EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

Parallel Generation of Architecture on the GPU

Markus Steinberger[1], Michael Kenzel[1], Bernhard Kainz[1], Jörg Müller[1], Peter Wonka[2], and Dieter Schmalstieg[1]

[1]Graz University of Technology, Austria
[2]King Abdullah University of Science and Technology, Saudi Arabia

Figure 1: With our parallel approach to procedural architecture, large cities can be generated in less than a second on the GPU. The city overview shows a scene with 38 000 low-detail buildings generated in 290 ms (left). The 520 buildings in the skyscraper scene consist of 1.5 million terminal shapes evaluating to 8 million vertices and 4 million triangles, all generated in 120 ms (center). The highly detailed skyscrapers are built using context-sensitive rules, to, *e. g.*, avoid overlapping balconies (right).

Abstract

In this paper, we present a novel approach for the parallel evaluation of procedural shape grammars on the graphics processing unit (GPU). Unlike previous approaches that are either limited in the kind of shapes they allow, the amount of parallelism they can take advantage of, or both, our method supports state of the art procedural modeling including stochasticity and context-sensitivity. To increase parallelism, we explicitly express independence in the grammar, reduce inter-rule dependencies required for context-sensitive evaluation, and introduce intra-rule parallelism. Our rule scheduling scheme avoids unnecessary back and forth between CPU and GPU and reduces round trips to slow global memory by dynamically grouping rules in on-chip shared memory. Our GPU shape grammar implementation is multiple orders of magnitude faster than the standard in CPU-based rule evaluation, while offering equal expressive power. In comparison to the state of the art in GPU shape grammar derivation, our approach is nearly 50 times faster, while adding support for geometric context-sensitivity.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1. Introduction

During the last years, the demand for larger, more realistic, and more vivid virtual environments has seen an upsurge. Games such as *Grand Theft Auto*, *The Elder Scrolls* series, or *World of Warcraft* offer a first glimpse into a fascinating future of truly-open virtual worlds. The appeal of films like *Thor* or *Man of Steel* partially stems from the visuals their characters' fantastic universes offer. Gamers and moviegoers expect these worlds to be unique, stunning, and studded with detail. In both the game and movie industry, crafting ever larger and yet more detailed environments is a time consuming, tedious task for artists. Procedural modeling has the potential to drastically reduce manual efforts for creating such huge environments: Given just a small set of rules, a procedural grammar can generate entire worlds from only a single input seed. If large parts of a new world could be generated with little human effort, artists could focus their creativity on the elements relevant to narrative and gameplay. Unfortunately, the grammar derivation of cities the size of Manhattan can take up hours, even on a modern CPU, greatly limiting the practical usefulness of such approaches.

With the stagnating increase in clock rate, the only way towards still higher performance seems to be going parallel. Procedural grammar derivation holds great potential for parallelization. During the generation of large scenes, millions of intermediate symbols have to be processed. A large portion of these symbols are independent and thus fit for massively parallel execution. Consisting of up to thousands of cores, the *graphics processing unit* (GPU) puts itself forward for parallel grammar derivation. Using the GPU also has the advantage that the generated geometry already resides on the rendering device. However, due to the peculiarities of modern GPU architectures, all previous attempts to GPU-based grammar derivation were only mildly successful, at best achieving moderate speed-ups over derivation on the CPU, while relying on substantially simplified grammars.

While the GPU potentially offers tremendous processing power, the workload presented by grammar derivation does not easily translate to the GPU execution model. The throughput-oriented architecture of the GPU heavily relies on hiding slow memory access with large amounts of arithmetic computation to achieve peak performance. Grammars, however, consist of small rules, performing just a few arithmetic computations each. During evaluation of large grammars, large numbers of intermediate symbols are created and need to be stored. Additionally, the choice of which rule takes effect might depend on random variables and complicated, nested branches. Tasks with little arithmetic load, but lots of complex control flow, are pathological for the single instruction, multiple data (SIMD) model of the GPU. Execution of branches on the same SIMD unit has to be serialized. Thus, parallelism is easily lost to suboptimal execution configurations. The fact that state of the art context sensitive shape grammars such as CGA shape [MWH*06] introduce complex interdependencies between rules further adds to the complexity of the problem. Taking into account geometric context, is essential to avoid generation of implausible scenery such as windows partially covered by a wall or a fence blocking a door. Such dependencies, however, imply that execution of the dependent rules has to be serialized. In sequential grammar evaluation, this problem can be ignored; one simply keeps choosing a rule for which all dependencies can be evaluated next. In a parallel derivation scheme, however, the additional complexity and the loss in parallelism due to context sensitivity are major issues.

In summary, we identify three obstacles for efficient evaluation of state of the art shape grammars on the GPU:

O1 Context sensitivity: State of the art shape grammars require the rule evaluation to consider the geometric context. Features such as occlusion queries and snap-lines need to be supported. Nontrivial inter-rule dependencies introduced by geometric context sensitivity make parallelization a difficult problem.

O2 Divergence: The SIMD architecture typical of a modern GPU only works well given homogeneous workloads. Grammar derivations involving many different rules introduce control flow divergence on SIMD units, easily leading to performance decreases of more than an order of magnitude.

O3 Memory access: Individual rules typically involve only a few arithmetic computations. Even if one can assign homogeneous sets of rules to fully occupy all SIMD units, the overhead of reading the state associated with each rule from and writing results back to global memory on the GPU can easily make up 90% of the total time.

To overcome these obstacles, we design a rule evaluation scheme for *parallel generation of architecture* (PGA) with GPU execution in mind. The PGA grammar provides *fully featured modeling capabilities* and is able to create visually appealing, realistic cities. Despite its expressive power, PGA lends itself to parallelization and *efficient execution* on an SIMD architecture. Slow global memory access is avoided by intelligent rule scheduling and local rule grouping strategies.

2. Related work

The current state of the art in procedural architecture modeling is still *CGA shape* [MWH*06]. The evolution that lead to CGA shape includes Stiny's original *shape grammars* [Sti75], *set grammars* [Sti82], and split operations for façade modeling [WWSR03] combined with transformation operations from *L-systems* [PL90]. Occlusion queries and snap-lines in CGA shape provide the ability to control grammar derivation based on local geometric context. Environmental influence is also exploited using synthetic topiary [PJM94], user designed curves [PMKL01], guided derivations [BŠMM11], interconnected structures [KK11], vector field guidance [LBZ*11], and self-sensitive L-systems [PM01]. Other noteworthy extensions to grammar-based modeling are more general terminal symbols [KPK11] and mesh refinement [Hav05]. There are a number of alternatives to grammar-based modeling that are also able to generate procedural models of high quality [LHL10, MM11, LCOZ*11].

Degree of parallelism in grammars. L-systems allow to compute the transformation for each symbol in a string in parallel, implying an average degree of parallelism proportional to the length of the string $\overline{S}$. Context sensitivity in L-systems is usually defined in terms of neighboring symbols in the string and does not affect parallelism. For architecture, split grammars are preferred over L-systems [WWSR03]. Split grammars use tree-shaped derivations. Because of parent-child relationships, the degree of parallelism is again proportional to the average number of shapes on one level $\overline{S}$. Context-sensitivity in CGA shape is formulated in a geometric, rather than a symbolic manner. While enhancing expressiveness of the approach, the geometric formulation imposes a partial ordering upon rule evaluation. Therefore, current implementations of CGA shape are based on sequential rather than parallel evaluation.

method	context sensitivity	max rewrites	parallelism	launches	SIMD control	memory writes
CGA shape [MWH*06]	geometry	∞	1	-	-	-
L-system [Mag09]	n.a.	∞	$\overline{S}/R$	$D \cdot R$	sort to bins	$\overline{S} \cdot D$
L-system [LWW10]	symbols	∞	$\overline{S}$	$4 \cdot D$	n.a.	$4 \cdot \overline{S} \cdot D$
split-grammar [MBG*12]	n.a.	< 8	B	1	n.a.	$\overline{S} \cdot D$
PGA	geometry	∞	$\approx 4 \cdot \overline{S}$	P	(local) grouping	$\approx \overline{S} \cdot P$

$\overline{S}$ average symbols on level; B buildings; D depth of tree/rewrites; R rules; P PGA phases

Table 1: While CGA shape is of high expressiveness and supports arbitrary context queries between rules, no parallel implementations of CGA shape exists. Previous parallel grammar implementations hardly support context sensitivity, limit the maximum depth of the execution tree/number of rewrites, and do not allow to express complex buildings (**O1**). Although PGA supports CGA shape-style context-sensitivity, our approach shows the highest degree of parallelism (**O1**), optimizes for the SIMD model (**O2**), only requires one kernel launch per phase (**O3**), and reduces the writes to slow global memory (**O3**). For a large city, one can observe: $\overline{S} > B > D > R > P$, with P being three in a typical PGA setup.

GPU-based grammar evaluation. Due to their implicit parallelism, L-systems and split grammars are candidates for parallel evaluation. While one could use CPU clusters [YHL*07], a GPU is more attractive for this purpose, as it provides inexpensive parallelism and the ability to render results directly. However, even the task of mapping simple L-systems to SIMD architectures leads to difficulties. A recent L-system generator [LWW10] for the GPU requires multiple expensive kernel launches and round trips to slow global memory in each iteration (**O3**) to count symbols, compute symbol output positions and execute actual rewrites. Moreover, neighboring elements in the string usually map to different rules, so assigning each symbol to one thread leads to divergence (**O2**). As a result, GPU evaluation of context-sensitive grammars on the GPU was reported to be even slower than on the CPU.

Lacz and Hart [LH04] evaluate split grammars using vertex and pixel shaders combined with a render-to-texture loop (**O3**). The main load in their system comes from global sorting of intermediate symbols. A similar grammar evaluation scheme using multi-pass rendering and GPU stream output was presented by Magdics et al. [Mag09]. They reduce divergence by employing a different shader for each output symbol, but still require many rendering passes (**O3**). Multi-pass rendering can be circumvented by using a fixed-size stack [MBG*12]. Due to the limits of GPU shaders, however, stacks are placed in slow global memory (**O3**). Additionally, the usage of stacks by itself leads to divergence (**O2**) and limits its parallelism to the number of axioms. More severely, stack size limits derivation complexity and thus makes it necessary to rely on instancing of detailed polygonal stock models for terminal shapes. Neither geometric context nor stochasticity is supported. Table 1 compares the above approaches to PGA.

Finally, sequential grammars can still be evaluated in parallel per pixel. Façade textures [HWA*10,MPHG11] are one example of such an approach. For increased detail,instances of prop models can be used as terminals [KBK13]. Approaches deriving grammars on a per pixel level are, of course, computationally extremely redundant for neighboring pixels.

3. Parallel generation of architecture

In the following discussion, we assume a stream processing model. Note that a shader-based implementation would face exactly the same issues. The central concept in the stream processing model is that of a *kernel function*, which is the program executed by each GPU thread. Kernel execution is initiated by a *kernel launch*. At the point of the kernel launch, the number of threads to be launched and their organization into blocks has to be specified. All threads within the same block are guaranteed to execute concurrently and are given access to the same region of local shared memory. The smallest unit of execution on the GPU is a *warp*. A warp is a group of threads that fits the SIMD width of the device. All threads within a warp execute in lockstep.

Parallelism is the key to fast rule derivation on the GPU. To introduce parallelism into the rule set of CGA shape while preserving context sensitivity, our parallel shape grammar for architecture (PGA) relies on a multi-phase evaluation model. To derive PGA grammars on the GPU, we propose an autonomous GPU rule scheduler based on local and global grouping mechanisms. Furthermore, PGA introduces intra-rule parallelism to further increase the efficiency of SIMD execution and the effectiveness of local rule grouping.

3.1. Context sensitive parallel evaluation

In CGA shape, geometric context sensitivity is supported through queries between shapes. These queries introduce dependencies among rules. For instance, all walls must be generated before any window, so that the window rule can query the walls to avoid partially occluded windows. CGA shape handles such dependencies in terms of priorities. Rule priorities make it very simple to determine a correct execution order by sorting. However, for parallel rule evaluation, we are interested in efficient identification of *independent* rather than dependent rules (**O1**).

Evaluation phases. To support queries between arbitrary rules while retaining parallelism, PGA replaces the notion of

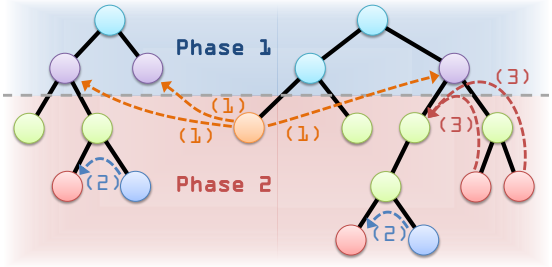


Figure 2: To maximize parallelism while still supporting geometric context sensitivity, PGA distinguishes three types of queries: (1) cross phase queries to any shape generated in a previous phase, (2) sibling queries between direct neighbors in the derivation tree, and (3) bilateral evaluation queries, in which the source and destination of a query have a common ancestor.

CGA shape priorities with evaluation phases. Every rule is assigned to a phase and can only generate shapes for the same or a later phase. Rules belonging to the same phase are evaluated concurrently. This approach differs from CGA shape by explicitly expressing independence of rules. The end of a phase marks a global synchronization barrier. Thus, any shape generated in a previous phase can always be subject of geometric queries. Separation into phases also has the advantage that acceleration structures can efficiently be built on the shapes generated in a completed phase. Each evaluation phase has to be implemented as separate kernel launch, because global synchronization on the GPU is currently only achieved between successive kernel launches. Intermediate shapes have to be stored in global memory as the contents of local shared memory are lost when a kernel finishes. Therefore, it is essential to keep the number of phases to a minimum. We find three to five evaluation phases to be usually sufficient. A separation into city layout, mass modeling and façade details—common in CGA shape—seems to work well in practice.

To help keeping the number of phases low, PGA also supports queries to shapes within the same phase given that certain criteria are met. The derivation of shape grammars can be viewed as forest of trees with axioms at the root and terminals forming the leaves [Sip06]. In city generation, the axioms are commonly building lots or cells forming the terrain. Queries introduce additional edges in this derivation graph. As outlined in Figure 2, PGA distinguishes between three types of queries, *sibling queries*, *bilateral evaluation queries*, and *cross-phase queries*, based on the relation of the rules involved.

Cross phase queries. As has already been mentioned, the separation into evaluation phases enables queries to any shape generated during a previous phase. We call these kinds of queries *cross phase queries*.

Sibling queries. If all shapes involved in a query have the same parent, we can evaluate the query in the parent shape, and forward the result to the querying child shape. For instance, a wall tile can check if it is located at a corner by querying the location of its neighbors, which are known to the floor shape generating the wall tiles.

Bilateral evaluation queries. Geometric context queries usually target shapes in close proximity, but these shapes are not necessarily limited to siblings in the symbolic derivation. However, we can still create an efficient query mechanism by exploiting the fact that spatially close shapes will likely correspond to symbols close in the derivation tree. To this aim, we analyze all possible generation paths for the shape being queried. If the querying and the queried shape have a common ancestor in the graph, we can pass this common ancestor along the path to the querying shape and compute the needed information independently of whether the queried shape has already been generated or not. Although this method involves redundant computation, it will be more efficient than introducing additional evaluation phases, as long as the common ancestor is not too far away. According to our experiments, for a distance of up to six rules, bilateral evaluation queries are preferable to introducing additional phases. An example for the use of such a query would be balcony doors: for a balcony, a door should be generated instead of a window.

Snap lines. Like CGA shape, we also allow the specification of snap lines. Rules splitting a shape into multiple new shapes automatically snap their splits to these lines, allowing elements of different mass model parts to be aligned (Figure 3). Our concept of evaluation phases facilitates the efficient implementation of snap lines. Snap lines specified in one phase can be referenced in all following phases, acceleration structures can efficiently be constructed between phases for fast access to all snap lines.

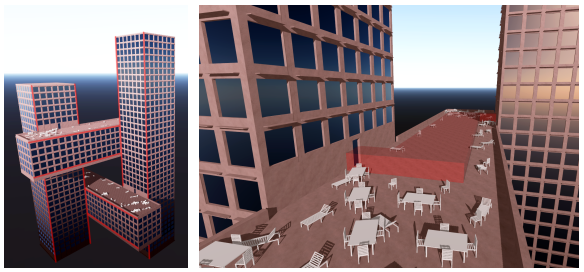


Figure 3: Model of the Cross-Towers being built in Seoul. (left) Using snap-lines, the floors and windows of the individual tower parts are aligned. (right) Non-terminal mass model shapes can be used to control the derivation. Geometric context sensitive rules keep a corridor (red) free of outdoor furniture and trigger the generation of a door.

3.2. Rule scheduling

The design of the PGA grammar allows for a high degree of parallelism, but additional measures are needed to address thread divergence (**O2**) and to minimize the number of kernel launches and accesses to slow global memory (**O3**).

A common strategy for executing tree-like algorithms is to operate on one level after the other interleaved with prefix sums [SHZO07] to determine the number of active nodes [ZGHG11]. This strategy has also been used by Lipp et al. [LWW10] for the derivation of L-systems. This approach requires a high number of kernel launches interrupted by read-backs from the GPU. Every kernel launch requires the current state to be flushed to slow global memory and read back from the GPU, which stalls the entire device. As only little computation is performed in each kernel, memory access cannot be hidden, having an additional, negative impact on overall performance.

Reduced number of launches. To overcome the issue of kernel launches, we propose a persistent threads approach [AL09]. For each phase, PGA keeps a queue of unprocessed shapes in global memory. Derivation is kicked off by launching a single kernel filling up the entire GPU. Each thread draws one shape from the queue and executes the rule associated with this shape. If new shapes are generated, they are immediately inserted into the queue. Instead of stopping after a shape has been processed, we keep drawing elements from the queue, until the queue is empty and the phase is completed. Following this strategy, the number of required kernel launches equals the number of phases, and the number of stalls can greatly be reduced. Since elements are concurrently removed from and inserted into the active queue, a simple, atomically operated buffer is not sufficient. Thus, we rely on an element-wise locked queue [SKK*12].

Rule grouping. A key factor for fast rule derivation is avoiding divergence (**O2**). When using the a single queue per phase, shapes of different kinds end up together in the same queue. As every thread draws new shapes from the queue, threads within the same warp end up processing shapes of different kinds, which leads to divergence. One possible way to avoid mixing shapes, would be sorting the queue according to shape types, as, *e. g.*, proposed by Lacz et al. [LH04] for their L-system derivation. However, as they noted, sorting consumes most of the time in a such a setup. Instead of storing the whole queue, we propose a dynamic rule grouping mechanism. Instead of individual shapes, we store groups of shapes of the same type in the global queues. The size of each group is chosen to fill an entire block of threads with work. In this way, all threads are supplied with the same type of shapes, execute the same rule, and no divergence will occur.

Shape grammars can contain hundreds of different rules. To dynamically collect shapes into groups, we therefore rely on a hash map using multiplicative hashing. For each rule type, this hash map stores a singly-linked list of shapes not yet formed into a group. Whenever a new shape is generated, we allocate memory for a list node containing the shape using fast dynamic memory allocation [SKKS12]. We then copy the shape to this list node and append the node to hash map entry associated with the rule needed for the shape. To move groups from the hash map into the global queues, we use a separate warp which continuously iterates over the hash map. If there are enough shapes to form a complete group in one of the lists, it takes these elements out of the map and inserts them into the queue. If the global queue is about to run low on elements, we also move incomplete groups from the hash map to the queue to keep the GPU occupied and ensure all generated shapes will be processed. As we reference shape groups only by pointer, they can be passed around very efficiently. After a shape group has been processed, the memory used to store the shapes comprising the group is freed.

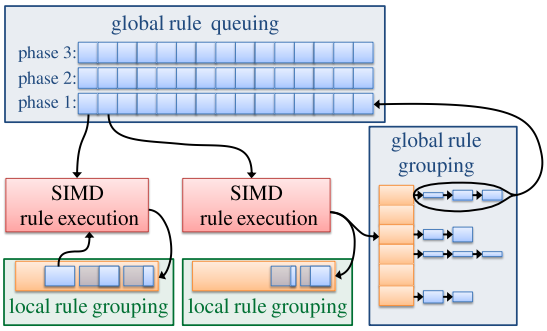


Figure 4: To efficiently manage rule derivation, we use a two level approach. Global grouping (blue) employs a hash map to merge rules of the same type, before they are inserted into the execution queues. To avoid the costly transfer to global memory, we introduce a local grouping mechanism (green). Each block keeps a list of a few rules in local shared memory. These rules are executed before new ones are drawn from the global queue. If local grouping runs out of memory or if there are not enough rules available, we resort to global grouping.

Local grouping. Although global rule grouping already boosts performance by orders of magnitude, the overhead of dynamic memory allocation and writing shapes to global memory can still be significant (**O3**). Multiple threads executing the same rule are likely to generate new shapes for the same rule. Therefore, a further optimization is to group shapes in local shared memory, instead of directly sending each shape to the global grouping mechanism. As dynamic shared memory allocation is infeasible due to fragmentation issues, we refrain from using hash map-based grouping in shared memory. Additionally, the limited size of shared memory hinders grouping based on fixed size lists for all rules. Thus, we propose a dynamically managed list of rule groups. When the first shape is generated, we assign a region of shared memory large enough to hold a shape group for the respective

rule type. Each such region is preceded by a header identifying the rule type, the number of shapes in the group, and a pointer to the beginning of the next group in shared memory. When the next shape is generated, we search through all groups using the information found in each header. If a matching group is found, we atomically increase the group's shape counter and store the shape at the respective position. If no matching group is found or all groups are full, we try to add a new group to the list. If no memory is left for a new group, we fall back to the global grouping solution.

Simple locks cannot be used in shared memory, as there is no safe way to avoid deadlock situations given the characteristics of the hardware scheduler. Instead, by requiring that all free memory be initialized to zero, we can rely on an atomic *compare-and-swap* (CAS) operation to concurrently allocate new groups. We allocate a new group by trying to write its header using CAS. If the CAS returns zero, we can be sure that no other thread added a group concurrently, otherwise, we have to retry. To reduce the number of retries, we use a warp voting scheme to determine a single thread per warp to interact with the grouping data structure.

In every iteration of the persistent threads main loop, we first check shared memory. If we find a complete group, we immediately consume it and execute the associated rule. After taking a group out of the list, we run a compaction step, closing the gap left behind by moving the remaining contents of the list to the front and zeroing the memory then available at the back. If no complete group is found, we draw a group from the global queue. If the memory assigned to local grouping is about to run full, we flush complete groups to the global queue, so that we retain the ability of capturing newly generated shapes in fast local memory.

The whole local and global rule grouping scheme is outlined in Figure 4. There are several advantages to this strategy: First, the number of reads and writes to global memory is minimized. In an optimal case, shapes are only written to global memory, if their rules belong to later phases. As local shared memory is nearly as fast as registers, the overhead of local rule grouping is very low compared to global rule grouping. Second, the available memory is used in a very effective way. During derivation, many different rules will eventually be encountered, but only a small number of different shapes need to be kept ready at any point in time. With our dynamic scheme, the entire memory could be used for a single shape type if needed. Third, if the grammar is strongly expanding, and many full groups are generated quickly and flushed to the global queue, we at least avoid the overhead of global grouping.

3.3. Operator-level parallelism

As mentioned before, a high degree of parallelism is the key to high performance on the GPU. We can go further than just executing multiple rules in parallel, as parallelism is inherent to many operators in CGA shape. For instance, the *Repeat* operator repeats a given shape as many times as it fits into the parent shape along a certain dimension. Instead of using a single thread to generate all n instances of the new shape, we could use n threads to generate each instance in parallel. In the following, we identify which operators can be parallelized, show the techniques used by PGA, and discuss how such *operator-level parallelism* influences overall performance.

Static parallelism. A large number of operators proposed in CGA shape can be distributed to a constant number of threads only depending on the type of the input shape. Among those operators are *Component Split*, *Generate Geometry*, and *Subdivide*. *Component Split* takes an input shape and generates a new shape for each face, edge, or vertex of the input shape. One thread can be assigned to each output shape. *Generate Geometry* generates the vertices and indices required for rendering a shape and inserts them into the vertex buffer and index buffer. Again, one thread can be use to compute a single vertex and a single index. We use the greatest common divisor of vertex count and index count to determine the number of threads. For instance, the geometry of a box is generated by twelve threads (24 vertices, 36 indices), each thread being responsible for two vertices and three indices. As a beneficial side effect, in this case, parallelization will enable coalesced memory access. *Subdivide* fits a fixed number of shapes into the parent shape. One thread can be used per shape.

While operator-level parallelism is completely transparent to the designer writing rules in PGA using these operators, great care has to be taken when implementing the operators themselves. Thread divergence must be avoided at all costs, as it will obliterate any potential gains on SIMD architectures. Consider the example C++ code for splitting a box into its six faces using six threads with thread id (`tid`) 0 to 5:

```
1  void SplitFaces(Box& box, Out s)
2  {
3    int select = (tid & 0x1) * 2 - 1;
4    int xCode = (tid < 2) * select;
5    int yCode = (tid & 0x2)/2 * select;
6    int zCode = (tid & 0x4)/4 * select;
7
8    float3 n(xCode, yCode, zCode);
9    float3 pos = box.pos + box.size * n / 2;
10
11   float3 s0(yCode, 0, xCode + zCode);
12   float3 s1(0, xCode + zCode, yCode);
13   float2 size;
14   size.x = dot(box.size, abs(s0));
15   size.y = dot(box.size, abs(s1));
16
17   Rotation r(0.5f*yCode * pi,
18     (xCode == -1) * pi + 0.5f*zCode * pi,
19     0);
20   Rotation rot = rot * box.rotation;
21
22   generate<Quad>(s, size, pos, rot);
23 }
```

Using nothing but arithmetics, we generate a vector pointing into the direction of each thread's face (line 3-6) without the need for branching. Based on this vector, we can determine the position (9) as well as the size (11-15) of the face to be generated. To provide a common local coordinate system, we compute a rotation using Euler angles for each face (17-20). Finally, all six quads are generated concurrently (22).

Input-dependent parallelism. For some operators, the number of generated shapes depends on the dimensions of the input shape. In theses cases, an optimal assignment of threads cannot be determined statically. An example for such an operator is *Repeat*. *Repeat* can, *e. g.*, be used to split a building into floors. Obviously, a skyscraper will have more floors than a suburban house, and, thus, a higher number of threads could be used. While it would be possible to dynamically assign threads to the execution of rules and provide an optimal thread count, our experiments have shown that a completely dynamic setup is not preferable. Dynamically assigning threads to rules complicates rule grouping and introduces a significant overhead during rule evaluation. Thus, we rely on a semi-dynamic setup: We provide implementations for a subset of all possible thread counts. When a new shape is generated, we check how many threads would be optimal and assign the shape to the variant with the next higher thread count. In this way, some threads might run idle, but their number is usually low. The rule scheduler simply treats these variants like different rules, and grouping works as described before.

If the number of shapes to be generated is larger than the thread count (`tc`) of the largest operator variant available, each thread has to generate more than a single shape. For *Repeat*, an implementation could look as follows:

```
1  void RepeatX(Box box, float width, Out s)
2  {
3    int N = box.size.x / width;
4    for (int i = tid; i < N; i += tc)
5    {
6      float3 size = box.size;
7      size.x = size.x / N;
8      float3 pos = box.pos;
9      pos.x -= 0.5f * box.size.x;
10     pos.x += (i + 0.5f) * size.x;
11     Rotation rot = box.rotation;
12     generate<Box>(s, size, pos, rot);
13   }
14 }
```

In every iteration, *tc* new shapes are generated in parallel. Threads can diverge only in the last iteration if N/tc is not integer.

Non-parallel operators. There are operators that cannot be parallelized themselves. Examples for such operators are *Scale*, *Translate*, and *Rotate*. However, these operators are often used in sequence with operators that can be parallelized, *e. g.*, scaling the shapes generated by *Repeat*. In these cases, parallel and non-parallel operators can be combined into a single rule. Note that the same amount of parallelism is available in a combined rule as in separate rules: If a *Repeat* generates four shapes, it can be evaluated by four threads. If another rule was used to scale the resulting shapes, these rules would be executed for each shape, providing work for four threads. If both rules are combined into one rule, the combined rule would still be executed by four threads. However, using a combined rule, no intermediate symbols are generated, saving the overhead of memory access and rule grouping that would otherwise be involved. In PGA, we automatically analyze the provided rule set and substitute combined operators for maximum efficiency.

Benefits. Operator-level parallelism brings about significant arithmetic overhead. However, the dominating bottleneck in grammar derivation usually is memory access. Therefore, any additional arithmetic overhead is usually hidden by memory latency, and the numerous benefits of operator-level parallelism prevail: First, thanks to the additional parallelism, even single buildings can achieve reasonable GPU utilization. Second, operator-level parallelism leads to very efficient memory access patterns. Third, fewer shapes are needed to fill an entire block of threads. Therefore, local rule grouping requires less shared memory. Additionally, as fewer shapes are needed to complete a group, groups are more likely to spill directly to the global queue instead of having to go to global grouping. Fourth, as local groups are completed more easily, shapes need to be taken from the global queue less often, reducing the number of global memory accesses. Finally, threads involved in the derivation of the same shape hardly diverge.

3.4. Example

Figure 5 demonstrates the advantages of PGA rule scheduling by the example of two façades of a simple building. Seven rules are used in this example. We split each façade into two floors. Each floor is split into tiles, each tile consists of a window and surrounding walls. Using context sensitivity, an alternate floor rule that generates a door in place of a window is selected at street level. In contrast to a traditional evaluation scheme using a single thread per rule, PGA can distribute rule evaluation amongst multiple cores (Figure 5 left). Considering the complete derivation (Figure 5 center and right), the advantages of PGA are obvious.

4. Results

To explore the characteristics of the various components of PGA, we compare the performance of different implementations. As baseline, we use the CPU implementation of the original CGA shape by Müller et al. [MWH*06] found in the software *CityEngine*. Because *CityEngine* is not open source, a direct measurement of the generation time is not possible. As our measurements also include rendering time, we reduce

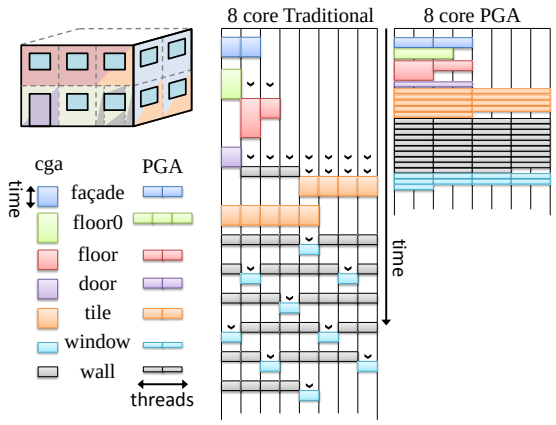


Figure 5: Comparison between a traditional GPU implementation of CGA shape and PGA, which uses multiple threads per rule.. A very simple building is specified by seven rules (left). The traditional implementation frequently leaves most SIMD units unoccupied (center) and is ridden by divergence (black chevrons). The need to flush and read back shapes from global memory introduces additional idle time for the whole GPU. With parallel rule evaluation and rule grouping, PGA can execute much more efficiently on SIMD units (right). Divergence and slow global memory access are avoided. Note that the same amount of work (colored area) is done in both cases, but PGA achieves more efficient scheduling.

the measured time by 1% as a conservative estimate. To avoid undesired side effects from memory management, we split the CGA shape evaluation into smaller generations of 400 buildings each. We compare a sequential CPU implementation *Seq* of the PGA rule set, which only performs combining of parallel and non-parallel operators. We further provide *CPU PGA*, a highly competitive, full implementation of PGA on the CPU. On the GPU, we implemented a traditional multi-pass derivation scheme launching a kernel which uses one thread per shape to be derived (*Kernels*). Axioms as well as intermediate shapes are buffered in global memory between kernel launches. Parallel execution is enabled by the PGA phase model and parallel/nonparallel operator combining is also performed. In a second GPU implementation (*Persist*), the multi-pass approach is replaced with a persistent threads implementation using global rule grouping to reduce divergence. Finally, we compare the full PGA implementation including local rule grouping and operator-level parallelism (*PGA*). All measurements were run on the same machine with an Intel Core i7-940 Quad Core CPU (2.9 GHz) and an NVIDIA Quadro 6000 GPU.

We selected a variety of test scenes, including small and large setups, geometry-heavy and rule-heavy derivations, using only context-free or also context-sensitive rules, and scenes with and without snap-lines. The characteristics of each scene are outlined in Table 3 and images from each scene can be seen in Figures 1 and 6.

5[htb]

	axiom	nodes	term	vert	tris
Tree4,3	1	121	283	3896	5232
Tree8,4	1	9k	23k	6M	10M
Residential*	1	297	2423	99k	62k
Cross-Towers*†	1	3327	17k	363k	182k
Skyscrapers*	523	311k	1.5M	8.10M	4.05M
City Overview*	38k	239k	851k	10.07M	5.84M
Suburban*	4244	796k	6.0M	278.8M	167.8M

* context sensitive; † snap lines

Table 3: Our test scenes feature up to 38 k axioms, 800 k non-terminals (nodes), 6 M terminals (term), 280 M vertices (vert), and 170 M triangles (tris). We cover different aspects, like context-sensitivity, snap-lines, and varying shapes of derivation trees. For the tree dataset, the numbers correspond to the number of recursions and tree branches. City Overview and Skyscrapers are shown in Figure 1; Cross-Towers in Figure 3; the trees, Residential, and Suburban in Figure 6.

Generation times are summarized in Table 2. Our optimized *CPU PGA* implementation (four threads) achieves speed-ups of 130–3500 over *CGA shape*. Making use of hyper-threading by increasing the thread count reduced the performance, which indicates that the four cores are well utilized. With the fully-featured GPU implementation, we achieved speedups between 400 and 12 000 in comparison to *CGA shape*, between 14 and 2700 over *Seq*, and between 4 and 7 over *CPU PGA*.

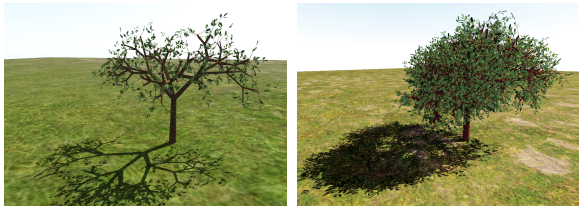
The simplest test scenes are the tree datasets, which consist of a few context-free rules only. We chose these datasets to draw a direct comparison to the state of the art in GPU shape grammar derivation: *GPU Shape Grammars* [MBG*12]. They report a generation time of 40 ms for their tree with 280 terminals (Tree4,3) on a GPU with processing power equivalent to ours. With *CPU PGA*, we derive the same tree in 4.5 ms. The full GPU implementation of PGA needs only 0.84 ms, which is nearly 50 times faster. Within a time frame of 20 ms, PGA can generate a tree with 23 000 terminals and 10 M triangles (Tree8,4). The speedup of GPU over CPU implementations is higher in this testcase, as the tree provides a large amount of parallelism. As the number of different rules is low, and a maximum of ten kernel launches is required by *Kernel*, *Persist* wins in performance by only 50%. However, by avoiding global memory access with local grouping and exploiting operator-level parallelism, *PGA* shows performance gains of another 150%. Starting from a single axiom, operator-level parallelism has the biggest effect during evaluation of lower levels of the derivation tree. The reduced number of global memory accesses makes up for the larger part of the performance increase.

		CPU		GPU				
	CGA shape	Seq	CPU PGA	GPU SG	Kernels	Persist	PGA	speedup
Tree4,3		11.7	4.49	40	2.77	1.55	0.84	
Tree8,4		53546.1	135.40		76.31	48.87	19.29	
Residential	1090	77.8	8.36	n. a.	12.91	4.90	2.31	472
Cross-Towers	6070	2092.8	24.12	n. a.	32.89	11.86	6.97	871
Skyscrapers	807300	9825.6	515.77	n. a.	689.27	201.62	116.78	6913
City Overview	3495800	16573.0	996.33	n. a.	608.69	323.15	289.43	12078
Suburban	4651400	53605.3	13115.90	n. a.	16041.19	6689.11	4551.21	1022

Table 2: Generation times in ms for different methods and test scenes. *CGA shape* corresponds to the CGA shape implementation found in *CityEngine*; the measurement setup is described in the text. *Seq* is a single-threaded CPU implementation of the PGA rule set, which only performs automatic combining of parallel and non-parallel operators. *CPU PGA* is a highly optimized, parallel implementation of PGA for the CPU. On the GPU side, we compare our full PGA implementation to GPU Shape Grammars (*GPU SG*), a kernel-based implementation based on the PGA rule set (*Kernels*), and a persistent threads implementation with our global rule grouping (*Persist*). *PGA* additionally uses local grouping and multiple threads per rule. The speedup measures compare *PGA* to *CGA shape*.

Rule derivations that require geometric context-sensitivity are more challenging to the GPU. The context-sensitive Residential and Cross-Towers are generated in 2 ms and 7 ms on the GPU, respectively. The CGA shape baseline implementation takes nearly 500 and 900 times as long. Even our *CPU PGA* implementation is 130 and 250 times faster than CGA shape, indicating that our strategies also help to vastly improve performance on the CPU. Operator-level parallelism and local rule grouping can increase performance by a factor of 2.1 for Residential and 1.7 for Cross-Towers. We explain this difference by the structure of their derivation trees. While Residential has many different rules expanding more slowly, the high number of floors and façade tiles of the Cross-Towers provide a large number of shapes right at the beginning, leaving less to gain from additional parallelism.

The Skyscrapers are a first test scene starting out with more than one axiom, providing more parallelism already at the beginning of grammar evaluation. Our optimized CPU implementation *CPU PGA* is more than 1500 times faster than *CGA shape*; *PGA* on the GPU is nearly 7 000 times faster. As all skyscrapers are based on the same rules, there is sufficient parallelism available throughout the entire derivation. Still, operator-level parallelism can increase performance by nearly 100%. We account that to more efficient memory access patterns and reduced divergence. The City Overview testcase consists of simple shapes only, which are created by a small number of rules. Thus, the performance increase of *Persist* and *PGA* in comparison to a kernel-based implementation is not as high as in Skyscrapers. However, in comparison to *CGA shape*, we achieve a speedup of up to four orders of magnitude. The most terminal and geometry heavy testcase is Suburban. Performance in this setup is mainly limited by memory access and thus, the speedups achieved with *PGA* are lower. Still, *PGA* is three times faster than *CPU PGA* and 1000 times faster than *CGA shape*.



(a) Tree4,3 (b) Tree8,4

(c) Residential (d) Suburban

Figure 6: The tree examples use the same rule set. The small tree consists of 283 terminals, while the full tree has 23 000. The Residential building requires many different rules, leading to 2400 terminals. The Suburban scene contains 4 000 full-detail residential buildings of varying size, generating 170 million triangles.

5. Conclusion

We have presented the first parallel shape grammar for the GPU which does not sacrifice expressive power of the grammar for the sake of accommodating rigid GPU execution models. By distributing the evaluation of a single rule to more than a single thread, we take advantage of the SIMD architecture of the GPU and keep up GPU utilization in situations low on parallelism. By explicitly modeling independence rather than dependence through the concept of evaluation phases, we provide geometric context-sensitivity while still being able to

extract parallelism from the derivation tree. PGA is the first context-sensitive parallel shape grammar for architecture that can be executed on the GPU. Our GPU implementation is significantly faster than its CPU counterpart, deriving shape grammars up to 10 000 times faster than current standard approaches on the CPU and 50 times faster than the state of the art in GPU shape grammars. Using our parallel shape grammar, the authoring and editing process of procedurally generated architecture can be enriched with instant feedback.

Given these benefits, we see PGA as a significant step forward in making procedural content generation more attractive for the video game industry and related domains. Our grammar is compatible with CGA shape, thus, existing rule sets can be ported to PGA with little effort. While we now can completely generate a detailed city in less than a second, there is still more we can do to bring procedurally generated content to life on screen. In our follow up work, we show how we can boost PGA by taking into account visibility, levels of detail, and frame-to-frame coherence, empowering us to derive detailed cities with more than 50 000 buildings on the fly during real-time rendering [SKK*14].

Acknowledgments This research was funded by the Austrian Science Fund (FWF): P23329.

References

[AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. High Performance Graphics* (2009), ACM, pp. 145–149. 5

[BŠMM11] BENEŠ B., ŠTAVA O., MĚCH R., MILLER G.: Guided Procedural Modeling. *Comp. Graph. Forum 30*, 2 (2011), 325–334. 2

[Hav05] HAVEMANN S.: *Generative Mesh Modeling*. PhD thesis, TU Braunschweig, 2005. 2

[HWA*10] HAEGLER S., WONKA P., ARISONA S. M., GOOL L. J. V., MÜLLER P.: Grammar-based Encoding of Facades. *Comp. Graph. Forum 29*, 4 (2010), 1479–1487. 3

[KBK13] KRECKLAU L., BORN J., KOBBELT L.: View-Dependent Realtime Rendering of Procedural Facades with High Geometric Detail . *Comp. Graph. Forum 32*, 2pt1 (2013). 3

[KK11] KRECKLAU L., KOBBELT L.: Procedural Modeling of Interconnected Structures. *Comp. Graph. Forum 30* (2011). 2

[KPK11] KRECKLAU L., PAVIC D., KOBBELT L.: Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Comp. Graph. Forum 29* (2011), 2291–2303. 2

[LBZ*11] LI Y., BAO F., ZHANG E., KOBAYASHI Y., WONKA P.: Geometry Synthesis on Surfaces Using Field-Guided Shape Grammars. *IEEE Trans. Visualization and Computer Graphics 17*, 2 (2011), 231–243. 2

[LCOZ*11] LIN J., COHEN-OR D., ZHANG H., LIANG C., SHARF A., DEUSSEN O., CHEN B.: Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph. 30*, 6 (2011), A183. 2

[LH04] LACZ P., HART J.: Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors* (2004), pp. 23–23. 3, 5

[LHL10] LEFEBVRE S., HORNUS S., LASRAM A.: By-example synthesis of architectural textures. *ACM Trans. Graph. 29* (2010), A84. 2

[LWW10] LIPP M., WONKA P., WIMMER M.: Parallel Generation of Multiple L-systems. *Computers & Graphics 34*, 5 (2010), 585–593. 3, 5

[Mag09] MAGDICS M.: Real-time Generation of L-system Scene Models for Rendering and Interaction. In *Spring Conf. on Computer Graphics* (2009), Comenius Univ., pp. 77–84. 3

[MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Comp. Graph. Forum 31*, 7-1 (2012), 2087–2095. 3, 8

[MM11] MERRELL P., MANOCHA D.: Model Synthesis: A General Procedural Modeling Algorithm. *IEEE Trans. Visualization and Computer Graphics 17* (2011), 715–728. 2

[MPHG11] MARVIE J.-E., PASCAL G., HIRTZLIN P., GAEL S.: Render-Time Procedural Per-Pixel Geometry Generation. In *Graphics Interface* (2011), pp. 167–174. 3

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural Modeling of Buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 2, 3, 7

[PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic Topiary. In *Proc. SIGGRAPH 94* (1994), pp. 351–358. 2

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990. 2

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proc. SIGGRAPH 2001* (2001), pp. 301–308. 2

[PMKL01] PRUSINKIEWICZ P., MÜNDERMANN L., KARWOWSKI R., LANE B.: The Use of Positional Information in the Modeling of Plants. In *Proc. SIGGRAPH 2001* (2001), pp. 289–300. 2

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan primitives for GPU computing. In *Proc. Symposium on Graphics Hardware* (2007), pp. 97–106. 5

[Sip06] SIPSER M.: *Introduction to the Theory of Computation*, vol. 2. Thomson Course Technology Boston, 2006. 4

[SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph. 31* (2012). 5

[SKK*14] STEINBERGER M., KENZEL M., KAINZ B., WONKA P., SCHMALSTIEG D.: On-the-fly generation and rendering of infinite cities on the GPU. *Comp. Graph. Forum 33* (2014). 10

[SKKS12] STEINBERGER M., KENZEL M., KAINZ B., SCHMALSTIEG D.: ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing* (2012). 5

[Sti75] STINY G.: *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, 1975. 2

[Sti82] STINY G.: Spatial Relations and Grammars. *Environment and Planning B 9* (1982), 313–314. 2

[WWSR03] WONKA P., WIMMER M., SILLION F. X., RIBARSKY W.: Instant Architecture. *ACM Trans. Graph. 22* (2003), 669–677. 2

[YHL*07] YANG T., HUANG Z., LIN X., CHEN J., NI J.: A Parallel Algorithm for Binary-Tree-Based String Rewriting in L-system. In *Proc. International Multi-symposiums of Computer and Computational Sciences* (2007), pp. 245–252. 3

[ZGHG11] ZHOU K., GONG M., HUANG X., GUO B.: Data-parallel octrees for surface reconstruction. *IEEE Trans. Visualization and Computer Graphics 17* (2011), 669–681. 5