. Adjusting for GPU differences (time multiplied by peak FLOPS ratio) results in about 10 ms on our GPU. The same test case is completed by OPT$_H, M$ in 0.09 ms. The comparisons to PGA (Overview, 3D Tree 8,3 and Skyscrapers) show that OPT finds schedules that are 5 to 30 times faster than their GPU approach (adjusted for GPU differences) and up to 100 times faster than their CPU approach. Bringing Monte Carlo procedural modeling to the GPU, an optimized schedule can speed up the generation process multiple hundred times compared to the CPU implementation by Ritchie et al. Considering all results, it is safe to assume that our compiler finds efficient schedules. The comparison to previous work suggests that the combination of our compiler with an efficient dynamic scheduler yields the currently fastest procedural generation system for the GPU.

However, finding the best schedule for a given operator graph comes with a cost. For every single schedule tested, we have to generate the partition from the operator graph, generate source code, compile it, and run a representative set of procedural generations. The partition and source code generation takes less than a second, the compilation takes between 20 seconds and 2 minutes, depending on the operator graph size. Thus, the optimization for large test cases can take multiple hours. As the same schedule can be used for different initial objects, the target scenario for our approach is optimizing a generation that is used heavily in a production system for different inputs. For example, we used our approach to optimize the generation for an entire randomly generated city as seen in Figure 1.

7 Conclusion

While previous approaches to procedural generation on the GPU employed specialized data structures and techniques, we looked at the problem from a high-level perspective and introduced the concept of the operator graph to handle procedural generation methods that can be described by directed graphs where each operation works on a single input object. This restriction is most often not an issue in practice, as language features that require multiple input objects can often be transformed into sequential subgraphs, as shown in the Appendix. Using the operator graph as an intermediate representation, we concisely describe the generation process for all possible input objects and we draw a connection between a partition of this graph and how a system can schedule the generation on the GPU.

Additionally, we identify similar structures within the graph and reduce the complexity of the generated code. Using our compiler and auto-tuner pipeline, we search the space of all partitions to find the schedule that performs the generation within the shortest time. While an exhaustive search of this exponential space is only possible for small graphs, we proposed three heuristics, which all increase search performance. Finally, we showed that our approach can significantly speed up the generation of a variety of common procedural generation systems. Our auto-tuner is able to increase performance over hand crafted solutions by up to fourteen times only by changing the schedule.

Although our heuristics help reduce the search space for finding the optimal schedule, in practice, the pruned search space of large operator graphs is still too large to be tested exhaustively. Viewing each choice for an edge as a binary value, the search for the best schedule can also be seen as a high dimensional binary optimization problem. Thus, we will investigate the use of probabilistic search algorithms like Simulated Annealing and Genetic Algorithms for the specific problem of finding a good schedule for procedural generation. Also, we want to extend our concept of operator graph scheduling to other graph-based traversal problems that can be described by objects moving through a graph. Our implementation is open source and can be downloaded from https://github.com/pboechat/OperatorGraph.

Acknowledgements

This research was supported by the Max Planck Center for Visual Computing and Communication.

References

Beneš, B., Štava, O., Měch, R., and Miller, G. 2011. Guided Procedural Modeling. *Comp. Graph. Forum 30*, 2, 325–334.

Guerrero, P., Jeschke, S., Wimmer, M., and Wonka, P. 2015. Learning shape placements by example. *ACM Trans. Graph. 34*, 4 (July), 108:1–108:13.

Haegler, S., Wonka, P., Arisona, S. M., Gool, L. J. V., and Müller, P. 2010. Grammar-based Encoding of Facades. *Comp. Graph. Forum 29*, 4, 1479–1487.

Havemann, S. 2005. *Generative Mesh Modeling*. PhD thesis, TU Braunschweig.

He, Y., Foley, T., Tatarchuk, N., and Fatahalian, K. 2015. A system for rapid, automatic shader level-of-detail. *ACM Trans. Graph. 34*, 6 (Oct.), 187:1–187:12.

Krecklau, L., and Kobbelt, L. 2011. Procedural Modeling of Interconnected Structures. *Comp. Graph. Forum 30*.

Krecklau, L., Pavic, D., and Kobbelt, L. 2011. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum 29*, 2291–2303.

Krejcie, R. V., and Morgan, D. W. 1970. Determining sample size for research activities. *Educ Psychol Meas*.

Lacz, P., and Hart, J. 2004. Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors*, 23–23.

Li, Y., Bao, F., Zhang, E., Kobayashi, Y., and Wonka, P. 2011. Geometry Synthesis on Surfaces Using Field-Guided Shape Grammars. *IEEE Trans. Visualization and Computer Graphics 17*, 2, 231–243.

Lipp, M., Wonka, P., and Wimmer, M. 2010. Parallel Generation of Multiple L-systems. *Computers & Graphics 34*, 5, 585–593.

Magdics, M. 2009. Real-time Generation of L-system Scene Models for Rendering and Interaction. In *Spring Conf. on Computer Graphics*, Comenius Univ., 77–84.

Marvie, J.-E., Pascal, G., Hirtzlin, P., and Gael, S. 2011. Render-Time Procedural Per-Pixel Geometry Generation. In *Graphics Interface*, 167–174.

Marvie, J.-E., Buron, C., Gautron, P., Hirtzlin, P., and Sourimant, G. 2012. GPU Shape Grammars. *Comp. Graph. Forum 31*, 7-1, 2087–2095.

Mullapudi, R. T., Adams, A., Sharlet, D., Ragan-Kelley, J., and Fatahalian, K. 2016. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph. 35*, 4 (July), 83:1–83:11.

Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. 2006. Procedural Modeling of Buildings. *ACM Trans. Graph. 25*, 3, 614–623.

Parish, Y. I. H., and Müller, P. 2001. Procedural modeling of cities. In *Proc. SIGGRAPH 2001*, 301–308.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. 2010. Optix: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July), 66:1–66:13.

Patney, A., Tzeng, S., Seitz, Jr., K. A., and Owens, J. D. 2015. Piko: A framework for authoring programmable graphics pipelines. *ACM Trans. Graph. 34*, 4 (July), 147:1–147:13.

Patow, G. 2012. User-friendly graph editing for procedural modeling of buildings. *Computer Graphics and Applications, IEEE 32*, 2 (March), 66–75.

Prusinkiewicz, P., and Lindenmayer, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

Prusinkiewicz, P., James, M., and Měch, R. 1994. Synthetic Topiary. In *Proc. SIGGRAPH 94*, 351–358.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*, 4 (July), 32:1–32:12.

Ritchie, D., Mildenhall, B., Goodman, N. D., and Hanrahan, P. 2015. Controlling procedural modeling programs with stochastically-ordered sequential monte carlo. *ACM Transactions on Graphics (TOG) 34*, 4, 105.

Schwarz, M., and Müller, P. 2015. Advanced procedural modeling of architecture. *ACM Trans. Graph. 34*, 4 (July), 107:1–107:12.

Steinberger, M., Kenzel, M., Boechat, P., Kerbl, B., Dokter, M., and Schmalstieg, D. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph. 33*, 6, 228:1–228:11.

Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. 2014. Parallel generation of architecture on the GPU. *Computer Graphics Forum 33*, 2, 73–82.

Steinberger, M., Kenzel, M., Kainz, B., Wonka, P., and Schmalstieg, D. 2014. On-the-fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum 33*, 2, 105–114.

Stiny, G. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag.

Stiny, G. 1982. Spatial Relations and Grammars. *Environment and Planning B 9*, 313–314.

Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., and Hanrahan, P. 2009. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph. 28*, 1 (Feb.), 4:1–4:11.

Sussman, G., Abelson, H., and Sussman, J., 1983. Structure and interpretation of computer programs.

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., and Koltun, V. 2011. Metropolis Procedural Modeling. *ACM Trans. Graph. 30*, 11:1–11:14.

Wadge, W. W., and Ashcroft, E. A. 1985. *Lucid, the dataflow programming langugage1*. Academic Press.

Wonka, P., Wimmer, M., Sillion, F. X., and Ribarsky, W. 2003. Instant Architecture. *ACM Trans. Graph. 22*, 669–677.

A Operator graph equivalence to conventional representations

Recall that a variety of methods and systems for procedural generation can be described using an operator graph. In this section, we provide additional detail on how the most prominent methods for procedural generation can be cast into operator graph form.

A.1 L-systems extensions

The operator graph can also be used to describe extensions for L-systems. Stochastic L-systems allow to specify multiple production rules for a single symbol, each being chosen with a certain probability. This behavior can be integrated into the previously described system by adding an operation that chooses one of its outgoing edges at random. Context-sensitive L-systems adjust the production to the characters before and after the input. This behavior can be modeled with dependency edges, making sure that all symbols for a certain iteration are processed, before executing the rules for the next iteration. Another extension to L-systems adds parameters to characters. Using our definitions, this can be achieved by extending the object type to $O = S \times \mathbb{N}^2 \times \Re^n$, with n being the number of parameters that should be associated with each character.

A.2 Shape Grammars

From a procedural generation point of view, shape grammars, like CGA shape [Müller et al. 2006], are similar to L-systems. A designer sets up production rules, which are associated with symbols to define the sequence of operations to be applied. However, the underlying objects are shapes (O), and the production rules themselves can be sequences (or trees) of operations. These operations can usually be chosen from a predefined set of parameterized operations (R). As it has been shown before that rule sets of simple, context-free shape grammars can be described by a direct acyclic graphs [Patow 2012], it is not surprising that our operator graph can be used for shape grammars. However, our operator graph can model more complex shape grammar operations, like context sensitivity or recursive productions.

For example, a rule set written in CGA shape to create the Menger Sponge (see Figure 2 and 3) could look as follows:

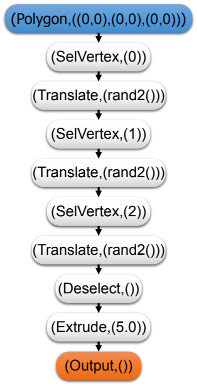
```
1: A -> Split(X){1r: B,
2:     1r: Split(Y){1r: C, 1r: Discard, 1r: C},
3:     1r: B }
4: B -> Split(Y){1r: D, 1r: C, 1r: D}
5: C -> Split(Z){1r: E, 1r: Discard, 1r: E}
6: D -> Split(Z){1r: E, 1r: E, 1r : E}
7: E -> IncRec(){IfRecGreater(4){Term, A}}
```

This rule set defines five rules, using the *Split*, *Discard*, *IncRec*, and *IfRecGreater* operations. $1r$ represents a relative size parameter. Translating a rule set into an operator graph is straight forward. Linking rules with symbols corresponds to setting up edges between nodes. Also, the nesting of operators within a rule (line 1 and 7 in the example) translates to edges. The operations, including the parameters, are captured by nodes, translating the rule set above into the operator graph shown in Figure 3. Note that the operator graph is to a certain degree independent of the way a designer chooses to group operators to rules. If the designer split the nested rules in line 1 and 7 into multiple rules, each containing a single operation, the resulting operator graph would still be the same.

Shape grammars often use random values and randomized rule selection to introduce stochastic variation into the generation process. Random parameters are simply added as parameters in the operator graph and can be defined by any random variable distribution.

```
(0,0)
vertex
(0,0)
vertex
(0,0)
rand2()
translate
rand2()
translate
rand2()
translate
3
polygon
5.0
extrude
```

(a) *Stack-based Program*



(b) *Operator Graph*

Figure 12: *A program written in a stack-based modeling language can be flattened to a data flow graph. As operations used in the operator graph only consume a single object at the time, operations which would require multiple inputs are replaced by a single object and selectors for the sub objects. Note that such a strategy is hardly ever needed in practice.*

Probabilistic rule selection can be added as another node. Finally, context-sensitivity can either be set up directly between nodes (as described before), or between entire subsets of nodes. CGA shape assigns priorities to rules, which translates into execution phases, *i.e.*, executing all rules with highest priority before rules of the next lower priority. To model this behavior with dependency edges, we add dependencies between all nodes of a higher priority to the next lower priority, forcing all nodes of a certain priority level to be executed before nodes of the next phase.

A.3 Stack-based Generation

Stack-based generation languages, like GML [Havemann 2005], also work on simple objects and offer a user a set of operations that can be applied to objects. As found by Havemann [2005], the similarity between GML and a generalized data flow network is striking. They can actually describe the same set of problems. By “flattening” out the stack (constructing the inputs and outputs of operations from the stack), it is possible to generate a graph of operations. However, there is a distinct difference to our operator graph. A single operation in a stack-based language can pop any number of objects from the stack. In the operator graph, only a single object (and a set of parameters) can be consumed by an operation. We do not allow for a join node that would combine multiple input objects. This limitation keeps objects independent of each other and allows for efficient parallel execution.

However, most operations in this type of procedural generation target a single input object. Additional inputs usually correspond to parameters controlling the operation itself, *e.g.*, the length of an extrude or the direction of a translation. Among the few exceptions that take multiple input objects are operations that combine low-level objects. For example, a variable number of vertices may be combined to a polygon. Each of those vertices can come from a chain of operations. While this is usually not the case, the operator graph can support such more complex setups, too. For example, instead of modeling the chains of operations that produce three vertices as input to a polygon generation, we can start with a polygon of three vertices and alter one after the other, as shown in Figure 12. In essence, we serialize operations from previously parallel paths. In this way, even operations which pop multiple complex objects from the stack can be represented by an operator graph.