Mark Dokter

Rule Scheduling for Shape Grammar Evaluation on the GPU

Master's Thesis

Graz University of Technology

Institute for Computer Graphics and Vision
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Dieter Schmalstieg

Supervisor: Dr. Markus Steinberger

Graz, April 2014

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Acknowledgments

I would like to thank the following people who supported me, in chronological order of them doing so.

My parents, who gave me a wonderful start in life and who will always stand behind and care about me. The rest of my family and friends for offering their time, their advice or their patience. My mentor Dieter Schmalstieg, who always encouraged me to get on with my studies and made it possible for me to tackle work for the institute and university simultaneously. Furthermore, his enthusiasm about computer graphics was and is very inspiring and he knows how to challenge people to push hard and give their best. My supervisor, Markus Steinberger. Not only did he lay out the foundations on which this thesis is built, he has also been very patient, supportive and always quick in response to my inquiries. Michael Kenzel, who supported me with his expertise during the work on this thesis and who was also ever swift in response to my questions, even at times far beyond the usual working hours.

Abstract

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

Kurzfassung

Aufgrund der steigenden Nachfrage an computergenerierten Inhalten ist prozedurale Modelierung ein wichtiges Thema für die Spiele- und Filmindustrie geworden. Das Erstellen großer Mengen digitaler Inhalte benötigt viel Handarbeit. Mit einem prozeduralen Regelsatz können ganze Welten vom Computer erstellt werden. Allerdings kann das Ableiten eines solchen Regelsatzes, zum Beispiel für eine grosse Stadt, mit CPU-basierten Methoden mehrere Stunden dauern, was schnelle Designentscheidungen unmöglich macht. In dieser Arbeit untersuchen wir verschiedene Strategien, um prozedurale Modellierung mittels CUDA auf einem Graphikprozessor auszufüren. Wir vergleichen sowohl einen Persistent Megakernel Ansatz mit einfachen Kernelaufrufen als auch verschiedene Strategien zur Einteilung der Abfolge der Regelableitung. Weiters erforschen wir die Vor- und Nachteile des Einsatzes von vorkompilierten Regelsätzen gegenüber zur Laufzeit interpretierten Regeln.

Contents

List of Figures

List of Tables

1. Introduction

In this work, we investigate a variety of rule derivation methods for procedural generation of architecture in particular and other miscellaneous geometric shapes. We are building on the foundations laid out by Steinberger, Kenzel, Kainz, J. Müller, et al. [2014], focusing on the drawbacks of their approach. They relied solely on precompilation of rules, and the implementation heavily involved dynamic memory allocation, which is know to be slow on the GPU [Steinberger, Kenzel, Kainz, and Schmalstieg, 2012].

1.1. Motivation

Generating graphical content in an automated fashion has become increasingly important during the last decade. Many recent computer games offer vast open virtual worlds, where the player can freely explore the environment. In the latest version of Grand Theft Auto, for example, the area is not limited to a single city, but also includes its surroundings, multiple square kilometers of content. A player is free to explore these worlds, walking, driving or even flying. In big movie productions, like Roland Emmerichs ”2012”, huge scenery of cities are shown and destroyed within seconds. Those are examples of extensive use of digitally created content, which requires vast amounts of manual labor to produce. By automating content creation as much as possible, artists can spend more of their time on elements relevant to narrative and gameplay, rather than on creating peripheral scenery.

1.2. Procedural Modeling

Procedural modeling is the process of constructing 3D models by describing the construction process in a set of rules. It is even possible to generate them on the fly [Steinberger, Kenzel, Kainz, J. Müller, et al., 2014], rather than having an artist do all the work by hand and loading the models into memory before they can be used. The task of creating reoccurring, parameterizable objects like houses or trees is well suited for this kind of modeling. Rules for model production can be defined in a shape grammar. Starting from an initial set of shapes, these rules iteratively add detail to the scene. After fully evaluating such a rule set, the geometry, which describes the entire scene, is ready for rendering. However, grammar derivation for thousands of buildings can take many hours on a conventional CPU, even with several cores and a high clock rate.

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

1.3. Organization of this Work

As will be discussed in chapter 2 on related work, various attempts of mapping this challenging task to a graphics processor have already been made. In chapter 2.1, we will give an overview of the kind of shape grammar used for defining rules to construct buildings procedurally. Chapter 4 describes two techniques of controlling the GPU rule evaluation process in more detail. The two approaches used are launching several successive kernels for each derivation step and deriving the entire scene using a single kernel launch in a persistent threads megakernel method. Another aspect of rule derivation is investigated in chapter 5: Precompiling rules and interpreting rule sets. While precompiled rule sets are expected to be faster in the derivation process, they do not offer the same flexibility to change rule sets quickly. In chapter 6, we discuss how the generated geometry data is rendered using three different methods: conventional non-instanced rendering, instanced rendering using special (hardware accelerated) OpenGL draw calls and a hybrid method we call software instancing. Implementation details are discussed in chapter 7. We describe the rule derivation process and give small code samples. After setting the scene in the previous chapters, the results of the investigation of the aforementioned techniques are presented in chapter 8. We evaluate five different test scenes on all possible combinations of derivation methods, rendering techniques and task scheduling approaches. Contrarily to our expectations, the precompiled rule derivation process did not work as well as expected for all test scenarios. It turns out that rule interpretation at run time also has its benefits in some cases.

In chapters 9 and 10, open research questions, future work and conclusions drawn from our findings are presented. Code samples, rule sets and more detailed result tables can be found in the appendix.

2. Related Work

2.1. Shape Grammars

Our grammar incorporates several concepts from split grammars [Wonka et al., 2003] and is loosely based on CGA [P. Müller et al., 2006], which is a sequential grammar, allowing spatial distribution of components. It is currently the most widely used grammar for procedural architecture modeling. CGA is based on Stiny's work on *shape-grammars* [G. Stiny, 1975] and *set-grammars* [G. Stiny, 1982]. Furthermore, it uses split operations for facade modeling as proposed Wonka et al. [2003] and transformation operations similar to *L-systems* [Prusinkiewicz et al., 1990].

In this work, we focus on the basic concepts behind rule scheduling, thus we use a simplified version of CGA. Our grammar does not support context sensitive rules or snap lines, similar to the grammar used by GPU Shape Grammars [Marvie et al., 2012].

2.2. Split Grammars

Split grammars, introduced by Wonka et al. [2003], are specialized set grammars, which impose restrictions on the allowed shapes and operations to make the grammar simple enough for automated derivation, but sufficiently expressive to allow the modeling of many different objects.

A split grammar builds on the notion of shapes and set grammars. A *shape* can be defined [George Stiny, 1980] as follows:

Definition 2.2.1 *A shape is a limited arrangement of straight lines in three-dimensional Euclidean space.*

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

A *grammar* can be defined as a set of production rules R on a set of symbols U, using the following definition similar to the one given by Wonka et al. [2003]:

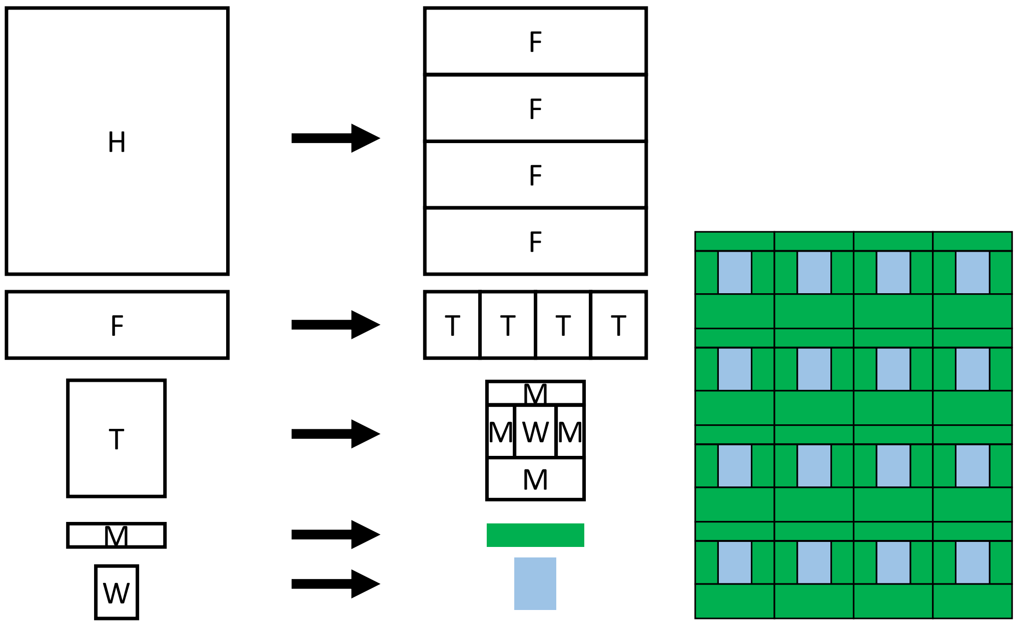
Definition 2.2.2 *A grammar $G = (N, T, R, I)$ consists of the non-terminal symbols $N \subseteq U$, the terminal symbols $T \subseteq U$, a set of initial symbols (axioms) $I \subseteq N$ and a set of rewriting rules (productions) $R \subseteq U \times U^*$.*

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

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

In the case of shape grammars, the production process works on shapes. Rules thus describe geometry operations on the input shape, generating any number of new shapes. For a grammar to be a split grammar, only two kinds of rules are allowed [Wonka et al., 2003]:

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



(a) A Simple Split Grammar (b) Result of the rule set (a)

Figure 2.1.: A split grammar operates on a set of shapes, each associated with a symbol. Rules (a) replace one shape by a group of other shapes. Using split grammars, more complex objects can be generated from very simple rules.

These restrictions allow for a simple grammar derivation, as rules can only influence a constrained volume, as shown in Figure 2.1. Furthermore, every shape can be treated independently of the other shapes in the active set. This allows for a fully parallel production process. CGA, and consequently our grammar as well, do not have these restrictions, and shapes can also increase in size, be moved or extruded.

2.3. Procedural Modeling

Approaches augmenting the functionality and usefulness of shape grammars exist on more general non-terminal symbols [Krecklau et al., 2011] and mesh refinement [Havemann, 2005].

Apart from grammar based approaches to the procedural generation of geometry, other methods can be used to obtain high quality models [Lefebvre et al., 2010; Lin et al., 2011; Merrell et al., 2011].

Parallel grammar derivation has been investigated in various approaches which differ greatly in their strategy. Deriving L-systems on CPU clusters has been done by Yang et al. [2007]. Considering the inherent parallelism of the algorithm, CPU clusters seem to be a good idea. However, when using a GPU, the results are already in the memory of the graphics card, which is obviously more convenient for rendering.

A recent L-system generator for the GPU has been proposed by Lipp et al. [2010]. In their work, they used multiple kernel launches to implement iterative rewriting of L-systems. Using a single thread per symbol without sorting the symbol stream has some drawbacks. First, memory accesses can become problematic, if symbol sizes are not coherent. Second, thread divergence, which is the effect of threads taking distinct execution paths, results in different times the threads need to finish their work. On a GPU, where it is desirable to have as many threads occupied as possible at any point during run time, this effect causes some threads to wait on others which might take longer. This drastically impacts performance. And third, the management overhead for keeping track of where to store symbols quickly becomes a dominant factor. Thus, for context sensitive grammars, the derivation process was even slower on a GPU than on a CPU.

Shader based derivation of split grammars has been proposed and investigated by Lacz et al. [2004], Magdics et al. [2009] and Marvie et al. [2012]. The method by Lacz et al. uses a render-to-texture loop and imposes the main workload of the algorithm on sorting intermediate symbols—similar to the overhead found in L-system generator by Lipp et al. The method Magdics et al. also requires several rendering passes. It tries to prevent divergence by using a different shader for each output symbol. In our evaluation, we incorporate an approach inspired by their work, efficiently grouping output symbols and launching individual kernels for each symbol type.

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

An approach focusing on parallelizing grammar derivation for procedural modeling of architecture has been published by Steinberger et al. [2014]. The *PGA* (Parallel Generation of Architecture) grammar is based on CGA [P. Müller et al., 2006] and uses a software scheduling GPU framework [Steinberger, Kainz, et al., 2012]. To avoid divergence, their approach groups shapes, which are to be processed by the same rule. Additionally, they draw parallelism from the rule itself. PGA compiles the entire rule set to achieve high performance rule derivation. Our work builds on PGA, focusing on its major drawbacks and analyzing different rule scheduling strategies.

2.4. GPU Task Scheduling

Ever since the CPU became capable of switching from one task to another at runtime, scheduling has become an important aspect of operating a computer [Tanenbaum, 2007]. Keeping the processing units busy to achieve peak performance is getting more complex with graphics processors and other stream co-processors being able to do massively parallel computation. Since the amount of parallelism that can be achieved with this kind of processors is much higher than on a general purpose CPU, conventional scheduling strategies can not simply be adopted, which is why GPU task scheduling is an active field of research.

When launching compute kernels with NVIDIA CUDA, a thread block scheduler chooses the threads to be executed on each multi-processor [NVIDIA, 2011]. As proposed by Fung et al. [2007], this could be improved by dynamic warp formation. Warps are units of 32 threads used by the hardware scheduler on an NVIDIA GPU.

Since the availability of CUDA 5.0 and hardware with compute capability 3.5, dynamic parallelism is supported [NVIDIA, 2012], which allows to launch new kernels from an already running one. As reported by Steinberger, Kenzel, and Schmalstieg [to appear] an implementation based on dynamic parallelism is slower than traditional CPU controlled kernels.

Launching compute kernels one after the other does not only suffer from overhead stemming from copying data back and forth between the CPU and the GPU, but also impacts performance negatively, when waiting on divergent threads to finish, until the next computation can be started. This is why Aila et al. [2009] propose a persistent thread execution model, where threads are kept running in a loop and are provided with new data to process without having to stop the running kernel and launch a new one. A good overview of persistent thread approaches is given by Gupta et al. [2012].

GPU computing was initially intended for one task at a time. To be able to process different problems at once, Hargreaves [2005] developed the ubershader, which calls different subroutines from a switch statement depending on the input. This approach has also been implemented in compute architectures like CUDA, where it is called a megakernel. The combination of the persistent thread and the megakernel technique is sometimes called persistent megakernel.

The behavior of a megakernel system is tremendously influenced by the thread granularity. Scheduling single threads can lead to thread divergence and does not allow to exploit GPU compute features like inter-thread communication via local shared memory. In some approaches [Parker et al., 2010; Tzeng et al., 2010], the tasks that can be scheduled have a minimal size of one warp, whereas others schedule whole blocks of threads [Chatterjee et al., 2011]. Previous work [Steinberger, Kainz, et al., 2012] even supports both. This research revealed that the performance suffers severely, if tasks of different thread sizes are scheduled simultaneously.

Sophisticated queuing of work items is essential for efficient software scheduling. Simultaneous access allowing quick transfer of data to and from the queues is necessary, as is load balancing for occupying all available worker threads. This has been implemented by Cederman et al. [2008], Chen et al. [2010] and Chatterjee et al. [2011]. Other techniques aiding performance include dynamic priorities [Steinberger, Kainz, et al., 2012] or message passing for synchronization [Luo et al., 2010; Stuart et al., 2009; Xiao et al., 2010].

3. Operators in Shape Grammars

To ease the process of writing rules, they are usually composed of *operators*. Operators can be seen as basic geometric transformations executed in sequence to form a rule. Our grammar supports the transform-only operators Translate, Rotate, and Scale and generative transformations that produce more shapes than existed before the transformation. Those operators are Repeat and Subdivide. Furthermore, we support two operations that change the dimension of a shape: the Extrude operator, applied to a quad, generates a box. The ComponentSplit operator, applied to a box, generates quads, representing the six faces of the box. To support stopping conditions in recursive rule definitions, we introduced an IfSizeLess operator, which stops rule derivation if an input shape has a size less than a specified value. Finally, we support the GenerateTerminal and the DiscardTerminal operator. The former calculates the geometry or simply copies the scaled model matrix (details on the implementation in chapter 7) With this set of operators, geometric descriptions of architecture can be articulated. For instance, the third rule in Figure 2.1(a), splitting shape T into five shapes, can be modeled as a combination of two Subdivide operators. All the operators mentioned above are described in more detail in sections 3.1.1, 3.1.2, 3.1.3, 3.1.4 and 3.1.5.

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

To avoid thread divergence, a scheduling system based on rule queuing can be set up to keep up the occupancy of a GPU [Steinberger, Kenzel, Kainz, J. Müller, et al., 2014]. The results of this rule scheduling paradigm are promising, as this system allows to generate whole cities in real time. However, the aforementioned work only focuses on a single strategy to schedule rules: The entire GPU is occupied with a persistent threads approach [Aila et al., 2009]. Symbols of equal type are collected in queues, while worker threads draw elements from these queues. All rules have to be available at compile time, requiring a full recompile when altering the rule set. In this work, we investigate alternative methods to schedule shape grammars on the GPU. We investigate the benefits and downsides of scheduling whole rules, which can consist of several operators in one rule, versus scheduling work for each operator individually, which from the scheduling system's point of view treats every operator as an individual rule. On the other hand, we investigate the trade-off between compiling entire rule sets and interpreting the provided rule set during runtime.

3.1. Operator Descriptions

The operators supported by our grammar are described in the following sections. Since C++ template parameters do not allow to use the floating point data type, parameters are specified as integers and multiplied by 0.001 when fed to the rule derivation engine. Table 3.1 shows a list of supported operators. The rotation parameters have to be specified in degrees. The axis parameter takes an integer between zero and two for the X, Y and Z axis.

3.1.1. Transform-Only Operators

- **Translate** takes three parameters for the x, y and z coordinate and the successive rule to place a shape on the desired spot in the scene. In the example in listing 3.1 below, the input shape is moved by 0.2 on the x axis, 0.35 on the y axis and 4.2 on the z axis.

Listing 3.1: Translate Operator

```
translate<200, 350, 4200, CallRule<Successor>>
```

Table 3.1.: Operator Overview

Operator	**Short Description**
$translate < X\_COORD, Y\_COORD, Z\_COORD, CallRule < Successor >>$	translate by a given amount
$rotate < X\_ROT, Y\_ROT, Z\_ROT, CallRule < Successor >>$	rotate using Euler angles
$scale < X\_SIZE, Y\_SIZE, Z\_SIZE, CallRule < Successor >>$	scale by a given amount
$repeat < AXIS, EXTENT, CallRule < Successor >>$	fills the area with as many copies of the shape as fit
$subdivide < AXIS, SubdivParam < EXTENT, CallRule < Successor1 >,$ $SubdivParam < EXTENT, CallRule < Successor2 >, \ldots >>>$	fills the area with shapes given by a recursive list of SubdivParams and calls an individual successor
$compsplit < CSP < CallRule < Bottom >, CSP < CallRule < Top >,$ $CSP < CallRule < Right >, CSP < CallRule < Back >,$ $CSP < CallRule < Front >>>>>>>>$	converts a shape into lower dimensional geometric primitives
$extrude < AXIS, EXTENT, CallRule < Successor >>$	extrude a shape into a higher geometric dimension
$ifSizeLess < AXIS, SIZE, CallRule < Successor1 >,$ $CallRule < Successor2 >>$	conditional execution
$GenerateTerminal$	insert a terminal shape
$DiscardTerminal$	do nothing

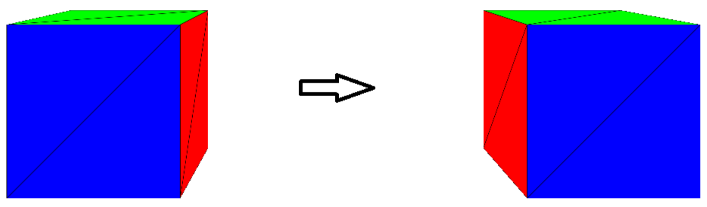


Figure 3.1.: *Translate* moves a shape to a position in the scene

- **Rotate** takes three parameters for the x, y and z coordinate and the successive rule to rotate a shape around the three axes. An example usage can be seen below, where the input shape is rotated 5 degrees on the x axis, 3.5 degrees on the y axis and 42 degrees on the z axis.

Listing 3.2: Rotate Operator

```
rotate<5000, 3500, 42000, CallRule<Successor>>
```

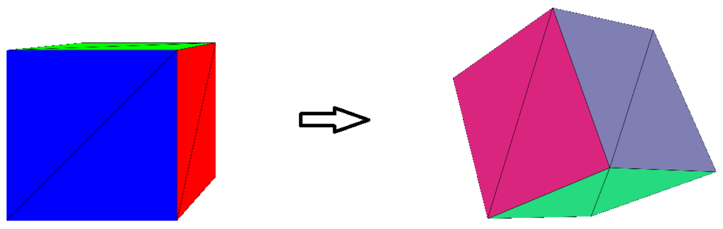


Figure 3.2.: *Rotate* turns a shape around the three axes

- **Scale** takes three parameters for the x, y and z coordinate and the successive rule to scale the input shape to the desired extents on the respective axis. In the example below, the input shape will have the extents 0.1, 7.36 and 4.9. The scale operator sets the size of the shape in an absolute fashion, since this information is stored in a vector of three floats during the derivation process.

Listing 3.3: Scale Operator

```
scale<100, 7360, 4900, CallRule<Successor>>
```

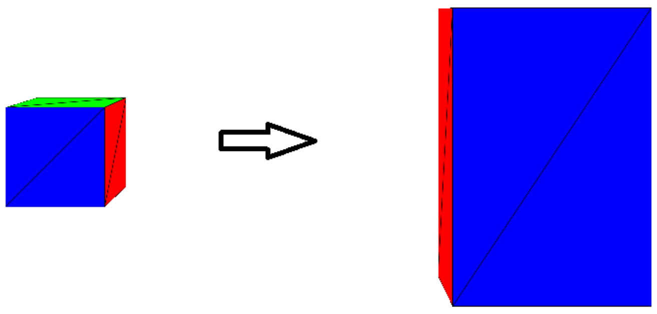


Figure 3.3.: *Scale* changes the size of a shape

3.1.2. Generative Operators

- **Repeat** takes two parameters, a successive symbol and a shape and produces as many new shapes with the width specified by the second parameter as fit into the original shape. The operator can work on either of the other two dimensions. An example usage of the repeat operator can be seen in listing 3.4.

Listing 3.4: Repeat Operator

```
repeat<X, 200, CallRule<Successor>>
```

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

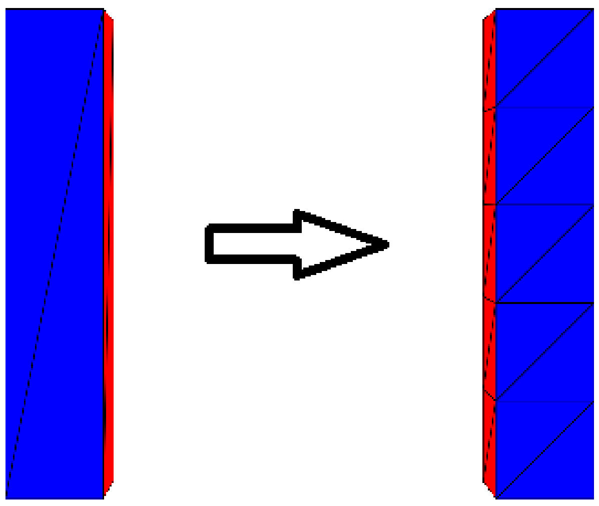


Figure 3.4.: *Repeat* generates new shapes within the old one as many times as they fit in

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

Listing 3.5: Subdivide Operator

```
subdivide<Y,
          SubdivParam<500, CallRule<Successor1>,
          SubdivParam<500, CallRule<Successor2>>>
```

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

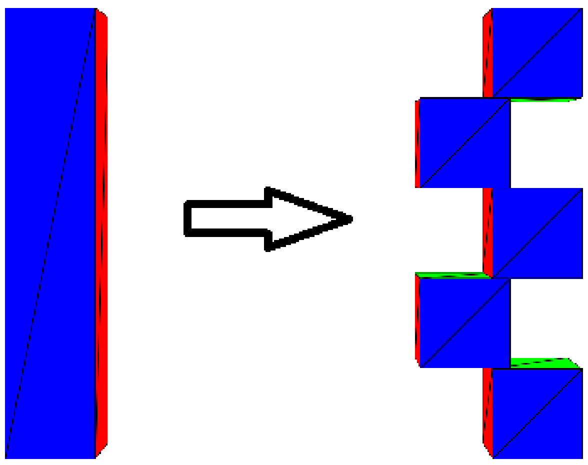


Figure 3.5.: *Subdiv* replaces a shape with smaller instances

3.1.3. Dimension-Change Operators

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

Listing 3.6: Component Split Operator

```
compsplit<CSP<CallRule<Bottom>,
          CSP<CallRule<Top>,
          CSP<CallRule<Right>,
          CSP<CallRule<Left>,
          CSP<CallRule<Back>,
          CSP<CallRule<Front>>>>>>>>
```

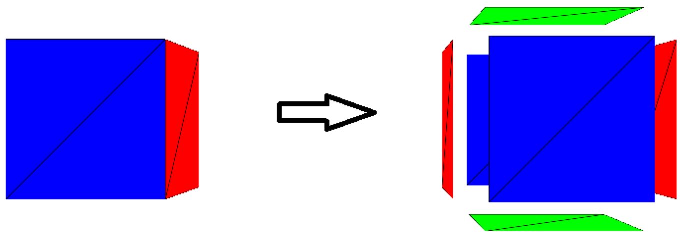


Figure 3.6.: *ComponentSplit* splits a cube into its six faces

- **Extrude** needs an axis and a distance as parameters and converts a quad into a cube by setting the extent of the shape to the specified value at the requested axis.

Listing 3.7: Extrude Operator

```
extrude<Z, 5000, CallRule<Successor>>
```

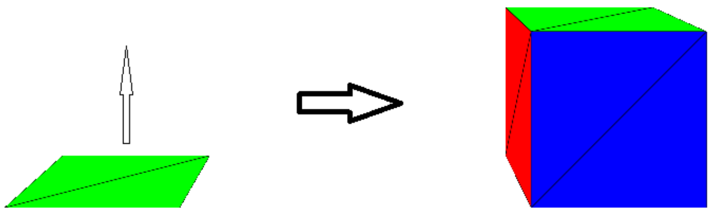


Figure 3.7.: *Extrude* adds height to a two dimensional shape

3.1.4. Conditional Operators

- **IfSizeLess** takes an axis, a distance and two successive symbols as parameters. If the size of the processed shape at the given axis is less than the value specified, the third parameter is taken as the next symbol. The fourth parameter will be the successive symbol if the condition evaluates to false. In the example below, the input shape will be generated if its size on the x-axis is less than 2.0.

Listing 3.8: Conditional Size Operator

```
ifSizeLess<X, 2000, GenerateTerminal, DiscardTerminal>
```

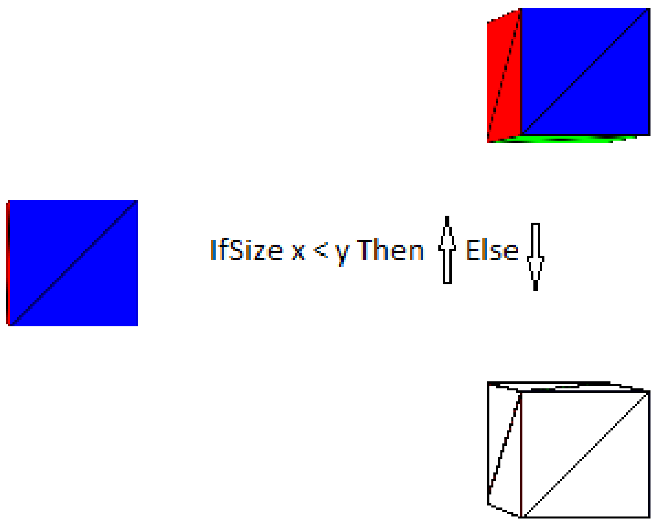


Figure 3.8.: *IfSizeLess* executes the next operator only if the condition evaluates to true

3.1.5. Terminal Operators

- **GenerateTerminal** This operator does not take any parameters. It will call a method appropriate for the input shape that will be fed to it. Depending on the chosen rendering method, GenerateTerminal will perform one of the following computations: Calculate vertices, normals and indices when using non-instanced rendering or simply apply a scale transformation according to the size parameter of the shape. A counter is incremented using the atomicAdd CUDA operation to get an index into the buffer on the graphics card, where the data will be stored.

- **DiscardTerminal** does not need any parameters either. This operator simply returns from the execution method right away, thus discarding the input shape.

4. GPU Task Scheduling

4.1. Iterative Production

The most straight forward way to tackle shape grammar evaluation is starting a single thread for each symbol in the currently active set. This is similar to the approach by Lipp et al., 2010. As mentioned before, this method has the drawback of extensive thread divergence. Inspired by the approach by Laine et al., 2013, we avoid thread divergence, providing individual queues for each rule. Before running the rule evaluation, we allocate a queue for each rule on the GPU. We then insert the axioms into the per-rule queues. After querying the queue fill rates, we start an individual kernel for each queue, launching just as many threads as there are elements in each queue. Each thread fetches one element from the queue and executes the rule associated with it. During rule evaluation, new shapes are generated, which are again inserted into the respective queues. Terminal shapes are placed into a separate set of arrays, for which no rule evaluation is taking place. These arrays are later used for rendering. After all kernel

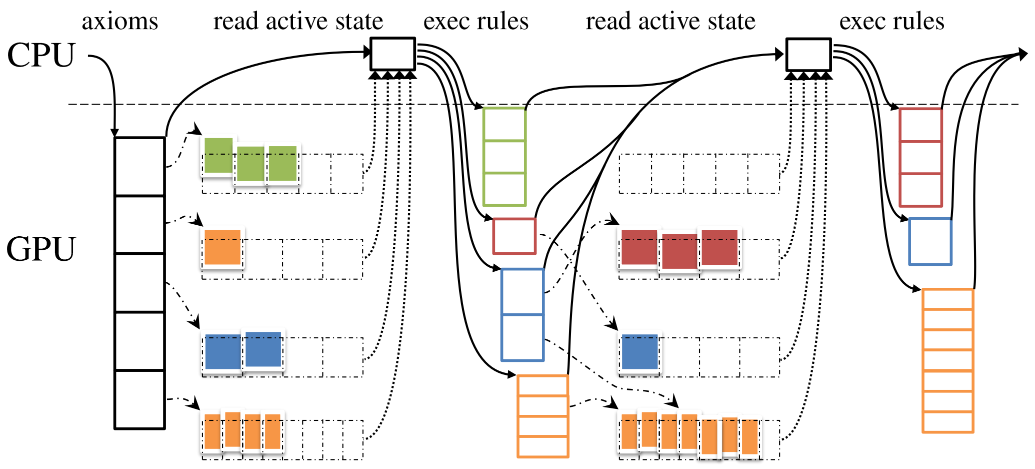


Figure 4.1.: Using an individual queue for each rule, we provide an iterative shape rewriting algorithm, which does not suffer from divergence. At first all axioms are being placed in the queues. Then, we read the queue fill rate back to the CPU before launching just enough threads to process all queued shapes. We continue this process, until there are only terminal shapes left.

launches are completed, we read the queue fill rates from the GPU and again launch kernels to evaluate rules for all shapes currently being held by all queues. We continue this process until all non-terminal shapes have been processed and all queues reach an empty state. Shapes currently being queued represent the current active set. This process is visualized in Figure 4.1.

Using this approach, all threads within one kernel evaluate the same rule, executing the same set of instructions. Thus, no thread divergence occurs and execution is efficient on the GPU hardware. While this approach is set up easily, deriving a whole rule set requires many kernel launches.

Additionally, the queue fill rates need to be read back from the GPU before a new set of kernels can be launched. This step cannot be avoided, as the number of threads to be launched needs to be known.

4.2. Persistent Megakernel Production

As an alternative way to tackle shape grammar evaluation on the GPU, we use a persistent threads approach as proposed by Aila et al. [2009]. We again use a single queue per rule. But instead of launching a new kernel for every rule production, we run threads in an a loop. In every loop iteration, each thread draws a shape from one of the queues and executes its associated rule. If new shapes are being generated, we add them back into the respective queues. As shapes are being drawn from and inserted into the queues concurrently, we use a flag per queue element to avoid errors due to read-before-write dependencies [Steinberger, Kainz, et al., 2012]. All threads continue in their loop, until all queues are empty and no thread is still evaluating a rule. To avoid thread divergence, we force all threads within a block to draw shapes from the same queue in every iteration. This setup is outlined in Figure 4.2.

A persistent megakernel setup avoids synchronization with the CPU and does not have any kernel launch overhead. On the downside, all rules must be compiled into the same kernel. As kernels are optimized as a whole, the characteristics of the most resource-hungry rule determines the efficiency

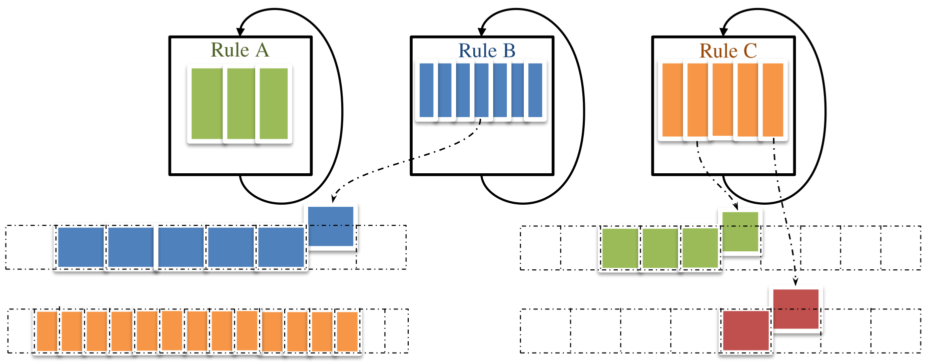


Figure 4.2.: As alternative rule derivation algorithm, we use a persistent megakernel setup. Worker blocks are running in a loop. At the beginning of each loop iteration, they draw a new set of shapes from one of the queues and evaluate the associated rules, before inserting the generated shapes back into the queues. The kernel is kept alive until all non-terminal shapes have been processed.

of all others. Furthermore, the persistent setup requires a more complex queuing strategy to avoid errors due to read-before-write dependencies. As our grammar does not introduce any priority between rules, shapes can be drawn from any queue at the beginning of each iteration. To avoid idle threads, we circle through all queues in a round-robin fashion and only draw shapes from a queue, if there are enough items in the queue to provide all threads in the block with work.

5. Rule Derivation

To aid the description of our rule scheduling algorithm, we define the following scheduling entitites:

Definition 5.0.1 *A* **procedure** *is an entity being scheduled by the rule derivation engine. It has several properties describing its behavior, for example, an ID and the number of threads that simultaneously execute the procedure. When a procedure is scheduled, a method is called to run the code defined within the procedure.*

Definition 5.0.2 *A* **Shape** *is a data structure for capturing the geometric description of a basic shape. For a box, the information used to describe it geometrically is the following:*

- *The* **size** *– extents on the x, y and z axis*
- *The* **model matrix** *transforming the shape from its own coordinate system to world coordinates in the final scene*

In addition to the geometric information, we store the following data to be able to distinguish between different shapes and associated rules:

- *A* **type ID** *defining what kind of shape will be generated (for example, a box, a quad, a sphere, etc).*
- *Additionally, in the case of interpreted rule derivation, a* **symbol ID** *is stored to identify which shape belongs to which rule. This is not necessary when using the precompiled method.*

5.1. Precompiled Rules

The goal of the precompiled rule set approach is to leave as many decisions as possible to the compiler. For this approach, we require the complete rule set to be specified beforehand. This includes all rules, their parameters, and outputs. The only information not required in advance are the axioms. All computations and branch decisions that are not input dependent need only be executed once, so we perform them at compile time. Thus, at runtime, all operators can be executed without any additional information. No lookups to rule tables or symbol translations are needed.

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

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

5.2. Interpreted Rules

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

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

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

6. Rendering Methods

Rendering is done in three different ways using OpenGL. Those variants are instanced and non-instanced rendering and a hybrid form we call software instancing. More details on the implementation are given in section 7.2 and a performance comparison can be seen in chapter 8.

6.1. Non-Instanced Rendering

The non-instanced rendering method is the conventional way of rendering geometry. At the end of the rule derivation process, the GenerateTerminal operator calculates the vertices, normals and indices which will be rendered by the draw method of the memory manager. When a lot of new geometry is produced every frame, this approach is not the best way to go because a lot of data has to be moved down the graphics pipeline. As will be show in the results chapter, if the scene gets complex, tenth of millions of vertices will be produced.

6.2. OpenGL Instanced Rendering

Instanced rendering uses a feature of OpenGL to render instances of basic shapes several times with one draw call. This has three advantages: First, during terminal evaluation, less data has to be written to slow global memory, as only the matrices need to be copied. Second, less storage is required between generation and rendering. And third, during rendering, less data needs to be read, saving memory bandwidth. However, even though this method is hardware accelerated, the number of vertices of the basic shapes is rather low. According to the hardware vendors, the overhead associated with rendering small objects using instancing is higher than just rendering non-instanced geometry. Thus, rendering is actually slower using instancing.

6.3. Software Instanced Rendering

Software instancing is a hybrid form of rendering, which was implemented to take advantage of both previously mentioned techniques. First, the geometry for all terminal shapes is generated once and stored in a buffer on the graphics card. Then during rule derivation, only the transformation matrices have to be calculated and stored in memory—as in the case of instanced rendering.

The rendering speed should not suffer from overhead, since we are using the same rendering technique like in the non-instanced version, whereas the generation time of the geometry should be about the same as in OpenGL instancing.

The negative aspect of software instanced rendering is it's memory consumption, which is as high as both previously described techniques together.

7. Implementation

Our implementation is written in CUDA and C++ and makes heavy use of templates. It consists of four major components. A rendering framework, a memory manager handling the allocation of memory on the graphics card and coordinating the flow of control between geometry generation and drawing, the rule derivation engine and a test case definition.

The rendering framework is responsible for setting up a window and an OpenGL rendering context. The Renderer class is provided with a camera class that takes parameters to set up a viewing frustum. Furthermore, an input handler class needs to be specified to process mouse and keyboard events. After everything is set up, the render loop method can be started. For every frame, the events generated in the window are passed to the registered input handler, which takes an appropriate action, if an event occurs that is defined to be worth of an action. For example, a mouse drag after a ”left mouse button down” event will trigger an action to move the camera. After the state of the renderer has been updated according to the processed events, the render method is responsible to call all methods required to update the screen content. After activating one of the shaders, the memory manager either initiates the (re)generation of the geometry by starting the rule derivation engine and then calls the draw method or draws the previously created geometry right away.

The memory manager takes care of the graphics buffers for the geometry and OpenGL draw calls. There are two classes named MeshInstanced and MeshNonInstanced. Only one of them is compiled into the executable according to the test case definition. The classes are responsible for allocating the required graphics memory for the generated geometry. Depending on the rendering method, either memory for vertices, normals and indices is allocated, or, in the case of one of the instanced rendering methods, space for the root shapes and transformation matrices is requested from the OpenGL driver. Since no interactive rule editing is supported by our implementation[1], the rule set is processed and loaded into the memory of the graphic card once if the interpretation method is chosen. We call this process *dispatch table initialization*. In the precompiled version, this is not necessary, as the rule set is already incorporated in the binary by the compiler. When everything is set up (this happens only once when the program is started), the memory is mapped to CUDA and the derivation engine is kicked off. Since these classes hold the reference to the geometry data, they implement the draw method.

[1]A rule editor is left for future work, as it is not necessary for the investigation of derivation strategies

Geometry is either rendered using instancing (glDrawElementsInstanced) or conventional indexed rendering (glDrawElements).

The derivation engine is the component, which executes all rules and where the final geometry calculation takes place. It is written entirely in CUDA. According to the test case definition, queues are set up and the chosen technique for either separate kernel launches or a megakernel is executed. For the interpretation of rules, all operators are implemented as procedures, which are a generic interface to the scheduling mechanism. This is why we also call this *procedure-per-operator* method. When using a precompiled rule set, a procedure is scheduled for every rule making this the *procedure-per-rule* method. Several operators that make up a rule, are executed sequentially by one thread. After the production is launched, an init procedure is executed, which the axiom shapes are fed to. In the procedure-per-rule method, new rules are enqueued via a subroutine called CallRule at the end of every rule if the derivation process does not hit a terminal symbol. This CallRule subroutine is necessary to form chains of operators in one rule. It provides the same C++ class interface like an operator, but instead of altering a shape's properties, it passes it on to the next rule and enqueues this rule, to let the scheduler know that it is ready for execution. When the rules should be interpreted, a new procedure is scheduled after every operator. This is done via a dispatcher switch, which looks up the successive symbol information, which is stored with the rule information in the dispatch table, by using the current shape's symbol id as an index into that table. More details on the rule derivation engine are presented in section 7.1 of this chapter on rule derivation.

The test case definition consists of three parts. First, the production rules have to be defined in a separate header file (see appendix A). From those rules, the precompiled code or the dispatch table for interpretation is generated. Second, an appropriate initial procedure has to be defined. This init procedure takes care of feeding the axioms to the start rules, from where the shapes get passed on from operator to operator until they hit a terminal. A code sample for initialization is listed the appendix (Appendix B.2, B.3). The third part of the test case definition consists of parameters controlling which derivation method to use, which scheduling technique and which test case to run. This is simply a header file containing preprocessor switches and variable definitions.

7.1. Rule Derivation

The rule derivation engine is the main component of our implementation. Once everything is set up by the memory manager, the chosen kernel technique is launched and the init procedure (see B.2, B.3) is called. This procedure can be executed by multiple threads simultaneously to allow the construction of separate buildings in parallel. Each of the instances spawns an axiom shape directly on the GPU, so no extra data has to be copied.

As defined at the beginning of chapter 5, the procedure is the interface to the scheduling system. For the precompiled version, there is one procedure class (listing 7.1), whereas for the interpreted method, every operator is a procedure (listing 7.2). The former calls the out() method of the operator passed as template parameter, which subsequently calls the operator subroutine and the latter calls the operator code in the execute() method of the procedure.

Listing 7.1: Rule procedure used in precompiled mode

```
template <typename Shape, typename Operator>
class RuleT : public ::Procedure
{
public:
  typedef Operator Head;
  typedef Shape ExpectedData;
  static const int NumThreads = 1;
  static const bool ItemInput = true;
  static const int symbol_count = Operator::symbol_count;

  template <int symbol_offset, typename RuleSet>
  __host__ __device__ static void initDispatchTable(Rule* tbl)
  {
    Operator::template initDispatchTable<symbol_offset, Shape, RuleSet>(tbl);
  }

  template <int symbol_offset, typename RuleSet>
  __host__ __device__ static int getSymbol()
  {
    return Operator::template getSymbol<symbol_offset, RuleSet>();
  }

  template <class Q, class Sync>
  static __device__ __inline__ void
  execute(int threadId, int numThreads, Q* queue, ExpectedData*
      data, uint* shared)
  {
    uint out_param = 0;
    Operator::out(*data, queue, out_param);
  }

  static std::string name() { return typeid(RuleT).name(); }
};
```

When interpreting code, the parameters for the operation to be executed have to be fetched from the dispatch table first. This is done in the execute() method of the procedure before calling the subroutine that does the actual transformation. In the precompiled version, the parameters do not have to be fetched, as they are already known at compile time and therefore put in their place in advance.

Listing 7.2: Operator procedure used in interpreted mode

```
template <int x, int y, int z, class Next>
class translate
{
public:
  typedef Next Tail;
  static const int symbol_count = Next::symbol_count + 1;

  template <class Shape, typename Q>
  __device__ __inline__ static void out(Shape& shape, Q* q, int unused)
  {
    typename translateOperator<Shape, Next, int>::InputParameters p;
    p.x = x * 0.001f;
    p.y = y * 0.001f;
    p.z = z * 0.001f;
    translateOperator<Shape, Next, int>::execute(p, shape, q, unused);
  }

  template <int symbol_offset, typename Shape, typename RuleSet>
  __host__ __device__ static void initDispatchTable(Rule* tbl);

  template <int symbol_offset, typename RuleSet>
  __host__ __device__ static int getSymbol()
  {
    return symbol_offset + symbol_count - 1;
  }
};
```

To implement a shape, we store its type, size and the model matrix (and the symbol ID in the interpreted method). All operators, except the terminal operators, only alter these attributes, which is all the information needed to produce the geometry data. Using the non-instanced rendering method, GenerateTerminal calculates, according to the type of the shape, the vertex attributes and stores them in an OpenGL buffer, which is mapped to CUDA before the generation process starts. If instanced rendering is used, all that is left to do for the GenerateTerminal operator, is to store the model matrix in the OpenGL buffer. See appendix B.1 for a code sample of the GenerateTerminal operator.

The precompiled method is implemented using the template meta-programming paradigm. All rule sequence decisions are made by the compiler according to the rule definitions, which creates instances of rules and operator chains at compile time. The rule definitions are written in C++ template code. In the example as shown in listing 7.3, the input shape is rotated and then split into four smaller boxes. One of them will be discarded by the IfSizeLess operator, one will be generated in its position after the split and two will be moved a little. The result can be viewed in figure 7.1. Further rule sets used for generating the results presented in chapter 8 are listed in appendix A.

Listing 7.3: Sample Rule Set

```
struct RuleB : RuleT<Box, IfSizeLess<X, 200,
                   DiscardTerminal, GenerateTerminal> > {};
struct RuleA : RuleT<Box, translate<0, 567, 0, GenerateTerminal > > {};

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

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

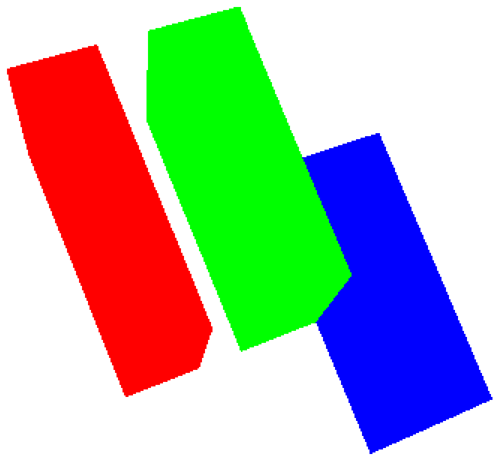


Figure 7.1.: *Simple rule set* splitting a box into four smaller ones and discarding one

We use the template code not only to generate the operator chains for the precompiled method, but also to fill the dispatch tables for the interpreted case. However, the compile process in the interpreted case does not involve the generation of GPU code, only the CPU code generating the dispatch table. Thus, a full runtime adjustment of the rule set could easily be achieved using a custom parser. With an interactive editor incorporating this parsers, a rule set could be edited and a dispatch table uploaded to the GPU at run time.

An array of operators in the dispatch table resulting from the rule set above (7.3) would look like in listing 7.5. This array, consisting of rule entries (7.4), is loaded into CUDA memory and used in the dispatch switch (appendix B.4), which is called every time before a new operator is enqueued. This method gets the next entry in the dispatch table by using the symbol ID, stored in the shape that is passed as a parameter, as an index. The symbol ID gets adjusted by every operator after its transformation is done, right before the shape is handed over to the dispatcher switch. After determining the operator code of the rule in next entry, the switch statement enqueues the appropriate operator and passes on the shape data structure. In case an operator does not have a successor, for example, in case of the GenerateTerminal, DiscardTerminal, or IfSizeLess operator, the successor ID is set to -1, as shown in listing 7.5.

Listing 7.4: Dispatch Table Entry

```
struct Rule
{
  unsigned char operatorCode;
```

```
    unsigned int numberOfParameters;
    float parameters[MAX_NUM_PARAMETERS];
    unsigned int predecessor;
    unsigned int numberOfSuccessors;
    int successors[MAX_NUM_SUCCESSORS];

};
```

Listing 7.5: Dispatch Table Example

```
d_rules[0].opCode: 13
  d_rules[0].successor[0]: -3
d_rules[1].opCode: 7
  d_rules[1].successor[0]: -2
d_rules[2].opCode: 9
  d_rules[2].successor[0]: -1
d_rules[3].opCode: 1
  d_rules[3].successor[0]: 1
d_rules[4].opCode: 9
  d_rules[4].successor[0]: -1
d_rules[5].opCode: 19
  d_rules[5].successor[0]: 4
d_rules[6].opCode: 3
  d_rules[6].successor[0]: 3
d_rules[7].opCode: 17
  d_rules[7].successor[0]: 6
```

The precompiled code does not use a dispatch table. This is why there is the

CallRule class at the end of each rule. All operators in a rule get executed by a thread, until it hits a CallRule, where the shape is fed to the next rule and enqueued for scheduling. This class imitates the behavior of an operator by doing the enqueuing instead of transformation calculations in the out() method. A code snipped illustrating this mechanism is shown in listing 7.6.

Listing 7.6: CallRule class

```
template <class RuleT>
class CallRule
{
public:
  static const int symbol_count = 0;
  template <typename Shape, typename Q>
  __device__ __inline__ static void out(Shape& s, Q* q, int unused)
  {
    q->template enqueue<RuleT>(s);
  }
};
```

7.2. Rendering Methods

7.2.1. Non-Instanced Rendering

Non-instanced rendering is the traditional rendering method drawing generated vertex data with the glDrawElements OpenGL call. In the initialization phase buffers for vertices, normals and indices are allocated. Before each run of the rule dervation engine, these memory regions are mapped to CUDA. The GenerateTerminal operator calculates the vertex, normal and index data and stores it in the supplied buffers (a code snippet is supplied in appendix B.1). After the derivation is completed, the memory segments have to be unmapped from CUDA before glDrawElements can be used.

7.2.2. OpenGL Instanced Rendering

We call this method also hardware instanced memory, because the OpenGL calls can be accelerated by the graphics processor.

In the initialization, the rule derivation engine is invoked once for every shape type (box, quad, etc) that has to be rendered in non-instanced mode. Small buffers are provided to store vertex, normal and index data for one instance of every shape.

The rule derivation engine is supplied with buffers for the transformation matrices, which have to be mapped and unmapped from CUDA for every run of the generation process. The GenerateTerminal operator only applies the scale transformation and stores the matrix. Very little computation and memory accesses are done, making this method faster in terms of geometry generation time.

Once generation is finished, the generated shapes of the initialization, we call them root shapes, are drawn by a call of the glDrawElementsInstanced OpenGL funtion. This functionis called only once for every shape type with a parameter how many instances should be drawn. The index into the matrix buffer is supplied via OpenGL vertex attribuges.

7.2.3. Software Instanced Rendering

Like in the other instancing method, the non-instanced geometry generation is invoked once in the initialization phase. Instead of generating only one instance per shape type, as many boxes, quads, etc, are generated as there will be transformation matrices produced by the GenerateTerminal operator. Since we are not supporting stochasticity, the numbers are known a priori.

The rule derivation process is the same like in OpenGL instancing. Only the transformation matrices are calculated and stored. The time it takes for deriving a set of rules should therefore be the same.

The transformation matrices are not supplied to the shader as vertex attributes, like in the hardware accelerated version, but in a shader storage buffer object—SSBO. During shader execution, the instance ID of the current shape is calculated by dividing the vertex ID by the number of vertices of the shape to be rendered. This instance ID is used to fetch the matrix from the SSBO in the vertex shader.

Each of the three implemented techniques has its advantages and disadvantages. OpenGL and software instancing benefit from optimized vertex rendering hardware, which results in higher frame rates, if little data changes from one frame to the next. The OpenGL and the software instancing benefit from the reduced memory bandwidth cost, but both have their down sides. OpenGL suffers from overhead involved in the instancing and software instancing has very high initial memory requirements. Another effect has been observed while testing, which may be taken into account positively. The OpenGL instancing, even though slower than the conventional vertex rendering, does not fluctuate in the frame rate, depending on the content of the visible scene. In non-instanced rendering, frame rates seem to depend on the amount of content that can be culled. When only little of the scene is visible on screen, frame rates are clearly higher. However, the frame rates when using glDrawElementsInstanced stay the same, which indicates that culling does not work as efficiently using instanced rendering. It is not known if this effect may be an implementation issue of the used graphics hardware or not.

8. Results and Evaluation

8.1. Testing Environment

The results of the tests presented in this section were carried out on a system with an Intel Core i7-4771 CPU at 3.5 GHz, 16 GB of main memory and a NVIDIA Geforce GTX TITAN graphics card with 6 GB VRAM.

As this thesis focuses on the rule derivation process, only rudimentary shaders have been applied to visualize the produced geometry. The absence of visually appealing rendering, as well as other features not relevant for this thesis, like optimizing the number of objects that need to be generated or ignoring geometry that need not be regenerated for every frame, is the topic of future work.

We compare twelve different configurations, which are variations of interpretation and precompilation, instanced, non-instanced and software instanced rendering, as well as iterative rewriting and persistent megakernel production. We applied these variations to five different rule sets, which are described in more detail in their respective sections of this chapter: *Houses* 8.2.1, *Detailed Houses* 8.2.2, *Many Rules Houses* 8.2.3, *Single Menger-Sierpinski* 8.2.4, *Multi Sierpinski* 8.2.5.

8.2. Test Cases

Table 8.1 offers an overview of the properties of the five rule sets, which were used to conduct the tests for the procedural modeling approaches we introduced in previous chapters. Further details, test results and example views are presented in the following subsections.

Table 8.1.: Test Scene Statistics

	Houses	D. Houses	M. R. Houses	Single M.S.	Multi M.S.
Rules	9	25	90	5	15
Operators	11	30	110	4	12
Terminals	748800	3.03 M	71500	1.92 M	1.52 M
Vertices	17.97 M	72.65 M	1.72 M	46.1 M	27.65 M
Indices	26.96 M	108.98 M	2.57 M	69.1 M	41.47 M

The statistics that were sampled to measure the performance are as follows:

- **The generation time t** in milliseconds, which is the time that the rule derivation process needed to calculate the geometry and fill the buffers.

- **The frames per second (fps)** gives an overview of the rendering performance
- **mspf** is the reciprocal value of fps and shows the milliseconds it took to render one frame
- **DRAM load** shows how many times a memory access was made during the derivation (given in millions)
- **DRAM store** displays the write operations in millions that were done by the rule derivation engine

8.2.1. Houses Test Case

The houses rule set A.1 is rather simple. It generates the floors of a house and adds a variable number of windows in each floor wall. Depending on the initial dimensions of the axioms, as many windows as fit in the width and depth are produced. We are repeating this step as many times as a floor fits into the height of the axioms. The top of each floor is either used as ceiling or roof. To produce a heavy workload, we generate a grid of 48 x 48 instances of such a simple house by feeding this number of axioms to the pipeline. The production process is started in parallel by starting a thread with the init procedure for each axiom. Table 8.2 shows the average performance statistics, while detailed numbers of each applied technique can be found in table 8.3.

Table 8.2.: Houses Average Results

Avg t	Avg fps	Avg mspf	Avg load	Avg store
19.04	153.29	7.93	3.60	31.65

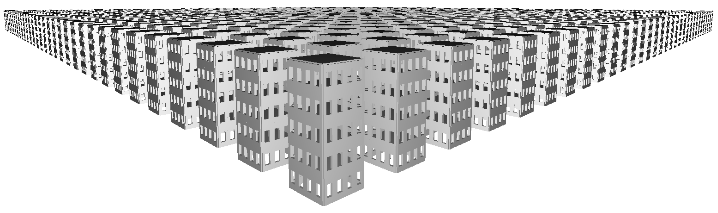


Figure 8.1.: The *Houses* testcase shows a scene of 48×48 simple house models.

Table 8.3.: Houses Test Results

[a]			t	fps	mspf	DRAM load	DRAM store	Warp Efficiency
IRD	KLS	NIR	35.85	**194.88**	**5.13**	4.38	58.10	0.98
		HIR	8.61	74.70	13.39	3.38	13.08	0.97
		SIR	9.47	192.04	5.21	3.41	13.10	0.97
	PMK	NIR	43.45	191.60	5.22	**6.78**	**73.03**	0.98
		HIR	8.23	74.59	13.41	5.86	20.28	0.97
		SIR	8.28	192.68	5.19	5.82	20.28	0.97
PRD	KLS	NIR	**47.28**	193.78	5.16	3.15	72.72	0.98
		HIR	5.42	**74.16**	**13.49**	1.87	8.67	0.97
		SIR	5.33	193.56	5.17	1.87	**8.61**	0.97
	PMK	NIR	46.77	191.47	5.22	3.10	71.88	0.99
		HIR	**4.87**	74.74	13.38	**1.82**	10.01	0.98
		SIR	4.89	191.34	5.23	**1.82**	10.02	0.98

[a] **IRD** …Interpreted Rule Derivation **PRD** …Precompiled Rule Derivation **KLS** …Kernel Launch Scheduling **PMK** …Persistent Megakernel Scheduling
NIR …Non-Instanced Rendering **HIR** …Hardware Instanced Rendering **SIR** …Software Instanced Rendering
t …geometry generation time (in milliseconds) **fps** …frames per second **mspf** …milliseconds per frame **load/store** …memory operations (in millions)

8.2.2. Detailed Houses Test Case

The Detailed House rule set should reflect the modeling power of shape grammars as well as constitute a test with a medium amount of different rules. In a real world scenario, house models would be much more detailed in terms of geometric detail and the use of textures. The Subdivide operator (listing 3.5) makes up most of the rules (appendix A.2), dividing the axiom into floors, doors, windows and walls. To put a heavy load on the graphics card for testing, an 80 x 80 grid of houses is generated. Table 8.4 shows the average performance statistics, while detailed numbers of each applied technique can be viewed in table 8.5.

Table 8.4.: Detailed Houses Average Results

Avg t	Avg fps	Avg mspf	Avg load	Avg store
78.35	38.18	31.87	16.38	131.94

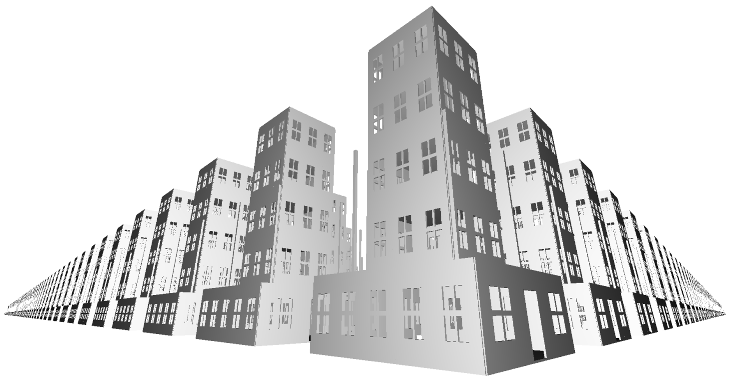


Figure 8.2.: The *Detailed Houses* dummy picture

Table 8.5.: Detailed Houses Test Results [a]

			t	fps	mspf	DRAM load	DRAM store	Warp Efficiency
IRD	KLS	NIR	145.95	47.58	21.02	18.74	239.10	0.98
		HIR	37.47	18.77	53.28	15.72	55.54	0.97
		SIR	37.40	48.85	20.47	15.72	55.55	0.97
	PMK	NIR	165.08	47.16	21.20	**28.76**	281.78	0.98
		HIR	37.05	18.70	53.49	26.47	89.27	0.96
		SIR	36.96	48.73	20.52	26.42	89.26	0.96
PRD	KLS	NIR	192.98	46.77	21.38	12.82	295.98	0.98
		HIR	**21.39**	18.44	54.22	**8.52**	**38.09**	0.97
		SIR	22.03	**49.07**	**20.38**	8.67	38.16	0.97
	PMK	NIR	**196.19**	46.72	21.40	14.63	**301.32**	0.98
		HIR	23.87	**18.31**	**54.63**	10.06	49.63	0.97
		SIR	23.85	49.02	20.40	10.08	49.59	0.97

[a] **IRD** . . . Interpreted Rule Derivation **PRD** . . . Precompiled Rule Derivation **KLS** . . . Kernel Launch Scheduling **PMK** . . . Persistent Megakernel Scheduling
NIR . . . Non-Instanced Rendering **HIR** . . . Hardware Instanced Rendering **SIR** . . . Software Instanced Rendering
t . . . geometry generation time (in milliseconds) **fps** . . . frames per second **mspf** . . . milliseconds per frame **load/store** . . . memory operations (in millions)

8.2.3. Many Rules Houses Test Case

The purpose of the Many Rules Houses test case (A.3) is to investigate the qualities of the scheduling approaches implemented in this work. The rule set of the simple house used in the Houses test (8.2.1) is replicated ten times to generate a total of 90 different rules. Those have to be treated by the scheduler as different entities which, if precompiled rule derivation is used, can not be merged for parallel execution as the scheduler views them as completely unrelated rules. To identify the ten different rule sets, each house is produced with a height descending from ten floors to one floor as can be seen in figure 8.3. Furthermore, to put some workload on the graphics card, those ten rule sets are utilized on 200 axioms in parallel. The complexity of this rule set is also reflected in the compile time, which can take up to 16 minutes on the used machine (8.1). Table 8.6 shows the average performance statistics, while detailed numbers of each applied technique can be viewed in table 8.7.

Table 8.6.: Many Rules Houses Average Results

Avg t	Avg fps	Avg mspf	Avg load	Avg store
3.20	1452.46	0.81	0.56	3.83

Table 8.7.: Many Rules Houses Test Results

			t	fps	mspf	DRAM load	DRAM store	Warp Efficiency
IRD	KLS	NIR	5.27	**1897.17**	**0.53**	**0.40**	6.48	0.98
		HIR	2.49	757.20	1.32	0.46	**1.46**	0.98
		SIR	2.48	1775.00	0.56	0.45	1.49	0.98
	PMK	NIR	4.45	1826.83	0.55	**0.97**	7.39	0.99
		HIR	1.12	763.89	1.31	0.67	2.34	0.99
		SIR	**1.07**	1772.75	0.56	0.68	2.35	0.99
PRD	KLS	NIR	**7.07**	1787.33	0.56	0.64	6.39	0.99
		HIR	3.47	**754.15**	**1.33**	0.42	2.39	0.99
		SIR	3.59	1774.75	0.56	0.42	2.44	0.99
	PMK	NIR	4.24	1783.67	0.56	0.71	**7.42**	0.99
		HIR	1.62	769.71	1.30	0.46	2.90	0.99
		SIR	1.60	1767.08	0.57	0.49	2.93	0.99

[a]

[a] **IRD** . . . Interpreted Rule Derivation **PRD** . . . Precompiled Rule Derivation **KLS** . . . Kernel Launch Scheduling **PMK** . . . Persistent Megakernel Scheduling
NIR . . . Non-Instanced Rendering **HIR** . . . Hardware Instanced Rendering **SIR** . . . Software Instanced Rendering
t . . . geometry generation time (in milliseconds) **fps** . . . frames per second **mspf** . . . milliseconds per frame **load/store** . . . memory operations (in millions)

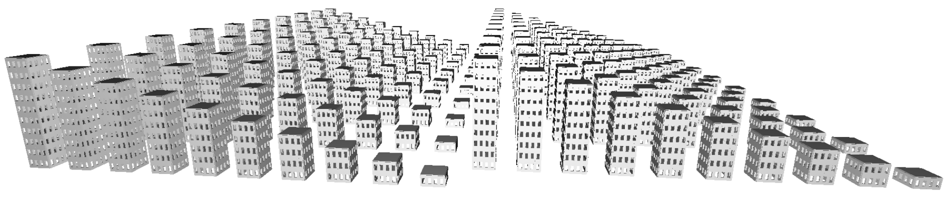


Figure 8.3.: *Many Rules Houses* showing 20 derivations of 10 different rule sets.

8.2.4. Single Menger-Sierpinski Test Case

This test scene shows a Menger-Sierpinski's cube with five levels of recursion. The rule set defines the steps to construct one level of the cube, resulting in twenty boxes. At the end of each stage, the IfSizeLess operator checks the side length of its input shape. If it is less than the specified value, the rule derivation continues with last recursion stage. If the boxes are still big enough, a proxy rule directs the path of execution back to the start of the rule set. (see appendix A.4 for a complete code listing). This proxy rule is needed because we use C++ template code for rule definition which has to be defined in reversed order, starting with the last rule. The compiler can only generate code from what it has already processed. So to jump back from the end of the rule set, it needs a known piece of code to create an instance from. Table 8.8 shows the average performance statistics, while detailed numbers of each applied technique can be viewed in table 8.9.

Table 8.8.: Single Menger-Sierpinski Average Results

Avg t	Avg fps	Avg mspf	Avg load	Avg store
44.91	58.62	20.79	6.39	73.21

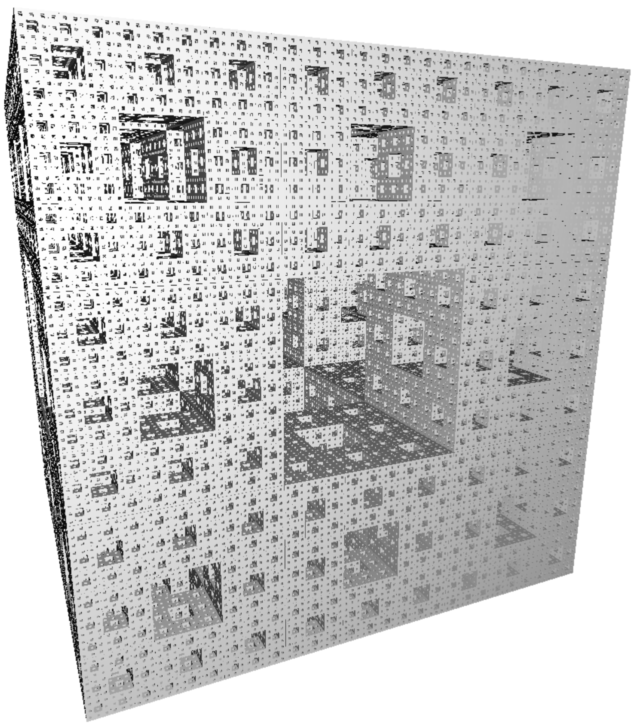


Figure 8.4.: *Single Menger-Sierpinski* shows one deep Sierpinski Cube at recursion depth 5.

Table 8.9.: Single Menger-Sierpinski Test Results

			t	fps	mspf	DRAM load	DRAM store	Warp Efficiency
IRD	KLS	NIR	87.12	**75.63**	**13.22**	9.50	142.85	0.98
		HIR	20.16	27.68	36.12	7.46	29.77	0.97
		SIR	11.27	72.11	13.87	7.45	29.73	0.97
	PMK	NIR	110.43	74.99	13.33	**10.71**	170.30	0.93
		HIR	12.59	29.02	34.46	8.74	27.30	0.93
		SIR	12.59	71.78	13.93	9.06	26.99	0.93
PRD	KLS	NIR	122.84	74.55	13.41	5.66	186.85	0.99
		HIR	9.14	**27.47**	**36.41**	**2.13**	17.57	0.98
		SIR	9.12	73.26	13.65	**2.13**	**17.56**	0.98
	PMK	NIR	**126.44**	74.93	13.35	7.32	**188.23**	0.98
		HIR	**8.56**	29.42	33.99	3.23	20.67	0.97
		SIR	8.65	72.66	13.76	3.32	20.77	0.97

[a]

[a] **IRD** … Interpreted Rule Derivation **PRD** … Precompiled Rule Derivation **KLS** … Kernel Launch Scheduling **PMK** … Persistent Megakernel Scheduling **NIR** … Non-Instanced Rendering **HIR** … Hardware Instanced Rendering **SIR** … Software Instanced Rendering **t** … geometry generation time (in milliseconds) **fps** … frames per second **mspf** … milliseconds per frame **load/store** … memory operations (in millions)

8.2.5. Multi Menger-Sierpinski Test Case

For the Multi Menger-Sierpinski test case, we have turned down the number of recursions to three. The three recursion levels are not truly defined recursively like in the previous test case (8.2.4). Each level of recursion is written out by hand in the code (see A.5). By doing so, we increase the number of different rules for the scheduler. Therefore it sees the rules for different stages of the recursion as separate procedures, which impacts scheduling decisions. With this rule set, we generate a reasonable amount of computation and scheduling workload and are still able to generate a lot of geometry to stress the rendering pipeline. Table 8.10 shows the average performance statistics, while detailed numbers of each applied technique can be viewed in table 8.11.

Table 8.10.: Multi Menger-Sierpinski Average Results

Avg t	Avg fps	Avg mspf	Avg load	Avg store
27.00	101.04	12.05	3.51	43.16

Table 8.11.: Multi Menger-Sierpinski Test Results

			t	fps	mspf	DRAM load	DRAM store	Warp Efficiency
IRD	KLS	NIR	50.45	**127.95**	**7.82**	5.24	83.67	0.98
		HIR	10.76	48.63	20.56	3.97	16.78	0.97
		SIR	11.27	127.12	7.87	3.98	16.78	0.97
	PMK	NIR	68.01	127.25	7.86	**6.15**	104.16	0.88
		HIR	6.93	**48.54**	**20.60**	4.88	14.92	0.91
		SIR	6.93	127.42	7.85	4.90	14.93	0.91
PRD	KLS	NIR	73.45	125.96	7.94	2.99	111.97	0.99
		HIR	5.19	49.37	20.26	1.09	**9.70**	0.98
		SIR	5.65	127.53	7.84	**1.08**	9.72	0.98
	PMK	NIR	**75.56**	125.62	7.96	4.65	**112.19**	0.99
		HIR	**4.92**	49.56	20.18	1.61	11.58	0.98
		SIR	**4.92**	127.56	7.84	1.64	11.58	0.98

[a]

[a]**IRD** ... Interpreted Rule Derivation **PRD** ... Precompiled Rule Derivation **KLS** ... Kernel Launch Scheduling **PMK** ... Persistent Megakernel Scheduling
NIR ... Non-Instanced Rendering **HIR** ... Hardware Instanced Rendering **SIR** ... Software Instanced Rendering
t ... geometry generation time (in milliseconds) **fps** ... frames per second **mspf** ... milliseconds per frame **load/store** ... memory operations (in millions)

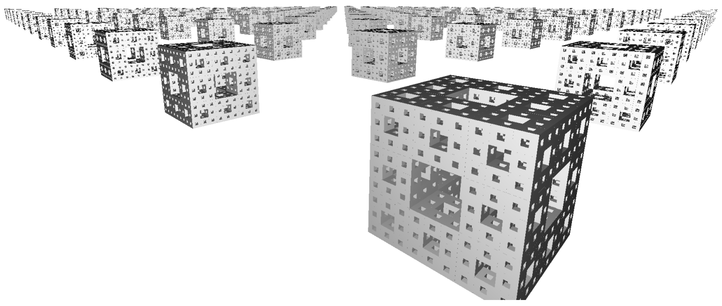


Figure 8.5.: *Multi Menger-Sierpinski* consists of 12×12 Sierpinski Cubes at recursion depth 3.

8.3. Evaluation

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

When looking at the raw generation times, we can observe that the fastest method can generate 153.9 million terminals per second (MTPS) in the Houses test case, 66.71 MTPS in the Many Rules Houses test case, 141.5 MTPS in the Detailed Houses test case, 224.3 MTPS in the Single Menger-Sierpinski test case and 234.2 MTPS for the Multi Sierpinski rule set with the used hardware (section 8.1).

8.3.1. Rule Derivation

Surprisingly, in the non-instanced variant, interpreted rule evaluation was faster than precompiled, achieving an overall faster average rule evaluation by 24%. In the instanced variant, the relationships are reversed, with precompiled outperforming interpreted by 59% on average (hardware and software instancing combined). In almost all instanced tests, compiled could generate the geometry faster. These are very interesting results, as precompiled is only significantly faster, when there is less memory traffic involved, due to the use of instancing. We can only assume that the interpreted evaluation can catch up in the non-instanced variant because all terminal operators are collected in the same queue. Thus, the terminal generation itself is highly efficient in comparison to the precompiled rule derivation, where the terminal generation is mixed with other operations. Thus, the interpreted version generates more homogeneous memory access patterns and overall runs faster in this case. This is also reflected by the lower number of DRAM stores in the interpreted non-instanced versions when comparing interpreted to percompiled. In the instanced variants, these numbers are reversed. Most of the memory access of the interpreted evaluation is due to dispatch table lookups and intermediate symbol generation, thus slowing down the generation process.

In the Many Rules Houses test case, which differs from the other tests significantly because it has more than three times as many rules than the other tests, interpreted rule derivation is fastest in every execution configuration. This can be explained by the procedure per operator paradigm of the interpretation method. The 90 different rule sets consist of the same operators. So while the precompiled approach has to handle 90 different queues, the relatively few queues to hold the operators in the interpreted method fill up nicely. Furthermore, because the same operators get executed repeatedly, very homogeneous memory access patterns emerge, fostering performance.

8.3.2. Task Scheduling

When comparing our iterative production implementation against the persistent megakernel approach, one can observe that the persistent megakernel implementation is on average 5% faster than the iterative production. So, interestingly, there is no generalizable pattern visible, as to when iterative production works better at first. When having a closer look, we can see that for the Many Rules Houses rule set, where there are significantly more procedures to schedule and less memory access involved, the megakernel approach outperforms iterative production by more than a factor of two and is 126.8% faster (both instancing methods combined) on the interpreted method, where all operators are procedures and can be merged in the same queue for execution. The iterative production gives best results for the combination of precompiled and instanced methods in four out of the five cases.

Having a look at the scheduling diagrams (full listing in appendix C), we can see that the utilization of the multi-processors on the graphics card is higher when using the persistent megakernel (PMK) approach (figure 8.7). The occupancy of the processing units strongly depend on the rule set. This supports the theory that not all computing tasks are suitable for massively parallel execution and that the optimal utilization of the hardware is heavily dependent on the sequence of operations. Furthermore, the chart 8.6 shows the delay caused by the round trip from the GPU to the CPU between the separate kernel launches (KLS) in the iterative production approach.

When comparing the plots 8.7 and 8.8, the characteristic variation in operations in the interpreted(IRD) stands out, while the execution paths are much more uniform in the precompiled (PRD) version.

The difference in memory access patterns, which has a large impact on performance, when comparing instanced (HIR) to non-instsanced rendering (NIR), is clearly visible, when comparing charts 8.8 and 8.9. The rule Crossbar is a terminal rule and occupies the processors for a significant amount of time.

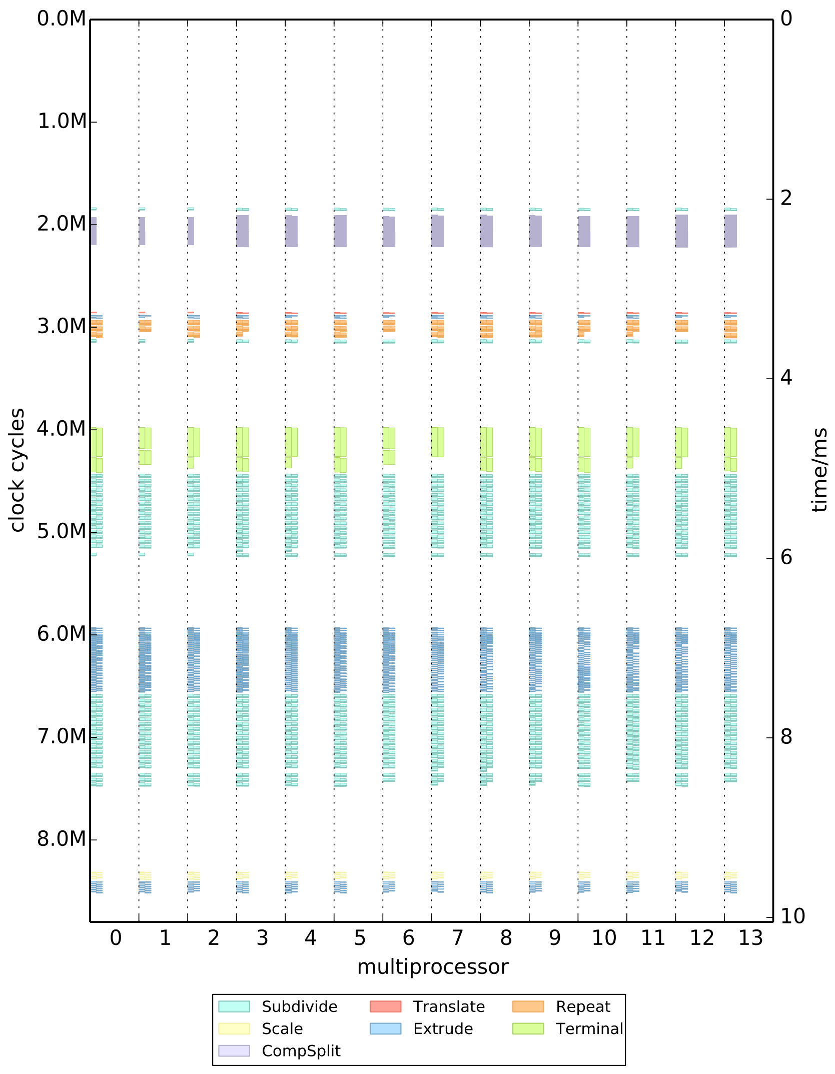


Figure 8.6.: *Detailed Houses Test Case Configuration: IRD KLS NIR*

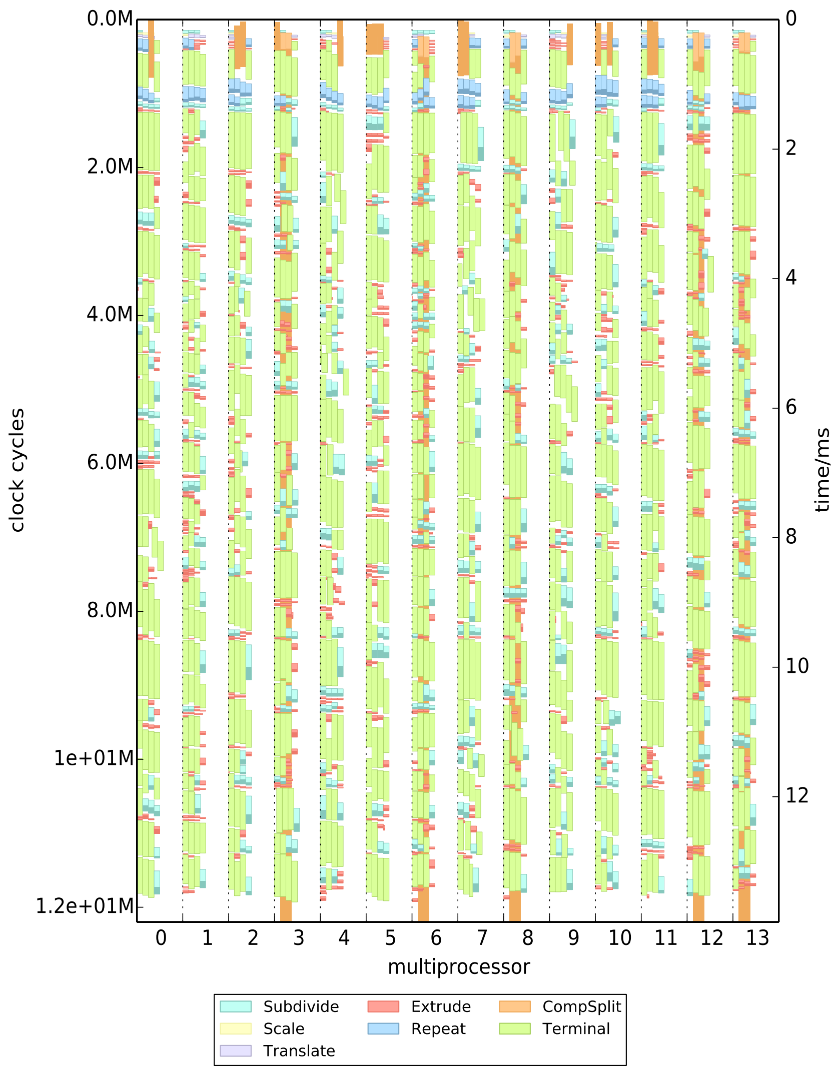


Figure 8.7.: *Detailed Houses Test Case Configuration: IRD PMK NIR*

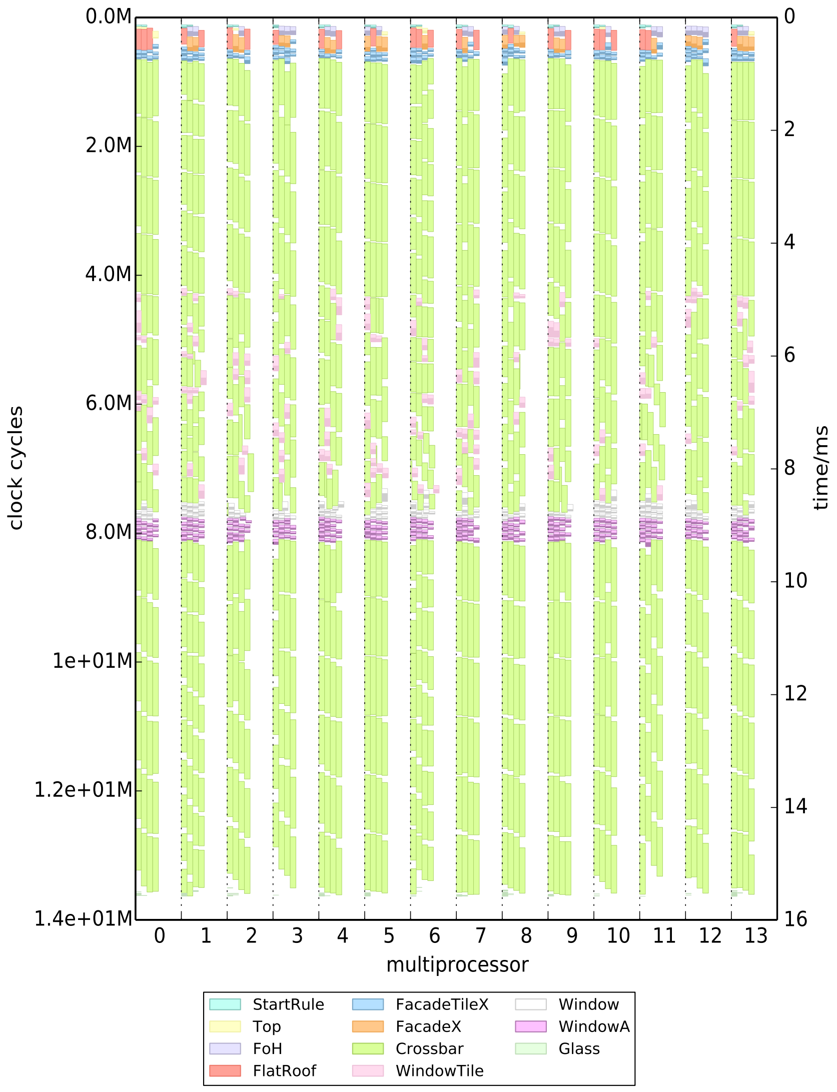


Figure 8.8.: *Detailed Houses Test Case Configuration: PRD PMK NIR*

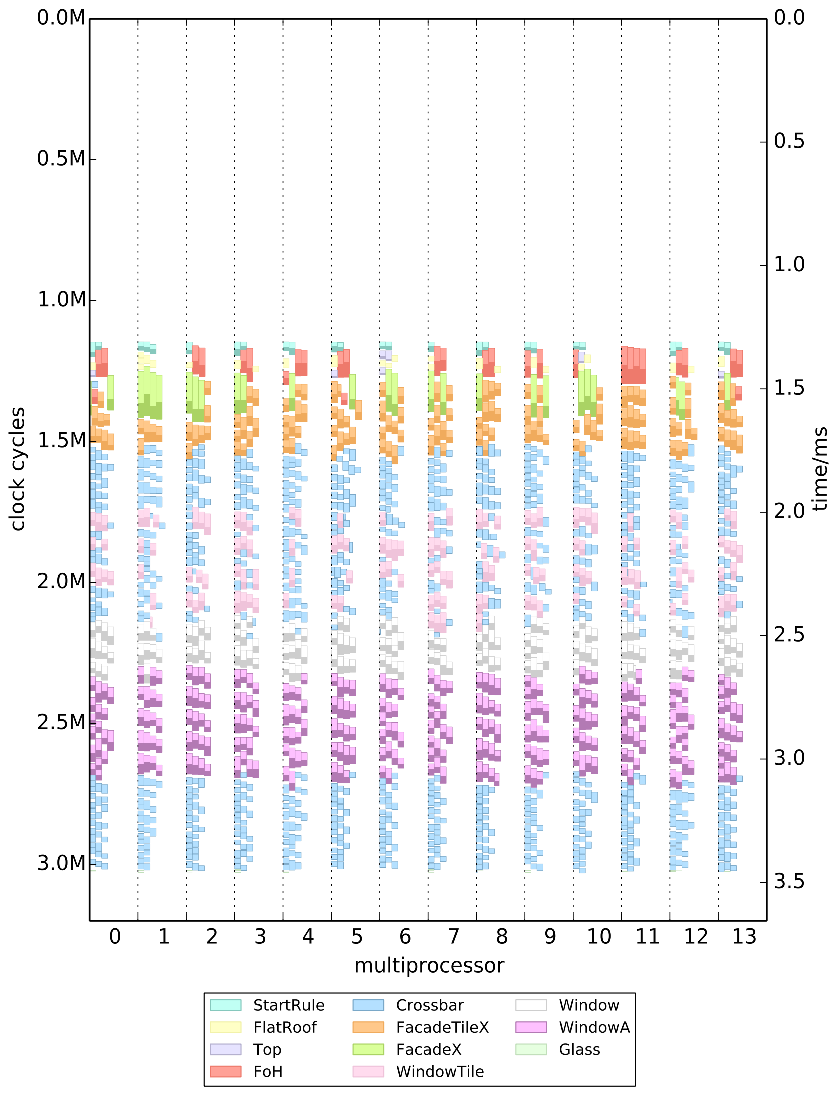


Figure 8.9.: *Detailed Houses Test Case Configuration: PRD PMK HIR*

8.3.3. Rendering

In all examples, the generation time in the instanced variant was between three to five times lower than the non-instanced variant. The highest difference was achieved in the Detailed Houses test case, which generates a vast amount of geometry with relatively few rule evaluations per terminal. This fact is clearly visible when we have a look at the number of DRAM stores.

When looking at the rendering times, we see that conventional non-instanced rendering always outperforms hardware instanced rendering, because on the one hand the hardware has been optimized for this kind of rendering, and on the other hand, there is an overhead involved in instancing.

What the numbers in the result tables do not tell, is that the performance of non-instanced rendering is dependent on the amount of visible content in the scene. When a lot of vertices have to be rendered, the frame rate drops significantly. When using hardware instancing, the frame rate stays constant whether or not the scene is filled with a great number of objects to draw. This leads to the conclusion that culling mechanisms have a significant impact on non-instanced rendering. The behavior of hardware instancing with the almost constant frame rate may be an implementation detail of the used graphics card.

The implementation of the software instancing approach was an interesting experiment to overcome the overhead of the OpenGL version. As can be seen from the numbers in the result tables, this method combines the benefits of both, instanced and non-instanced rendering. Unfortunately it comes at a price. This method consumes by far the most amount of memory since both the vertex data and the transformation matrices need to be stored for every terminal. But since the vertices need to be generated only once in the initialization phase, this data is already present in the graphic cards memory and does not involve expensive write operations anymore. Bandwidth requirements are therefore reduced to a relatively low amount for the transformation matrices. Therefore software instancing achieves the rendering speeds of the non-instanced version and the rule derivation times of the instanced method. If the software instancing approach can be improved, which can be investigated in future research, this rendering method is the most suitable one for procedural modeling. Another possibility would be to improve the OpenGL instancing implementation to achieve the frame rates of conventional vertex rendering.

9. Future Work

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

Two advanced rendering techniques would be interesting to see in a future release of our implementation. One is the implementation of a level of detail mechanism to produce simpler shapes of buildings that are far away with less detail visible, which increases performance significantly [Steinberger, Kenzel, Kainz, Wonka, et al., 2014]. The other technique that would be worth researching is a frame to frame coherence. This mechanism is used to determine what has changed in the currently visible scene, so expensive derivation steps can be saved if only little has changed from one frame to the next [Steinberger, Kenzel, Kainz, Wonka, et al., 2014].

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

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

Last but not least, an important aspect of CGA is context sensitivity. In our implementation this was deliberately left for future investigation, since the complex matter of rule interdependency is out of scope of this work. With this feature it would be possible to specify high level conditions like for example "generate a door, if the current floor is the ground floor".

10. Conclusion

We have shown in this work that scheduling of rule derivation work load on a GPU in the context of grammar based procedural modeling has several aspects influencing performance that have to be considered carefully.

First, decision making can be offloaded to the compilation stage in order to avoid expensive branching at run time. While this works out well in general, the amount of parallelization that can be achieved, which is higher in the interpreted approach in our implementation, is key to high performance.

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

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

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

Apart from the main topic, the derivation of grammar based procedural modeling rules, the comparison of the three different rendering methods has been insightful. Especially the hybrid software instancing looks promising, if it's obstacles can be mitigated.

Appendix

Appendix A.

Test Rule Sets

A.1. Houses Rule Set Code

Listing A.1: Houses Rule Set

```

#pragma once

#include "operators.cuh"

struct Roof : RuleT<Quad, extrude<Z, 100, GenerateTerminal> > {};

struct Window : RuleT<Quad, DiscardTerminal> {};

struct Ground : RuleT<Quad, DiscardTerminal> {};

struct WallTile : RuleT<Quad, extrude<Z, 100, GenerateTerminal > > {};

struct WindowTile : RuleT<Quad, subdivide<Y,
              SubdivParam<250, CallRule<WallTile>,
              SubdivParam<500, CallRule<Window>,
              SubdivParam<250, CallRule<WallTile> > > > > >  {};

struct FacadeTileX : RuleT<Quad, subdivide<X,
              SubdivParam<250, CallRule<WallTile>,
              SubdivParam<500, CallRule<WindowTile>,
              SubdivParam<250, CallRule<WallTile> > > > > > {};

struct FacadeX : RuleT<Quad, repeat<X, 500, CallRule<FacadeTileX> > > {};

struct Floor : RuleT<Box, compsplit<CSP<CallRule<Ground>,
                            CSP<CallRule<Roof>,
                            CSP<CallRule<FacadeX>,
                            CSP<CallRule<FacadeX>,
                            CSP<CallRule<FacadeX> ,
                            CSP<CallRule<FacadeX> > > > > > > > > {};

struct StartRule : RuleT<Box, repeat<Y, 1000, CallRule<Floor> > > {};

typedef RuleSet<RS<StartRule, RS<Floor, RS<FacadeX, RS<FacadeTileX,
        RS<WallTile, RS<WindowTile, RS<Window, RS<Roof,
        RS<Ground > > > > > > > > > > TheRules;
```

A.2. Detailed House Rule Set Code

Listing A.2: Detailed House Rule Set

```


#pragma once

#include "operators.cuh"

```

```
struct Ground : RuleT<Quad, rotate<180000, 0, 0,
          extrude<Z, 5, GenerateTerminal > > > {};

struct Glass : RuleT<Quad, DiscardTerminal> {};

struct FlatRoof : RuleT<Quad, extrude<Z, 5, GenerateTerminal> > {};

struct Roof : RuleT<Quad, extrude<Z, 100, GenerateTerminal> > {};

struct Crossbar : RuleT<Quad, extrude<Z, 25, GenerateTerminal> > {};

struct WindowA : RuleT<Quad, subdivide<Y,
          SubdivParam<450, CallRule<Glass>,
          SubdivParam<100, CallRule<Crossbar>,
          SubdivParam<450, CallRule<Glass> > > > > > {};

struct Window : RuleT<Quad, subdivide<X,
          SubdivParam<450, CallRule<WindowA>,
          SubdivParam<100, CallRule<Crossbar>,
          SubdivParam<450, CallRule<WindowA> > > > > > {};

struct WallTile : RuleT<Quad, extrude<Z, 25, GenerateTerminal > > {};

struct WindowTile : RuleT<Quad, subdivide<Y,
          SubdivParam<250, CallRule<WallTile>,
          SubdivParam<500, CallRule<Window>,
          SubdivParam<250, CallRule<WallTile> > > > > >  {};

struct FacadeTileX : RuleT<Quad, subdivide<X,
          SubdivParam<250, CallRule<WallTile>,
          SubdivParam<500, CallRule<WindowTile>,
          SubdivParam<250, CallRule<WallTile> > > > > > {};

struct FacadeX : RuleT<Quad, repeat<X, 400, CallRule<FacadeTileX> > > {};

struct GroundFacadeX : RuleT<Quad, repeat<X, 500, CallRule<FacadeTileX> > > {};

struct Floor : RuleT<Box, compsplit<CSP<CallRule<Ground>,
          CSP<CallRule<Roof>,
          CSP<CallRule<FacadeX>,
          CSP<CallRule<FacadeX>,
          CSP<CallRule<FacadeX> ,
          CSP<CallRule<FacadeX> > > > > > > > > {};

struct Inner : RuleT<Box, repeat<Y, 1000, CallRule<Floor> > > {};

struct BalkonyScope : RuleT<Box, DiscardTerminal> {};

struct Outer : RuleT<Box, scale<50, 2000 , 50,
          translate<0, -1000, 0, GenerateTerminal> > > {};

struct TopDivZ2 : RuleT<Box, subdivide<Z,
          SubdivParam<150, CallRule<BalkonyScope>,
          SubdivParam<700, CallRule<Inner>,
          SubdivParam<150, CallRule<BalkonyScope> > > > > > {};

struct TopDivZ1 : RuleT<Box, subdivide<Z,
          SubdivParam<150, CallRule<Outer>,
          SubdivParam<700, CallRule<BalkonyScope>,
          SubdivParam<150, CallRule<Outer> > > > > > {};

struct TopDivX : RuleT<Box, subdivide<X,
          SubdivParam<150, CallRule<TopDivZ1>,
          SubdivParam<700, CallRule<TopDivZ2>,
          SubdivParam<150, CallRule<TopDivZ1> > > > > > {};


struct DoorTile : RuleT<Quad, subdivide<Y,
          SubdivParam<25, CallRule<WallTile>,
          SubdivParam<725, CallRule<Glass>,
          SubdivParam<250, CallRule<WallTile> > > > > >  {};

struct Door : RuleT<Quad, subdivide<X,
          SubdivParam<250, CallRule<WallTile>,
          SubdivParam<500, CallRule<DoorTile>,
          SubdivParam<250, CallRule<WallTile> > > > > > {};

struct FoH : RuleT<Quad, subdivide<X,
          SubdivParam<350, CallRule<FacadeTileX>,
          SubdivParam<300, CallRule<Door>,
          SubdivParam<350, CallRule<FacadeTileX> > > > > > {};

struct GroundFloor : RuleT<Box, compsplit<
          CSP<CallRule<Ground>,
          CSP<CallRule<FlatRoof>,
          CSP<CallRule<GroundFacadeX>,
          CSP<CallRule<GroundFacadeX>,
          CSP<CallRule<FacadeX> ,
          CSP<CallRule<FoH> > > > > > > > > {};

struct Top : RuleT<Box, scale<2000, 4000, 2000,
            translate<0, 500, 0,
            CallRule<TopDivX> > > > {};

struct StartRule : RuleT<Box, subdivide<Y,
          SubdivParam<250, CallRule<GroundFloor>,
          SubdivParam<750, CallRule<Top> > > > > {};

typedef RuleSet<RS<StartRule, RS<Floor,
          RS<FacadeX, RS<FacadeTileX, RS<WallTile,
          RS<WindowTile, RS<Window, RS<Roof,
          RS<Ground, RS<WindowA, RS<Crossbar,
          RS<Glass, RS<GroundFloor, RS<FoH,
          RS<Top, RS<Door, RS<FlatRoof, RS<DoorTile,
          RS<GroundFacadeX, RS<TopDivX, RS<Inner,
          RS<Outer, RS<TopDivZ1, RS<TopDivZ2, RS<BalkonyScope
> > > > > >
> > > > > > > > >
> > > > > > > > > >
> TheRules;
```

A.3. Many Rules Houses Rule Set Code

Listing A.3: Many Rules Houses Rule Set

```
#define House(N) \
  struct Roof##N :    RuleT<Quad, extrude<Z, 100,
            GenerateTerminal> > {}; \
  struct Window##N :    RuleT<Quad, DiscardTerminal> {}; \
  struct Ground##N :    RuleT<Quad, DiscardTerminal> {}; \
  struct WallTile##N :  RuleT<Quad, extrude<Z, 100,
            GenerateTerminal > > {}; \
  struct WindowTile##N :  RuleT<Quad, subdivide<Y,
            SubdivParam<250, CallRule<WallTile##N>,
            SubdivParam<500, CallRule<Window##N>,
            SubdivParam<250, CallRule<WallTile##N> > > > > >  {}; \
  struct FacadeTileX##N : RuleT<Quad, subdivide<X,
            SubdivParam<250, CallRule<WallTile##N>,
            SubdivParam<500, CallRule<WindowTile##N>,
            SubdivParam<250, CallRule<WallTile##N> > > > > > {}; \
  struct FacadeX##N :   RuleT<Quad, repeat<X, 500,
            CallRule<FacadeTileX##N> > > {}; \
  struct Floor##N :   RuleT<Box,  compsplit<
            CSP<CallRule<Ground##N>,
            CSP<CallRule<Roof##N>,
            CSP<CallRule<FacadeX##N>,
            CSP<CallRule<FacadeX##N>,
            CSP<CallRule<FacadeX##N> ,
            CSP<CallRule<FacadeX##N> > > > > > > > > {}; \
  struct StartRule##N : RuleT<Box,  repeat<Y, 1000, CallRule<Floor##N> > > {};

House(9)
House(8)
House(7)
House(6)
House(5)
House(4)
House(3)
House(2)
House(1)
House()

#define HouseRuleDef(N) RS<StartRule##N, RS<Floor##N,
              RS<FacadeX##N, RS<FacadeTileX##N,
              RS<WallTile##N, RS<WindowTile##N,
              RS<Window##N, RS<Roof##N, RS<Ground##N

typedef RuleSet<HouseRuleDef(),  HouseRuleDef(1),  HouseRuleDef(2),
              HouseRuleDef(3), HouseRuleDef(4),
              HouseRuleDef(5),  HouseRuleDef(6),
              HouseRuleDef(7), HouseRuleDef(8) ,
              HouseRuleDef(9)
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > >
> > > > > > > > > > TheRules;
```

A.4. Single Menger-Sierpinski Rule Set Code

Listing A.4: Single Menger-Sierpinski Rule Set, Recursion Level 5

```

#define RECURSION_STOP 30

struct ZDiscard2 : RuleT<Box, subdivide<Z,
                  SubdivParam<333, GenerateTerminal,
                    SubdivParam<333, DiscardTerminal,
                    SubdivParam<333, GenerateTerminal > > > > > {};

struct SubY2 : RuleT<Box, subdivide<Y,
                    SubdivParam<333, GenerateTerminal,
                    SubdivParam<333, CallRule<ZDiscard2>,
                    SubdivParam<333, GenerateTerminal > > > > > {};

struct YDiscard2 : RuleT<Box, subdivide<Y,
                    SubdivParam<333, CallRule<ZDiscard2>,
                    SubdivParam<333, DiscardTerminal,
                    SubdivParam<333, CallRule<ZDiscard2> > > > > > {};

struct LastStage : RuleT<Box,  subdivide<X,
                    SubdivParam<333, CallRule<SubY2>,
                    SubdivParam<333, CallRule<YDiscard2>,
                    SubdivParam<333, CallRule<SubY2> > > > > > {};

struct StartRuleProxy {};

template <>
template <typename Shape, typename Q>
__device__ __inline__ void
CallRule<StartRuleProxy>::out(Shape & s, Q *q, int unused);

template <>
template <int symbol_offset, typename RuleSet>
__host__ __device__ __inline__ int CallRule<StartRuleProxy>::getSymbol();

struct ZDiscard : RuleT<Box, subdivide<Z,
                    SubdivParam<333, CallRule<StartRuleProxy>,
                    SubdivParam<333, DiscardTerminal,
                    SubdivParam<333, CallRule<StartRuleProxy> > > > > > {};

struct SubZ : RuleT<Box, subdivide<Z,
                    SubdivParam<333, CallRule<StartRuleProxy>,
                    SubdivParam<333, CallRule<StartRuleProxy>,
                   SubdivParam<333, CallRule<StartRuleProxy> > > > > > {};

struct YDiscard : RuleT<Box, subdivide<Y,
                   SubdivParam<333, CallRule<ZDiscard>,
                   SubdivParam<333, DiscardTerminal,
                   SubdivParam<333, CallRule<ZDiscard> > > > > > {};

struct SubY : RuleT<Box, subdivide<Y,
                   SubdivParam<333, CallRule<SubZ>,
                   SubdivParam<333, CallRule<ZDiscard>,
                   SubdivParam<333, CallRule<SubZ> > > > > > {};


struct StartRule : RuleT<Box,
                   IfSizeLess<X, RECURSION_STOP, CallRule<LastStage>,
                   subdivide<X, SubdivParam<333, CallRule<SubY>,
                   SubdivParam<333, CallRule<YDiscard>,
                   SubdivParam<333, CallRule<SubY> > > > > > > {};

template <>
template <typename Shape, typename Q>
__device__ __inline__ void
CallRule<StartRuleProxy>::out(Shape & s, Q *q, int unused)
{
  q->template enqueue<StartRule>(s);
}

template <>
template <int symbol_offset, typename RuleSet>
__host__ __device__ __inline__ int CallRule<StartRuleProxy>::getSymbol()
{
  return StartRule::getSymbol<
  RuleSet::template findSymbolOffset<StartRule>::value, RuleSet>();
}

typedef RuleSet<RS<StartRule, RS<SubY, RS<YDiscard, RS<SubZ,
        RS<ZDiscard, RS<LastStage,  RS<SubY2,
        RS<YDiscard2, RS<ZDiscard2 > > > > > > > > > > TheRules;
```

A.5. Multi Menger-Sierpinski Rule Set Code

Listing A.5: Multi Menger-Sierpinski Rule Set, Recursion Level 3

```
// level 3 ------------------------------------------------
struct ZDiscard3 : RuleT<Box, subdivide<Z,
                  SubdivParam<333, GenerateTerminal,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, GenerateTerminal> > > > > {};

struct SubZ3 : RuleT<Box, subdivide<Z,
                  SubdivParam<333, GenerateTerminal,
                  SubdivParam<333, GenerateTerminal,
                  SubdivParam<333, GenerateTerminal > > > > > {};

struct SubY3 : RuleT<Box, subdivide<Y,
                  SubdivParam<333,  CallRule<SubZ3>,
                  SubdivParam<333,   CallRule<ZDiscard3>,
                  SubdivParam<333, CallRule<SubZ3> > > > > > {};

struct YDiscard3 : RuleT<Box, subdivide<Y,
                  SubdivParam<333, CallRule<ZDiscard3>,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, CallRule<ZDiscard3> > > > > > {};

struct StartRule3 : RuleT<Box, subdivide<X,
                  SubdivParam<333, CallRule<SubY3>,
                  SubdivParam<333, CallRule<YDiscard3>,
                  SubdivParam<333, CallRule<SubY3> > > > > > {};


// level 2 ------------------------------------------------
```

```
struct ZDiscard2 : RuleT<Box, subdivide<Z,
                  SubdivParam<333, CallRule<StartRule3>,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, CallRule<StartRule3> > > > > > {};

struct SubZ2 : RuleT<Box, subdivide<Z,
                  SubdivParam<333, CallRule<StartRule3>,
                  SubdivParam<333, CallRule<StartRule3>,
                  SubdivParam<333, CallRule<StartRule3> > > > > > {};

struct SubY2 : RuleT<Box, subdivide<Y,
                  SubdivParam<333,  CallRule<SubZ2>,
                  SubdivParam<333,   CallRule<ZDiscard2>,
                  SubdivParam<333, CallRule<SubZ2> > > > > > {};

struct YDiscard2 : RuleT<Box, subdivide<Y,
                  SubdivParam<333, CallRule<ZDiscard2>,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, CallRule<ZDiscard2> > > > > > {};

struct StartRule2 : RuleT<Box, subdivide<X,
                  SubdivParam<333, CallRule<SubY2>,
                  SubdivParam<333, CallRule<YDiscard2>,
                  SubdivParam<333, CallRule<SubY2> > > > > > {};


// level 1 -----------------------------------------------
struct ZDiscard : RuleT<Box, subdivide<Z,
                  SubdivParam<333, CallRule<StartRule2>,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, CallRule<StartRule2> > > > > > {};

struct SubZ : RuleT<Box, subdivide<Z,
                  SubdivParam<333, CallRule<StartRule2>,
                  SubdivParam<333, CallRule<StartRule2>,
                  SubdivParam<333, CallRule<StartRule2> > > > > > {};

```

```
struct SubY : RuleT<Box, subdivide<Y,
                  SubdivParam<333, CallRule<SubZ> ,
                  SubdivParam<333, CallRule<ZDiscard>,
                  SubdivParam<333, CallRule<SubZ> > > > > > {};

struct YDiscard : RuleT<Box, subdivide<Y,
                  SubdivParam<333, CallRule<ZDiscard>,
                  SubdivParam<333, DiscardTerminal,
                  SubdivParam<333, CallRule<ZDiscard> > > > > > {};

struct StartRule : RuleT<Box, subdivide<X,
                  SubdivParam<333, CallRule<SubY>,
                  SubdivParam<333, CallRule<YDiscard>,
                  SubdivParam<333, CallRule<SubY> > > > > > {};

typedef RuleSet<RS<StartRule,  RS<SubY, RS<SubZ, RS<YDiscard, RS<ZDiscard,
        RS<StartRule2,  RS<SubY2, RS<SubZ2, RS<YDiscard2, RS<ZDiscard2,
        RS<StartRule3, RS<SubY3, RS<SubZ3, RS<YDiscard3, RS<ZDiscard3>
        > > > > > > > > > > > > > > > TheRules;
```

Appendix B.

Source Code Examples

B.1. GenerateTerminal Code Example

Listing B.1: GenerateTerminal Operator for a Quad

```
template <typename Q>
__device__ __inline__ static void out(Quad &shape, Q *q, int unused)
{
  if (d_instanced)
  {
    unsigned int instanceIndex =
    atomicAdd((unsigned int *)&d_instanceCounter[QUAD], 1);
    math::float4x4 scaleMat = math::float4x4::scale(shape.getSize());
    d_model_matrices[QUAD][instanceIndex] =
    transpose(shape.getModel4() * scaleMat);
  }
  else
  {
    unsigned int vertexIndex;
    vertexIndex = atomicAdd((unsigned int *)&d_vertexCounter, 4);
#ifdef INSTANCED
    if(vertexIndex / 4 > current_shapetype_max_instances)
    {
      return;
    }
    else
    {
#endif
    unsigned int index;
    index = atomicAdd((unsigned int *)&d_indexCounter, 6);

    math::float4 *vertices = d_vertices;
    math::float3 *normals = d_normals;
    unsigned int *indices = d_indices;

    math::float4x4 model = shape.getModel4();
    math::float3 halfExtents = shape.getHalfExtents();

    // generate a front facing quad (z == 0)
    vertices[vertexIndex] =
    model * math::float4(-halfExtents.x, halfExtents.y, -halfExtents.z, 1);
    vertices[vertexIndex + 1] =
    model * math::float4(halfExtents.x, halfExtents.y, halfExtents.z, 1);
    vertices[vertexIndex + 2] =
    model * math::float4(halfExtents.x, -halfExtents.y, halfExtents.z, 1);
    vertices[vertexIndex + 3] =
    model * math::float4(-halfExtents.x, -halfExtents.y, -halfExtents.z, 1);

    normals[vertexIndex] =
    normalize(cross((vertices[vertexIndex + 1] -
    vertices[vertexIndex]).xyz(), (vertices[vertexIndex + 2] -
    vertices[vertexIndex]).xyz()));
    normals[vertexIndex + 1] = normals[vertexIndex];
    normals[vertexIndex + 2] = normals[vertexIndex];
    normals[vertexIndex + 3] = normals[vertexIndex];

```

```
      unsigned int baseIndex = vertexIndex;
      indices[index++] = baseIndex;
      indices[index++] = baseIndex + 1;
      indices[index++] = baseIndex + 2;
      indices[index++] = baseIndex;
      indices[index++] = baseIndex + 2;
      indices[index++] = baseIndex + 3;
    }
#ifdef INSTANCED
  }
#endif
}
```

B.2. Precompiled Init Procedure Code Example

Listing B.2: Init Procedure for Precompiled Method

```
class PrecompiledInitProc
{
public:
  static const bool reuseInit = false;
  template <class Q>
  __device__ __inline__ static void init(Q *q, int id, int frame)
  {
    Box box;
    box.setModel(math::identity<math::float4x4>());
    box.setPosition(math::float3(0.0f, 0.0f, 0.0f));
    box.setSize(math::float3(INITIAL_WIDTH,
                    INITIAL_HEIGHT,
                    INITIAL_WIDTH));
    q->template enqueueInitial<StartRule>(box);


  }
};
```

B.3. Interpreted Init Procedure Code Example

Listing B.3: Init Procedure for Interpretation Method

```
class InterpretedInitProc
{
public:
  static const bool reuseInit = false;

  template <class Q>
  __device__ __inline__ static void init(Q *q, int id, int frame)
  {

    SymbolWrapper<Box> box;
    box.symbol = TheRules::symbol_count - 1;
    box.setModel(math::identity<math::float4x4>());
    box.setPosition(math::float3(0.0f, 0.0f, 0.0f));
    box.setSize(math::float3(INITIAL_WIDTH,
                  INITIAL_HEIGHT,
                  INITIAL_DEPTH));
    enqueueInitialProc<Box, Q, TheRules>(box, q);

  }
};
```

B.4. Dispatcher Switch Code Example

Listing B.4: Dispatcher Switch

```
template <typename RuleSet, typename Shape, typename Q>
__device__ bool enqueueNextProc(SymbolWrapper<Shape>& target, Q* q)
{
  Rule rule = d_rules[target.symbol];

  switch (rule.operatorCode)
  {
      case RuleSet::template findProcId<
                            OperatorRuleCompSplit<Shape, RuleSet> >::value:
        q->template enqueue<OperatorRuleCompSplit<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
                          OperatorRuleExtrude<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleExtrude<Shape, RuleSet> >(target);
      break;
      case RuleSet::template findProcId<
                OperatorRuleSubdivide<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleSubdivide<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
                OperatorRuleRepeat<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleRepeat<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
              OperatorRuleTranslate<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleTranslate<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
              OperatorRuleRotate<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleRotate<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
              OperatorRuleScale<Shape, RuleSet> >::value:
      q->template enqueue<OperatorRuleScale<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
```

```
              IfSizeLessOperatorProcedure<Shape, RuleSet> >::value:
      q->template enqueue<
              IfSizeLessOperatorProcedure<Shape, RuleSet> >(target);
      break;
    case RuleSet::template findProcId<
      OperatorRuleTerminal<Shape> >::value:
      {
        q->template enqueue<OperatorRuleTerminal<Shape> >(target);
        break;
      }
    case RuleSet::template findProcId<OperatorRuleDiscard<Shape> >::value:
      {
        q->template enqueue<OperatorRuleDiscard<Shape> >(target);
        break;
      }
    default:
      printf("Error: unknown operator %d\n", rule.operatorCode);
      return false;
  }
  return true;
}
```

Appendix C.

Scheduling Diagrams

C.1. Detailed House Test Case

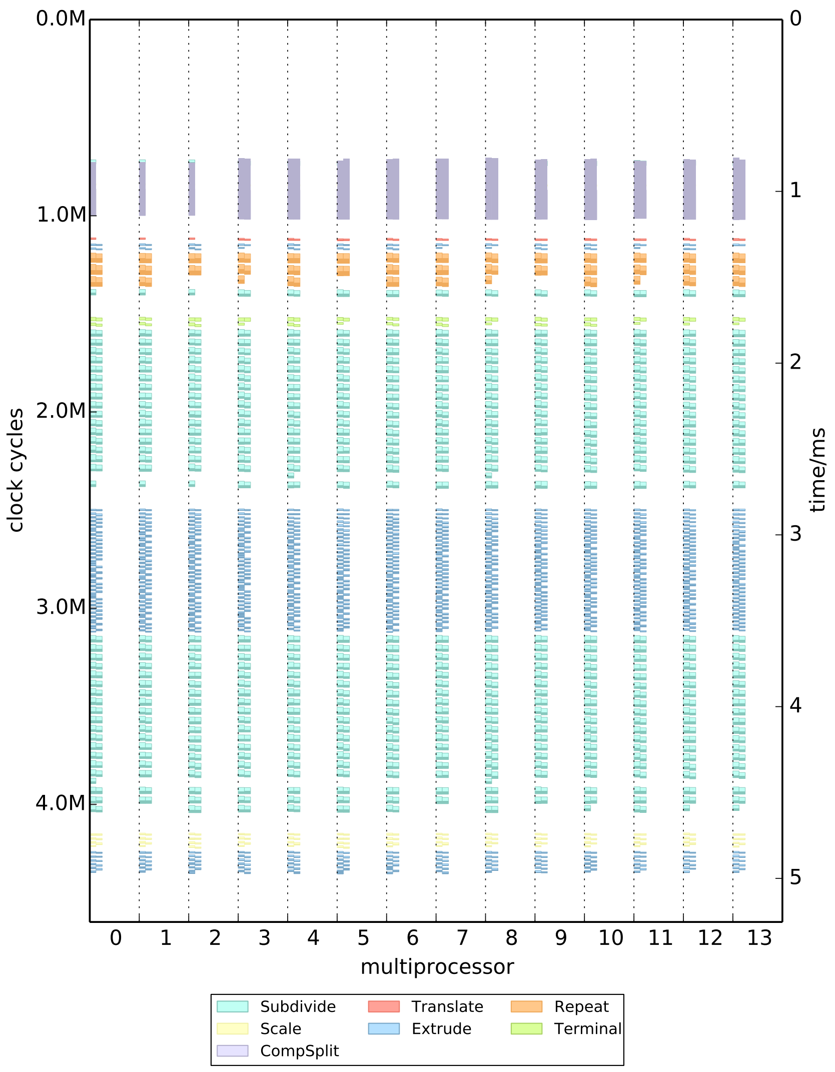


Figure C.1.: *Detailed Houses Test Case Configuration: IRD KLS HIR*

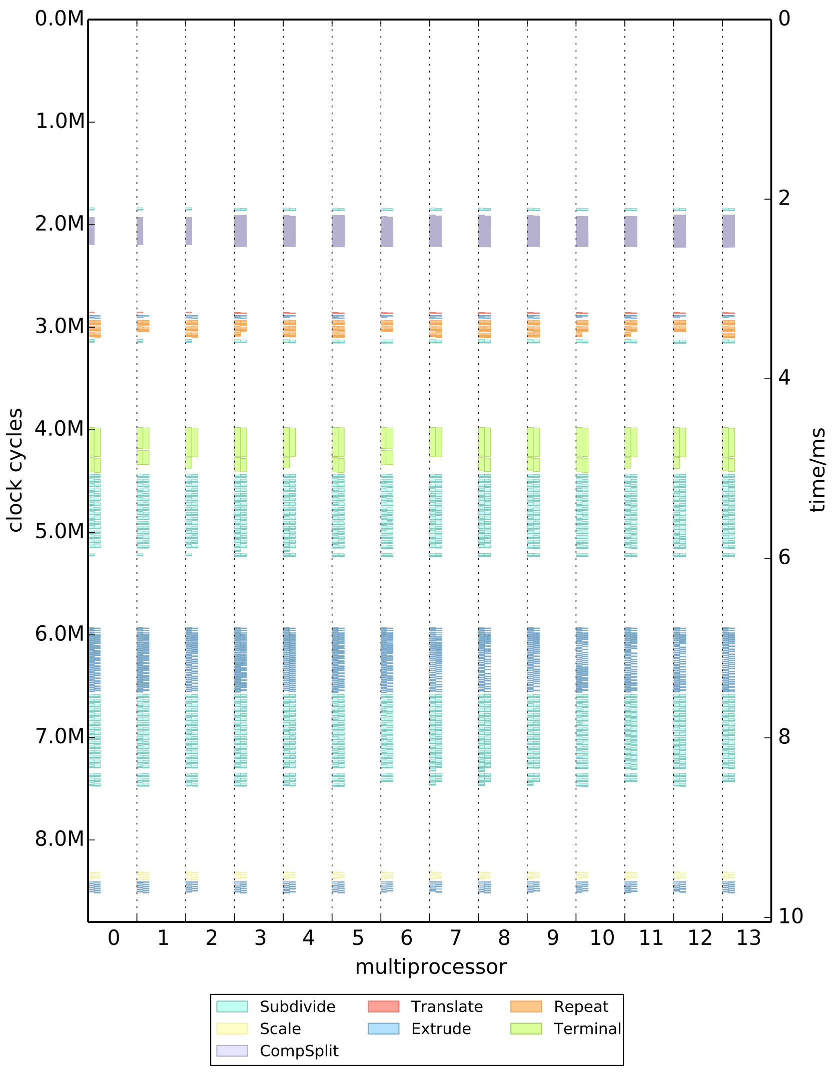


Figure C.2.: *Detailed Houses Test Case Configuration: IRD KLS NIR*

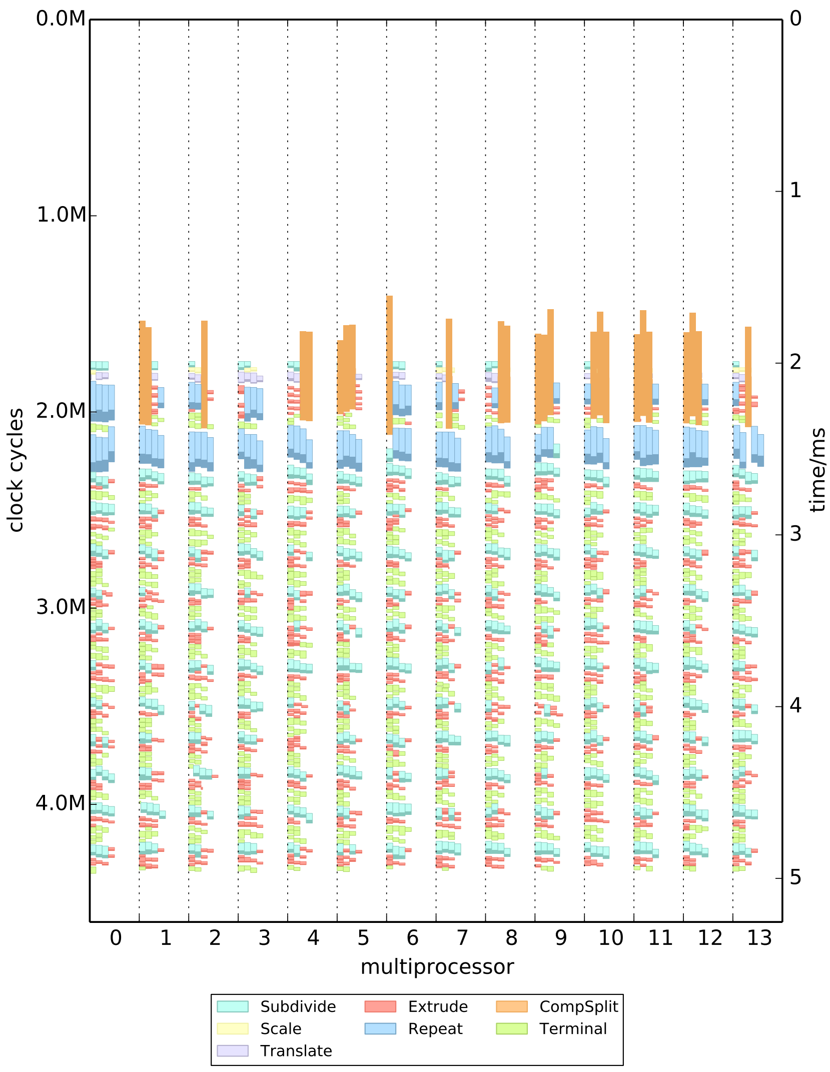


Figure C.3.: *Detailed Houses Test Case Configuration: IRD PMK HIR*

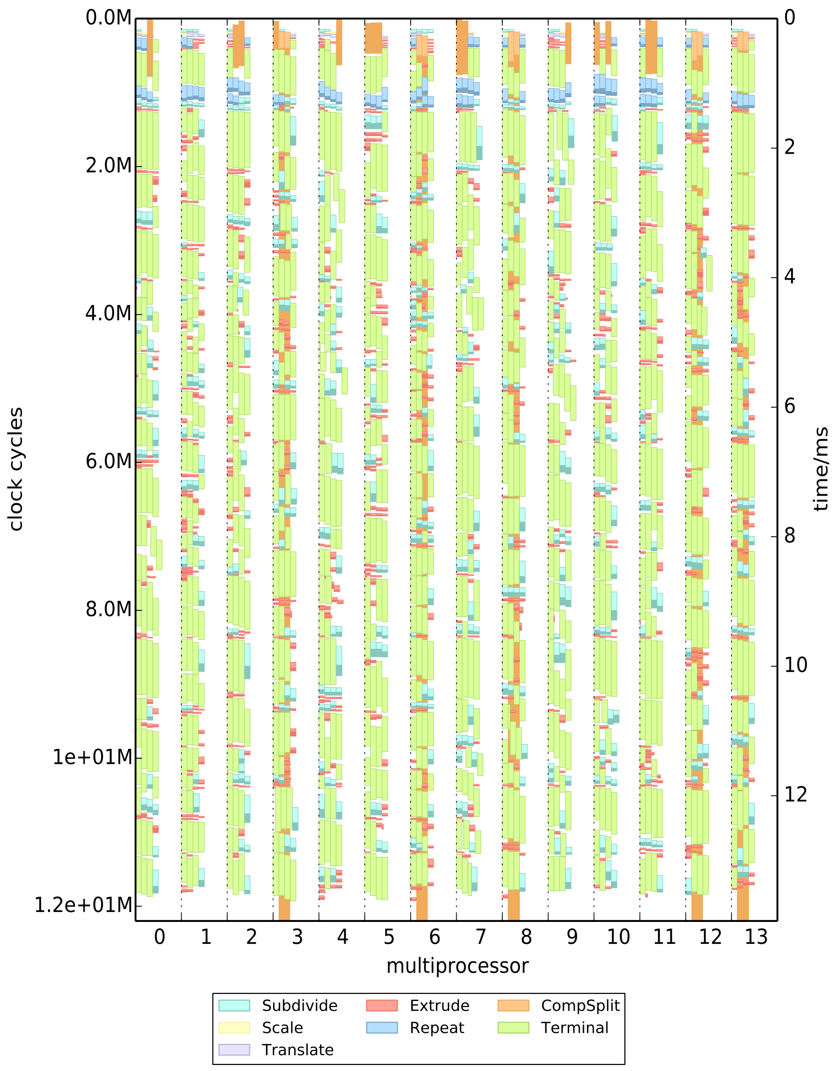


Figure C.4.: *Detailed Houses Test Case Configuration: IRD PMK NIR*

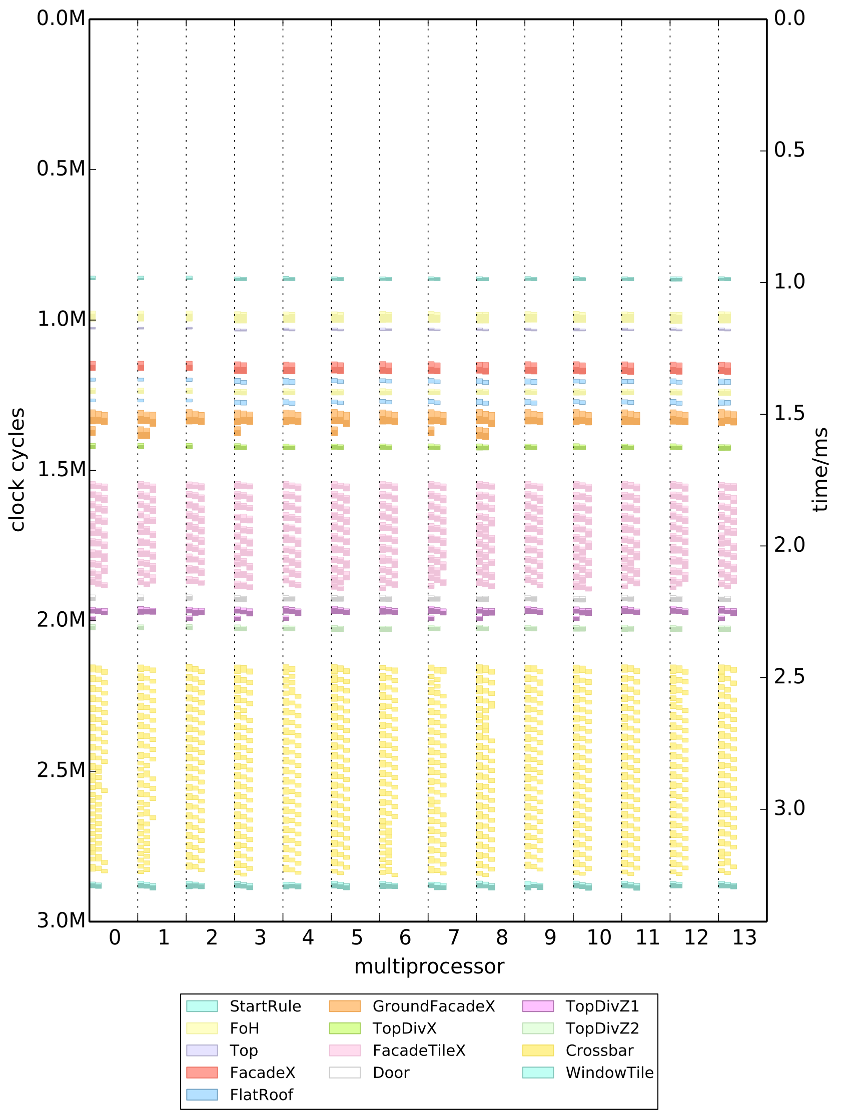


Figure C.5.: *Detailed Houses Test Case Configuration: PRD KLS HIR*

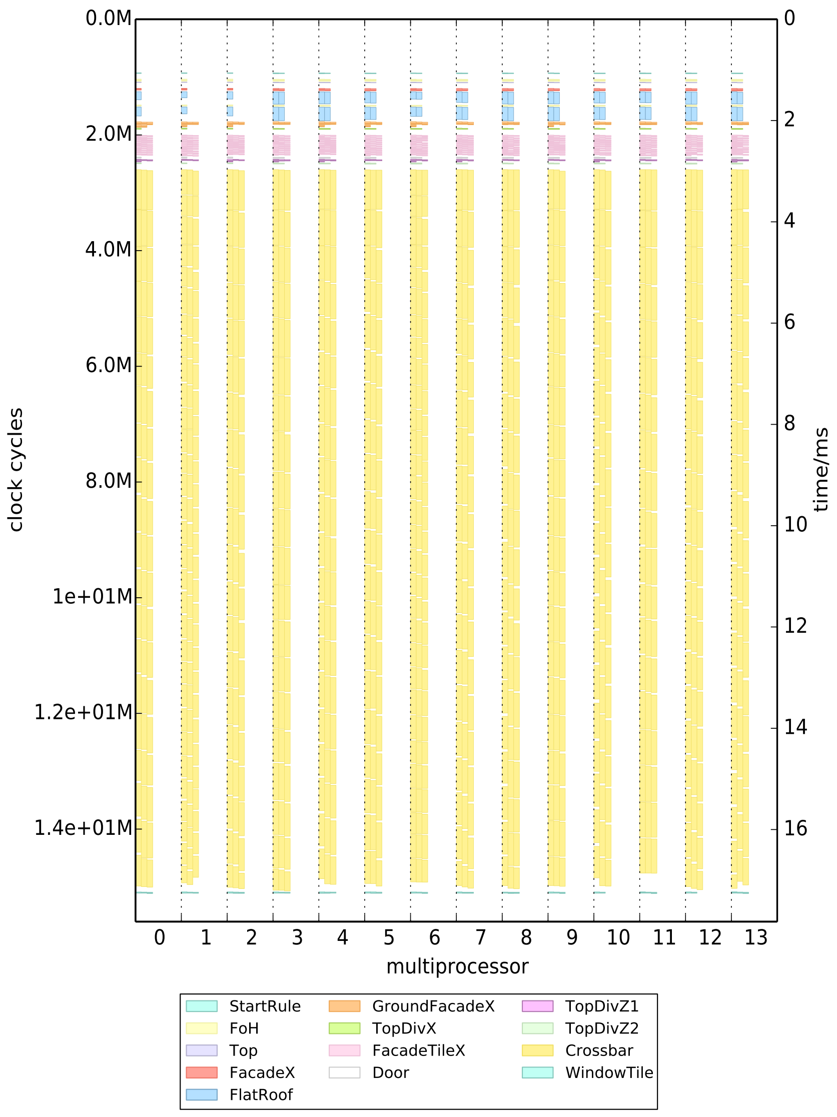


Figure C.6.: *Detailed Houses Test Case Configuration: PRD KLS NIR*

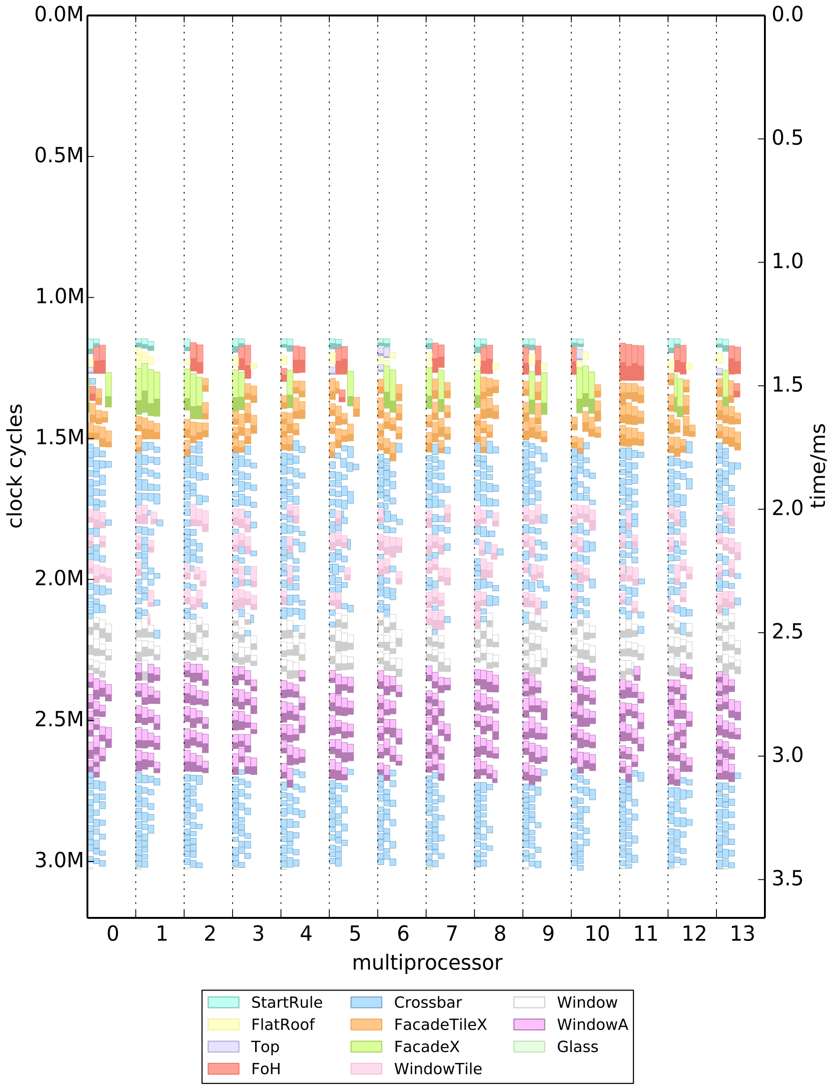


Figure C.7.: *Detailed Houses Test Case Configuration: PRD PMK HIR*

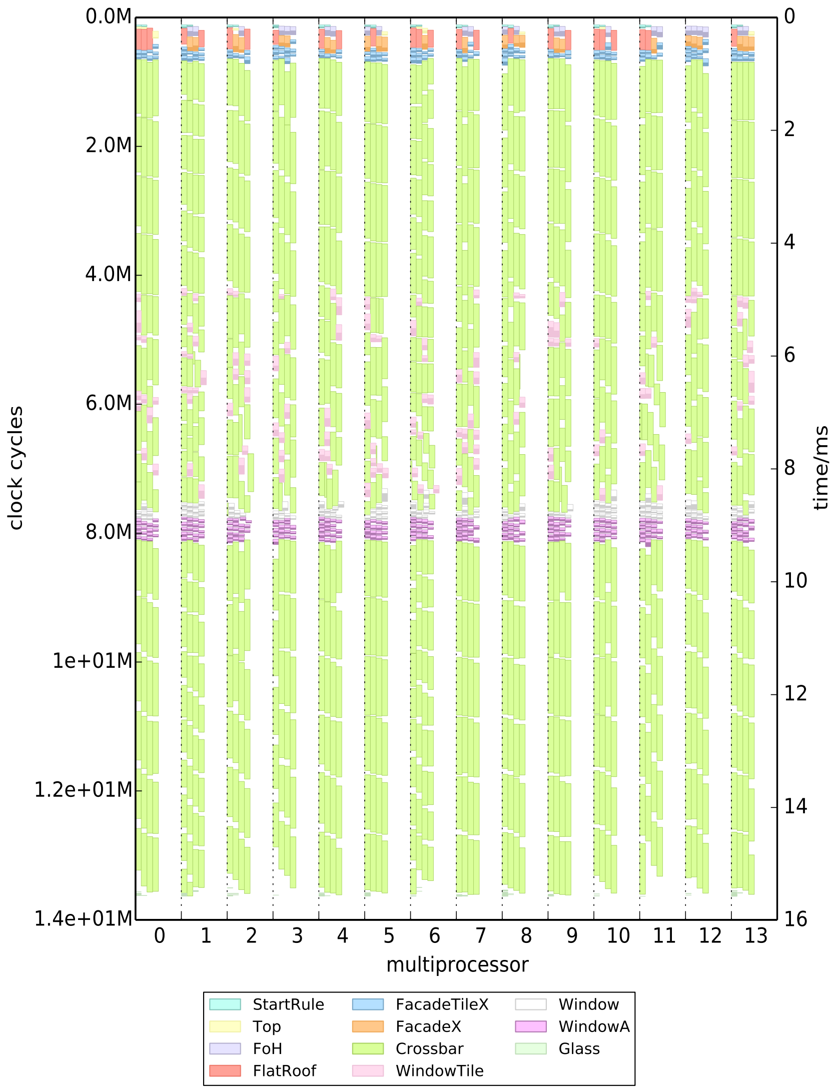


Figure C.8.: *Detailed Houses Test Case Configuration: PRD PMK NIR*

C.2. Houses Test Case

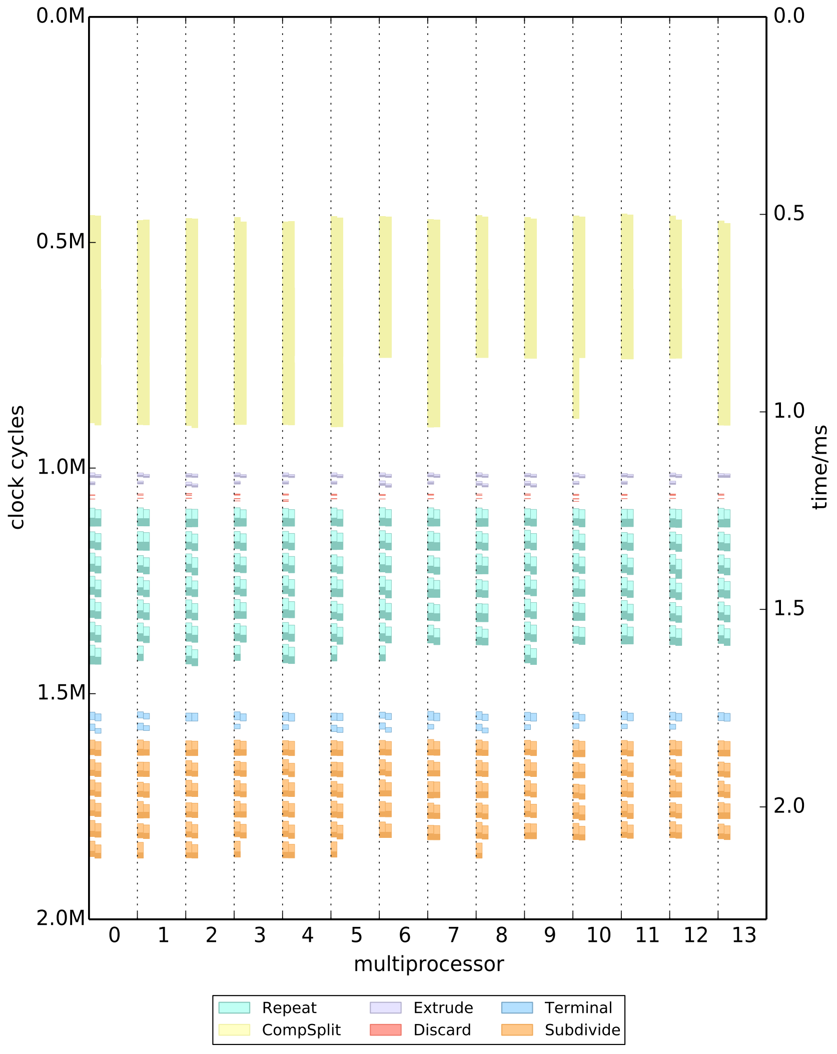


Figure C.9.: *Houses Test Case Configuration: IRD KLS HIR*

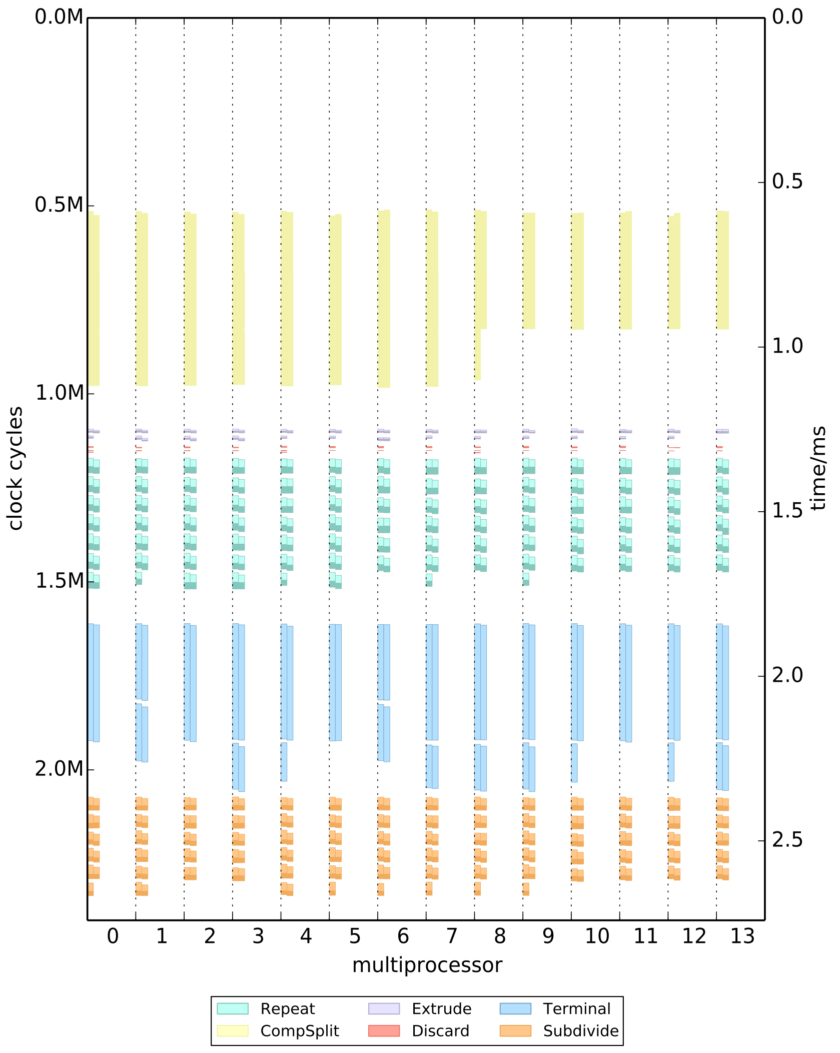


Figure C.10.: *Houses Test Case Configuration: IRD KLS NIR*

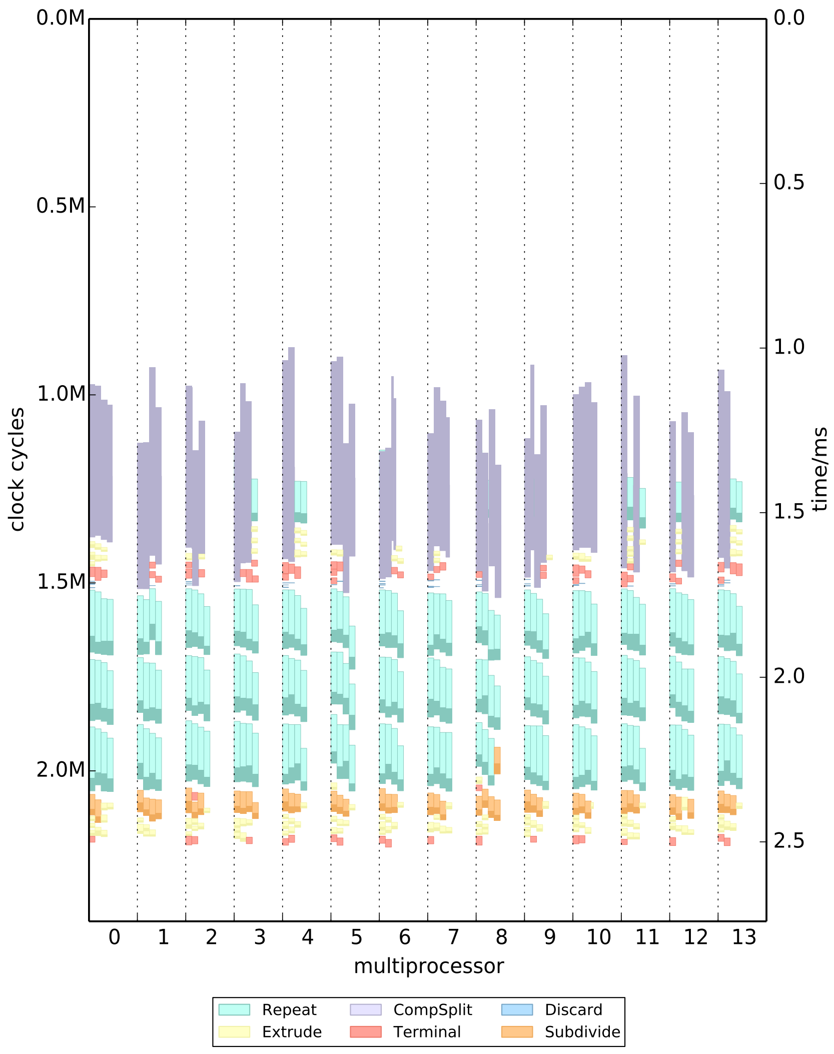


Figure C.11.: *Houses Test Case Configuration: IRD PMK HIR*

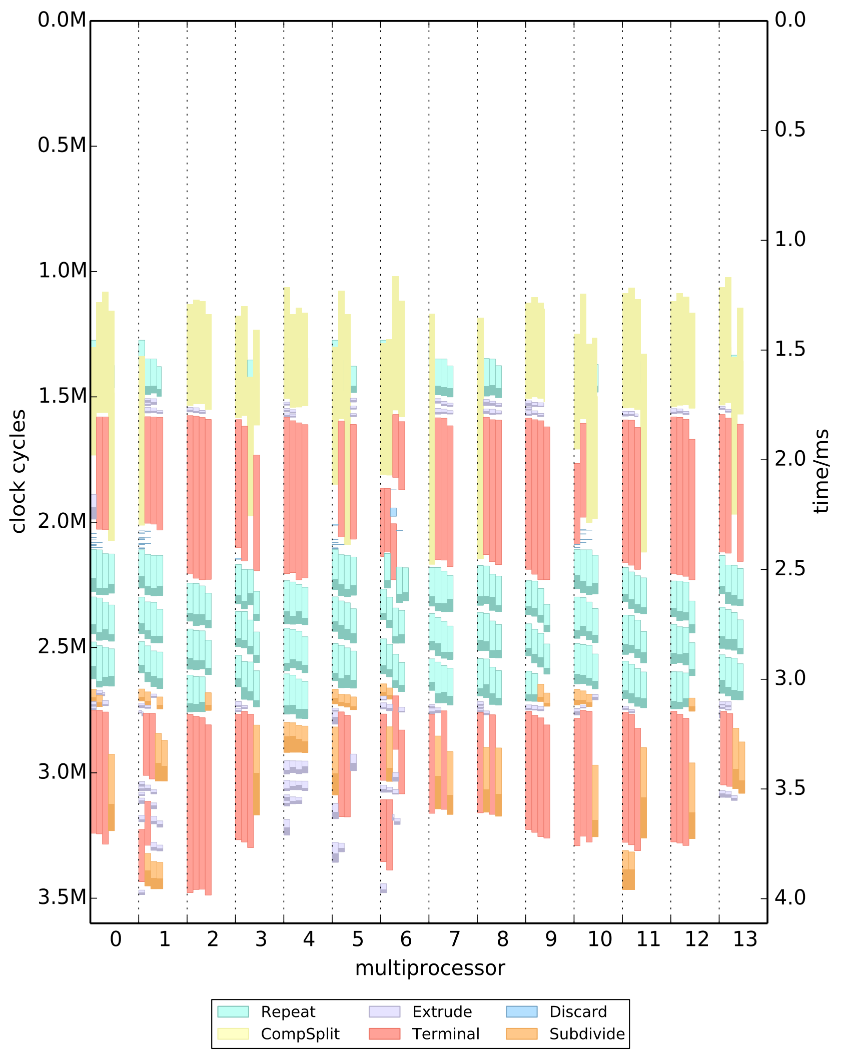


Figure C.12.: *Houses Test Case Configuration: IRD PMK NIR*

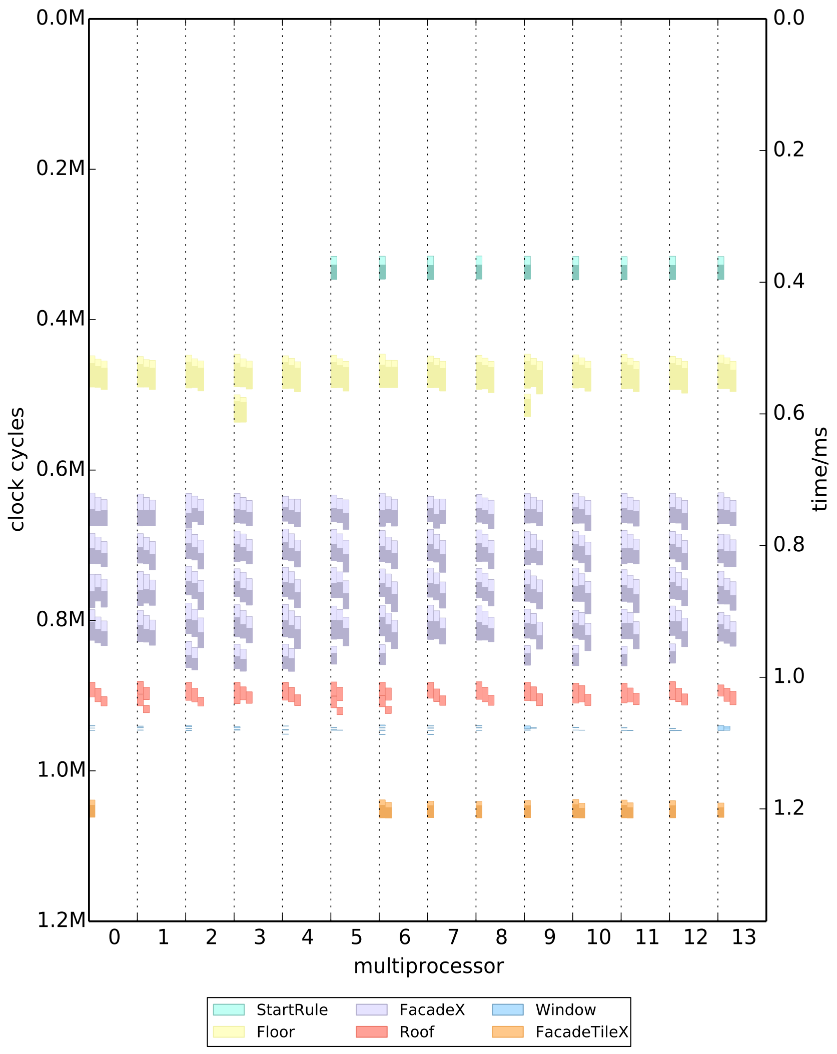


Figure C.13.: *Houses Test Case Configuration: PRD KLS HIR*

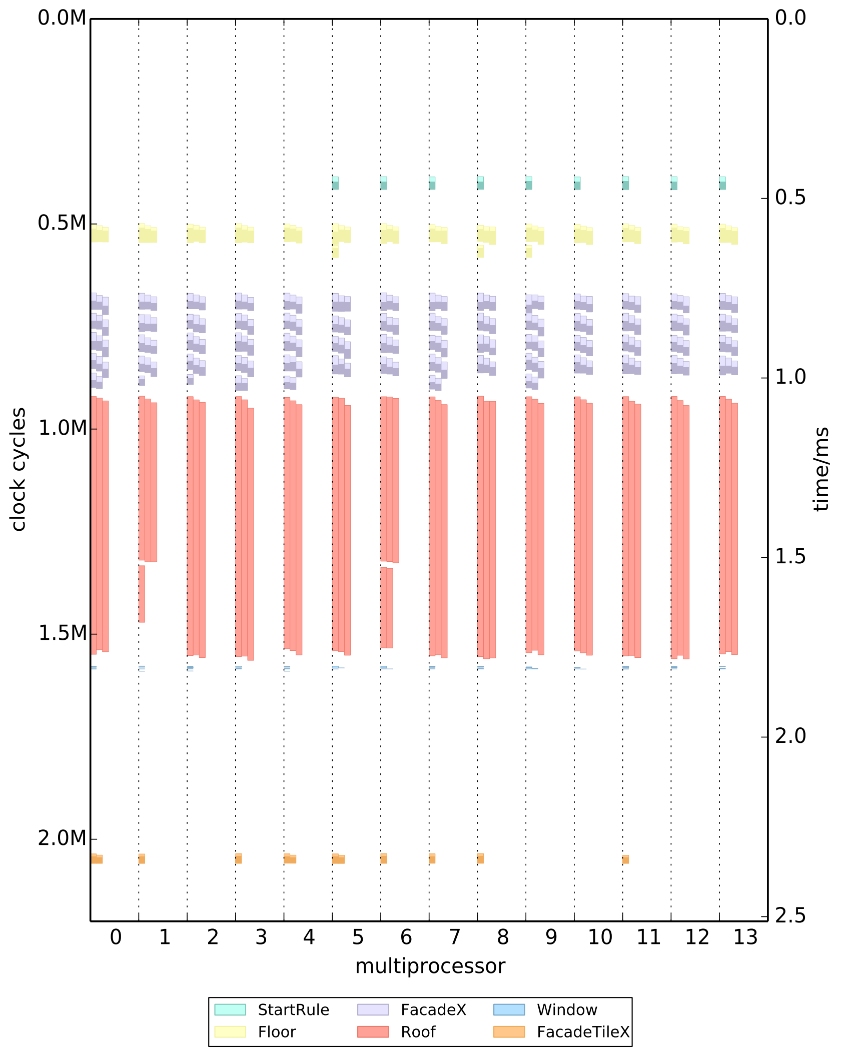


Figure C.14.: *Houses Test Case Configuration: PRD KLS NIR*

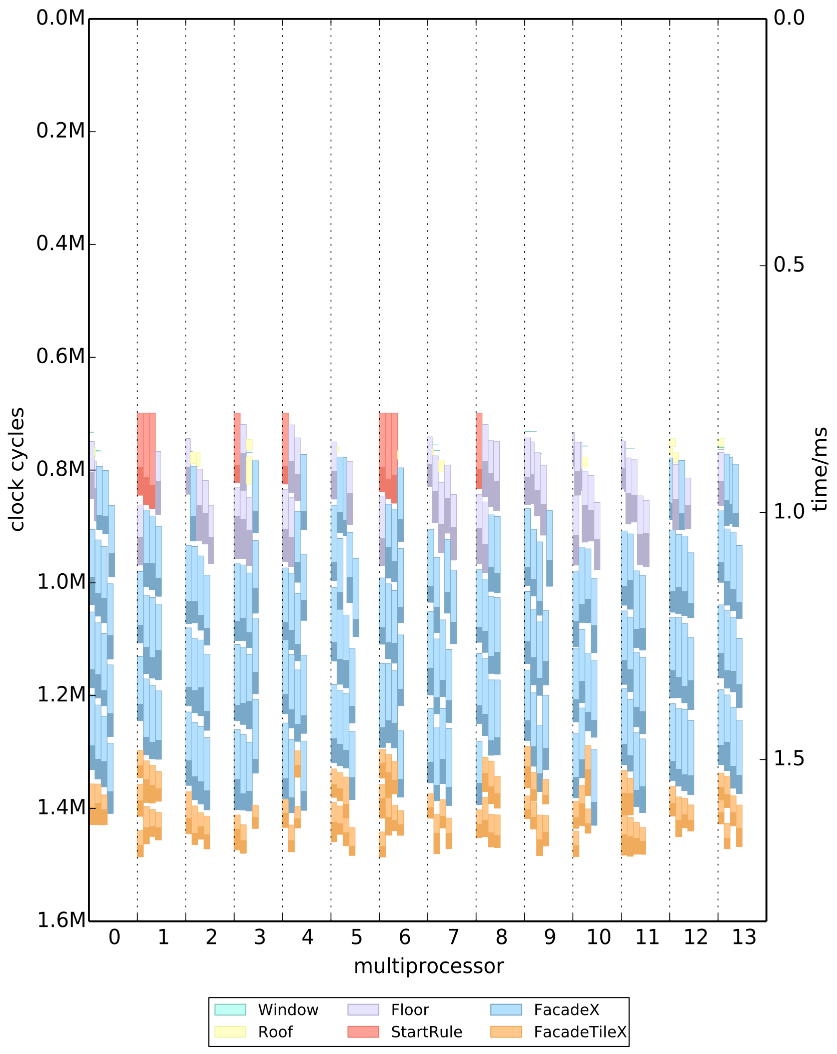


Figure C.15.: *Houses Test Case Configuration: PRD PMK HIR*

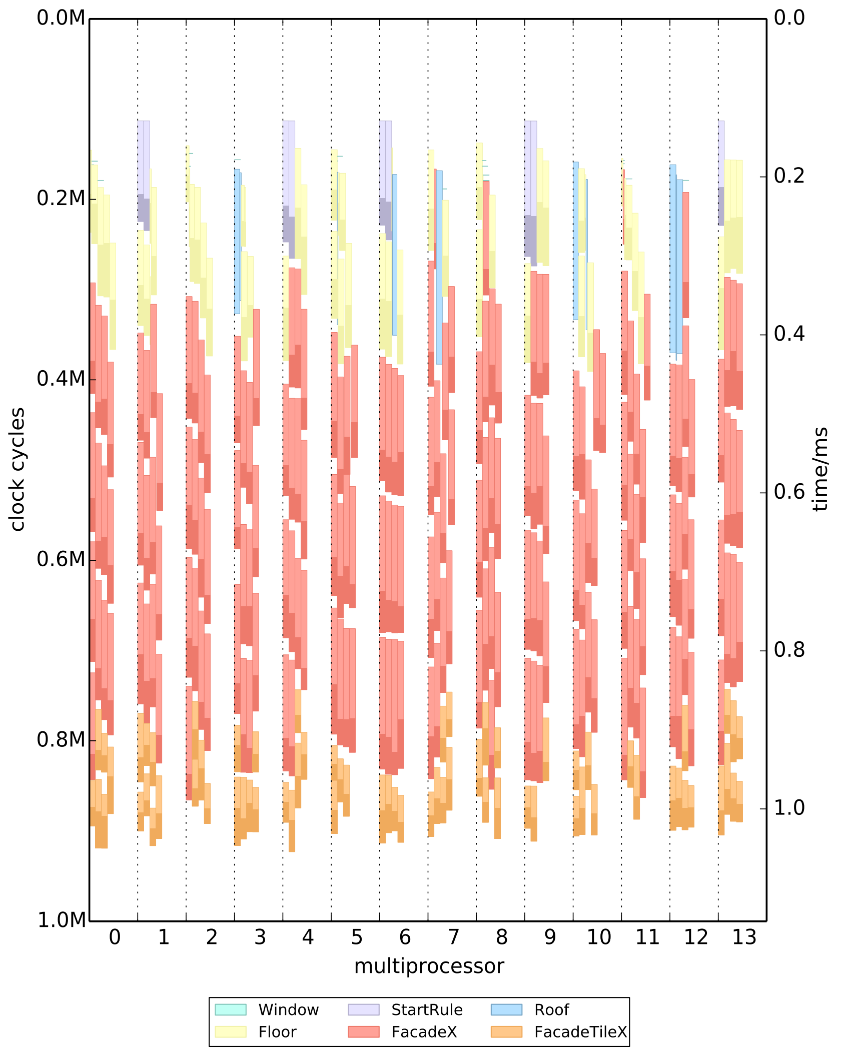


Figure C.16.: *Houses Test Case Configuration: PRD PMK NIR*

C.3. Multi Sierpinski Test Case

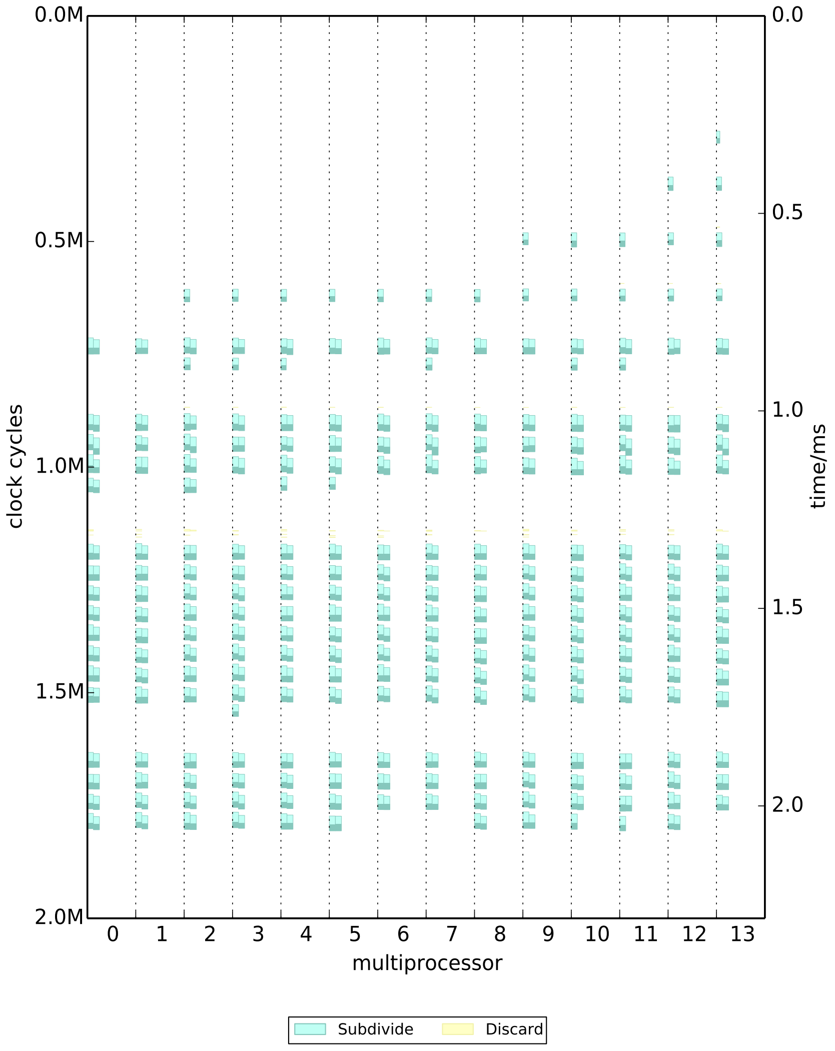


Figure C.17.: *Multi Sierpinski Test Case Configuration: IRD KLS HIR*

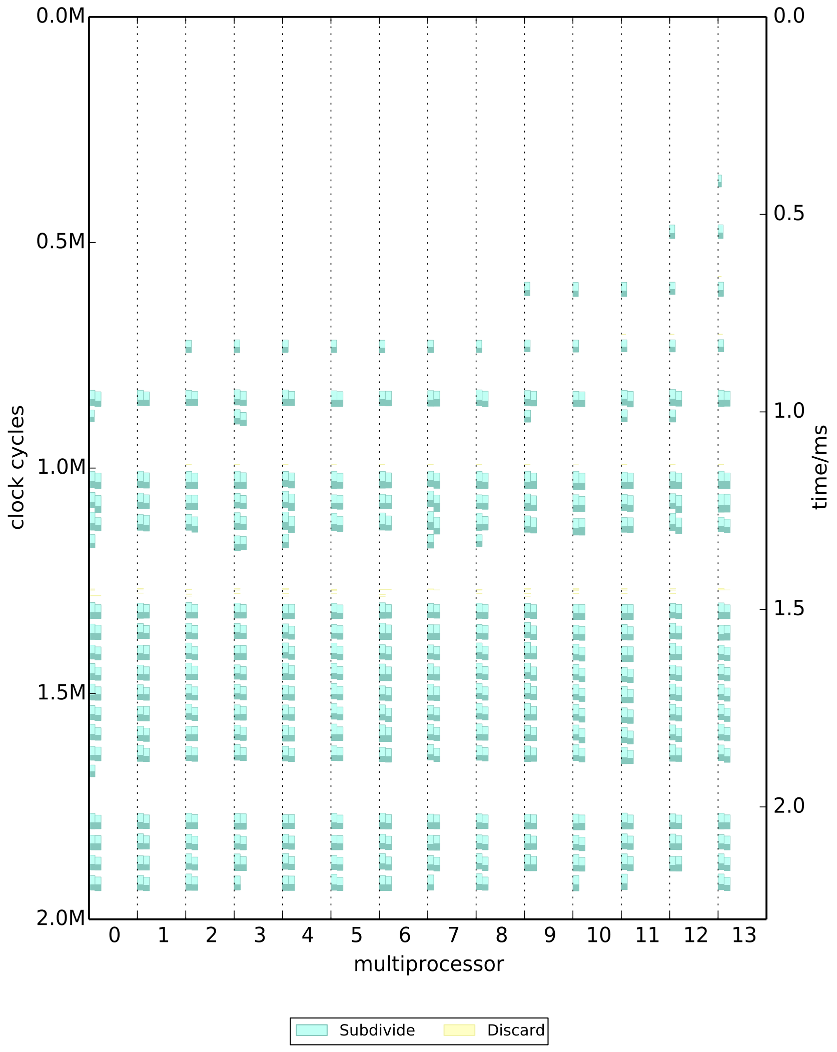


Figure C.18.: *Multi Sierpinski Test Case Configuration: IRD KLS NIR*

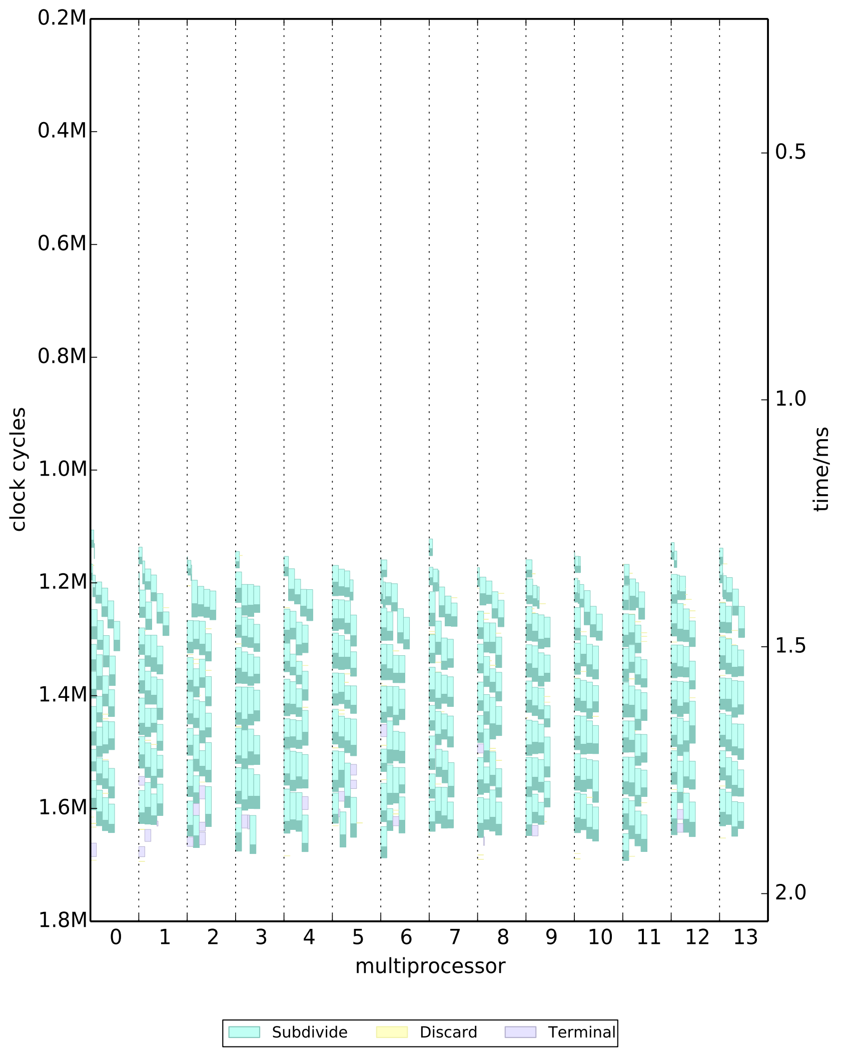


Figure C.19.: *Multi Sierpinski Test Case Configuration: IRD PMK HIR*

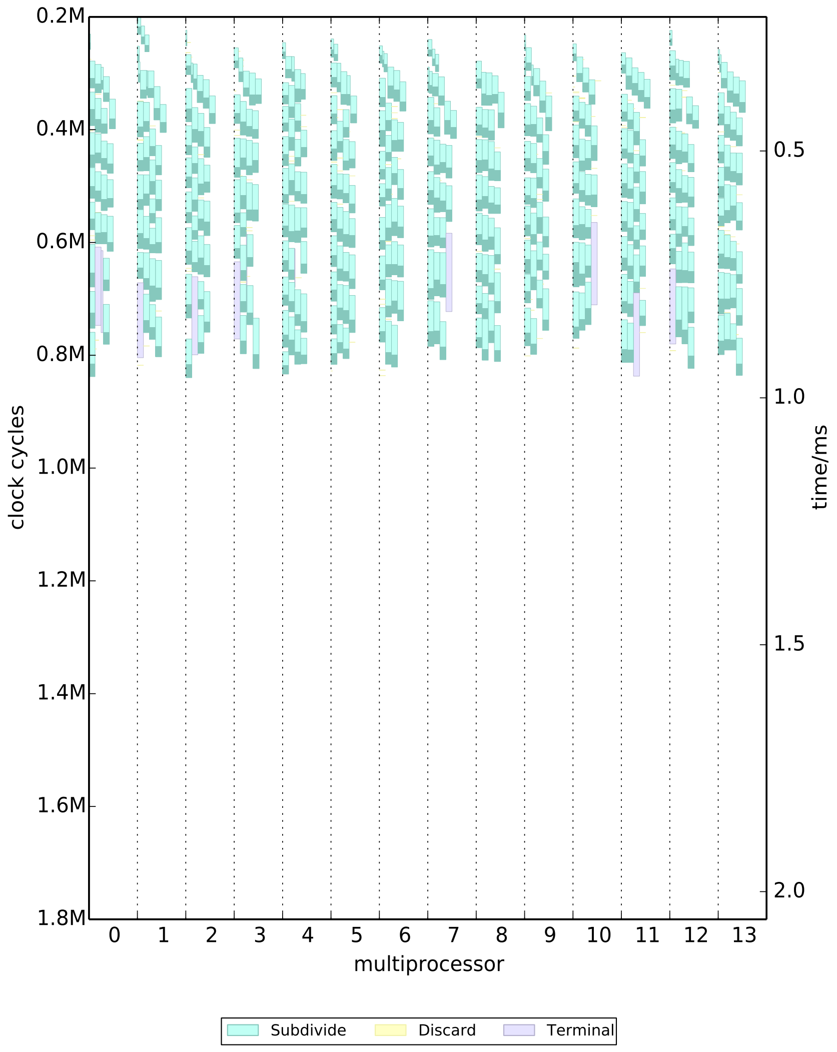


Figure C.20.: *Multi Sierpinski Test Case Configuration: IRD PMK NIR*

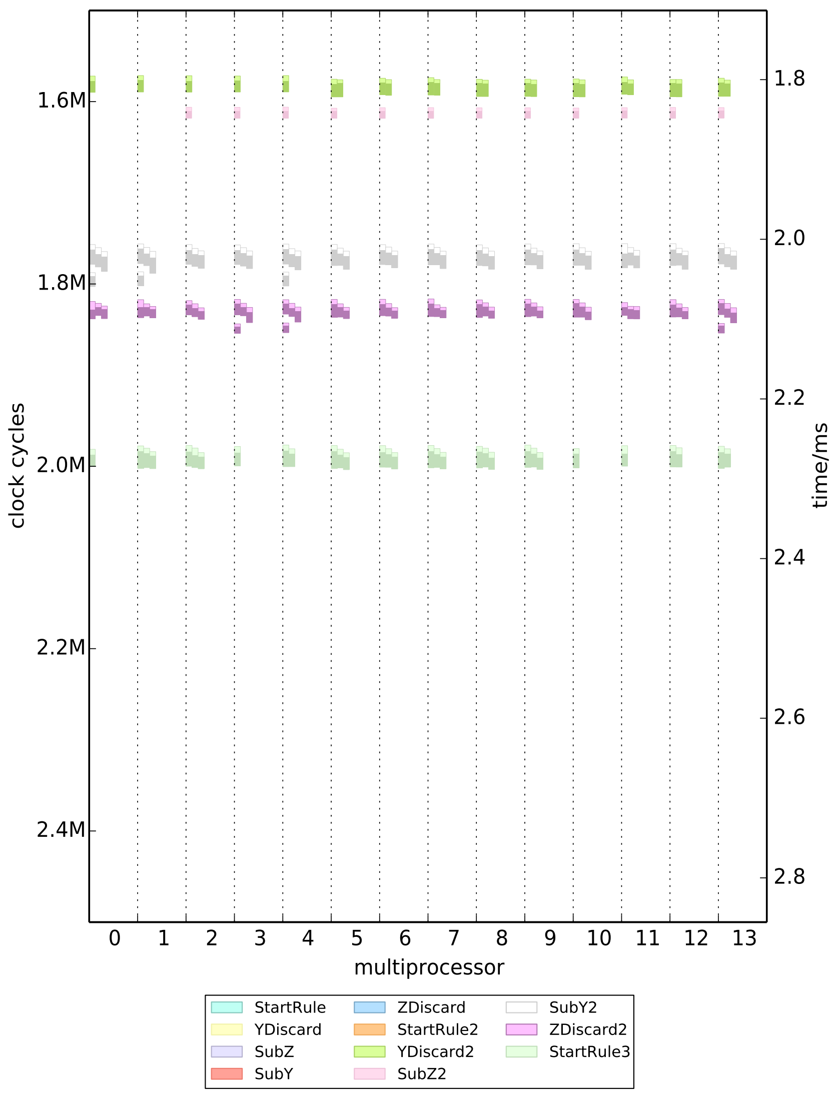


Figure C.21.: *Multi Sierpinski Test Case Configuration: PRD KLS HIR*

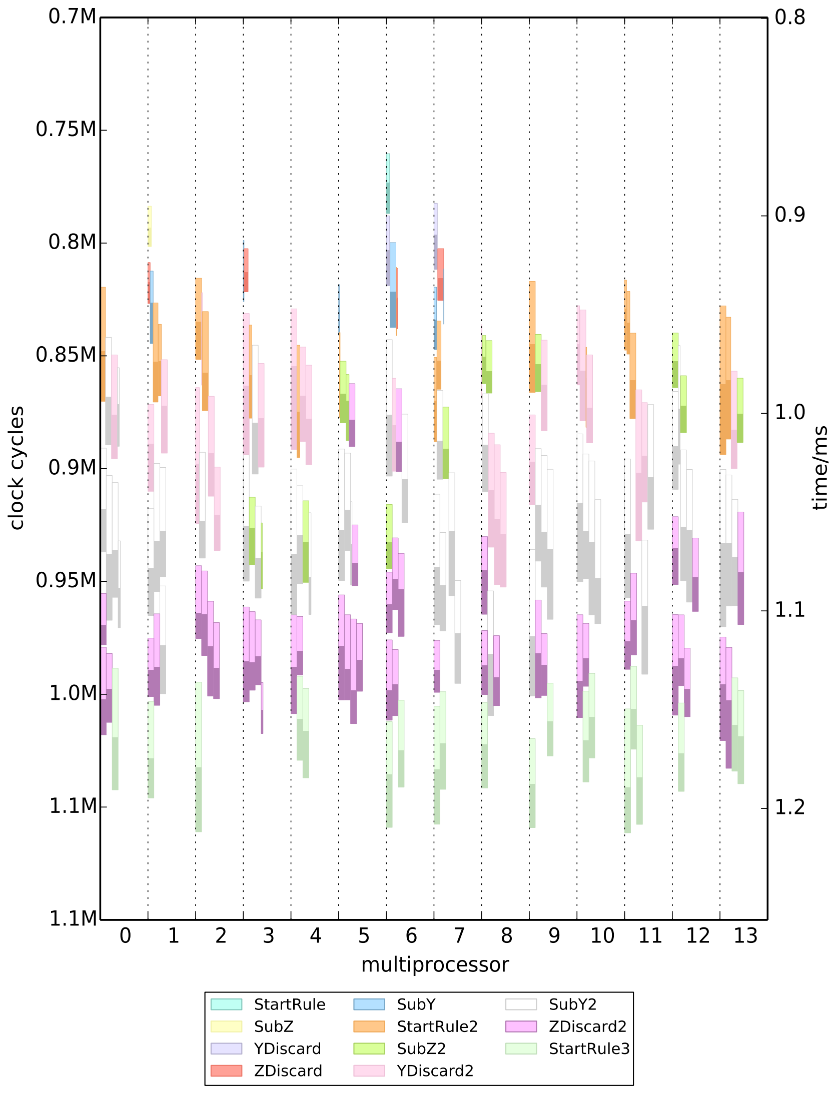


Figure C.22.: *Multi Sierpinski Test Case Configuration: PRD PMK HIR*

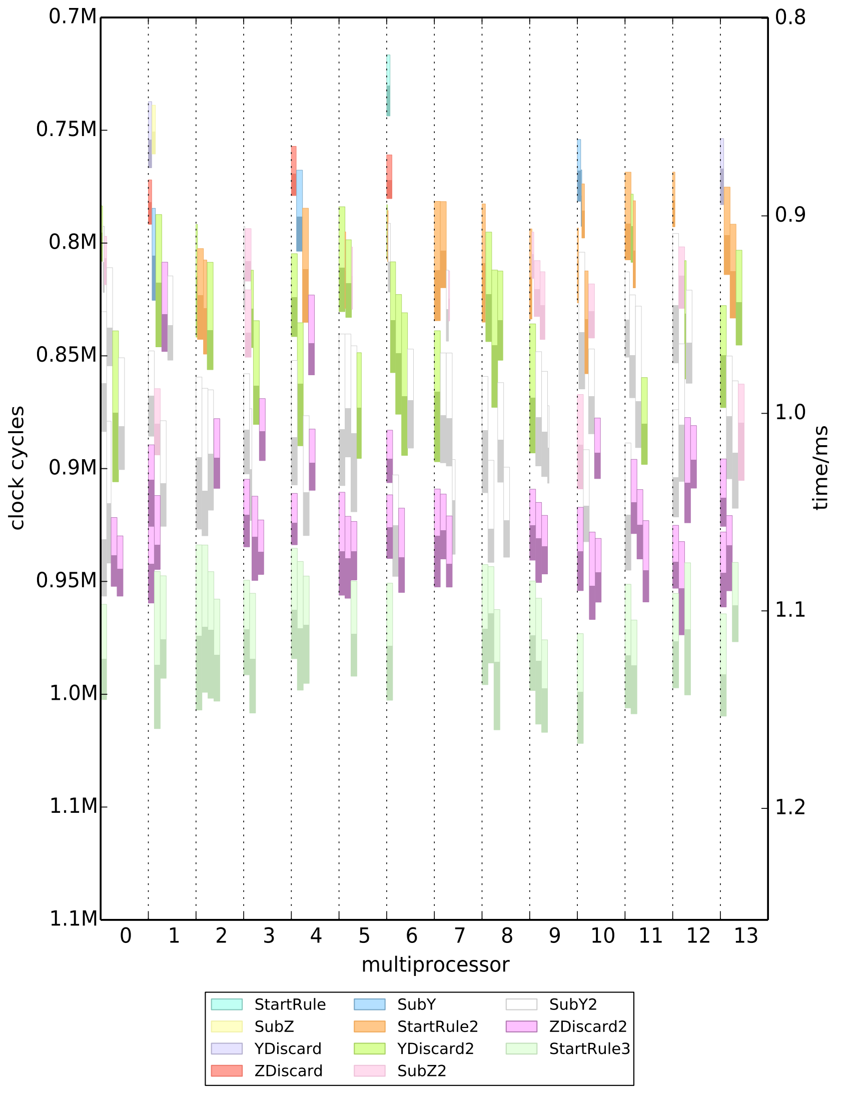


Figure C.23.: *Multi Sierpinski Test Case Configuration: PRD PMK NIR*

C.4. Single Sierpinski Test Case

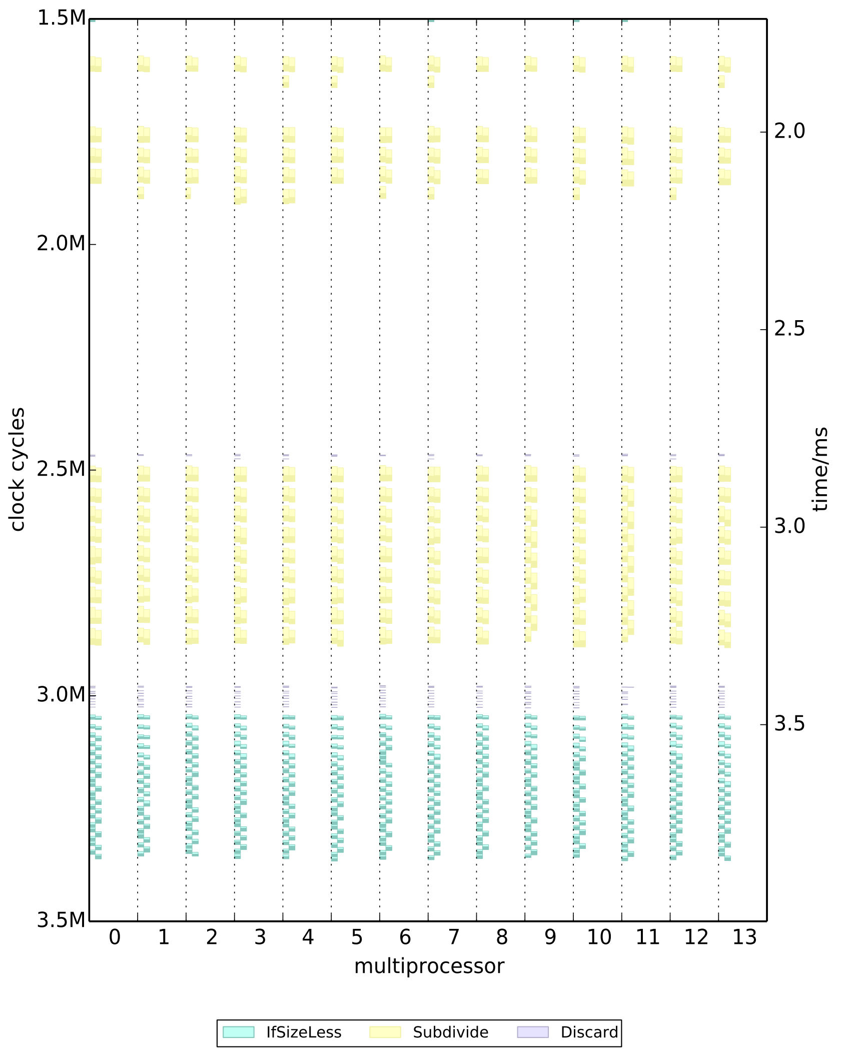


Figure C.24.: *Single Sierpinski Test Case Configuration: IRD KLS HIR*

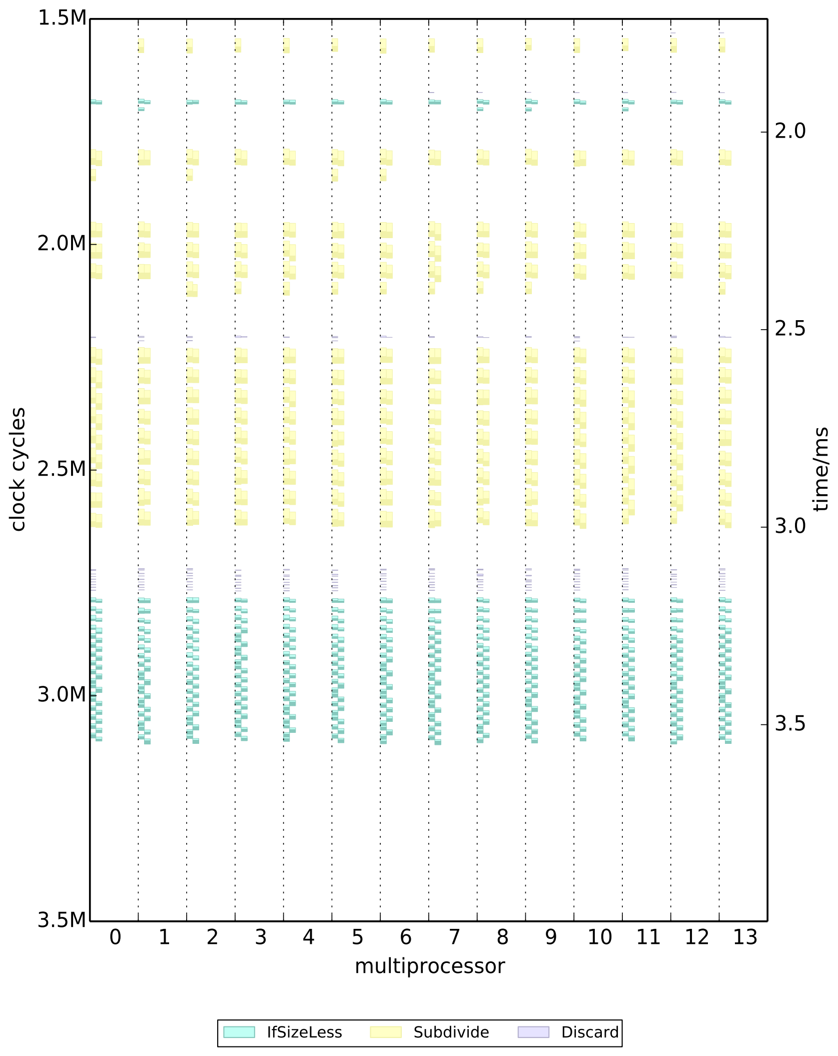


Figure C.25.: *Single Sierpinski Test Case Configuration: IRD KLS NIR*

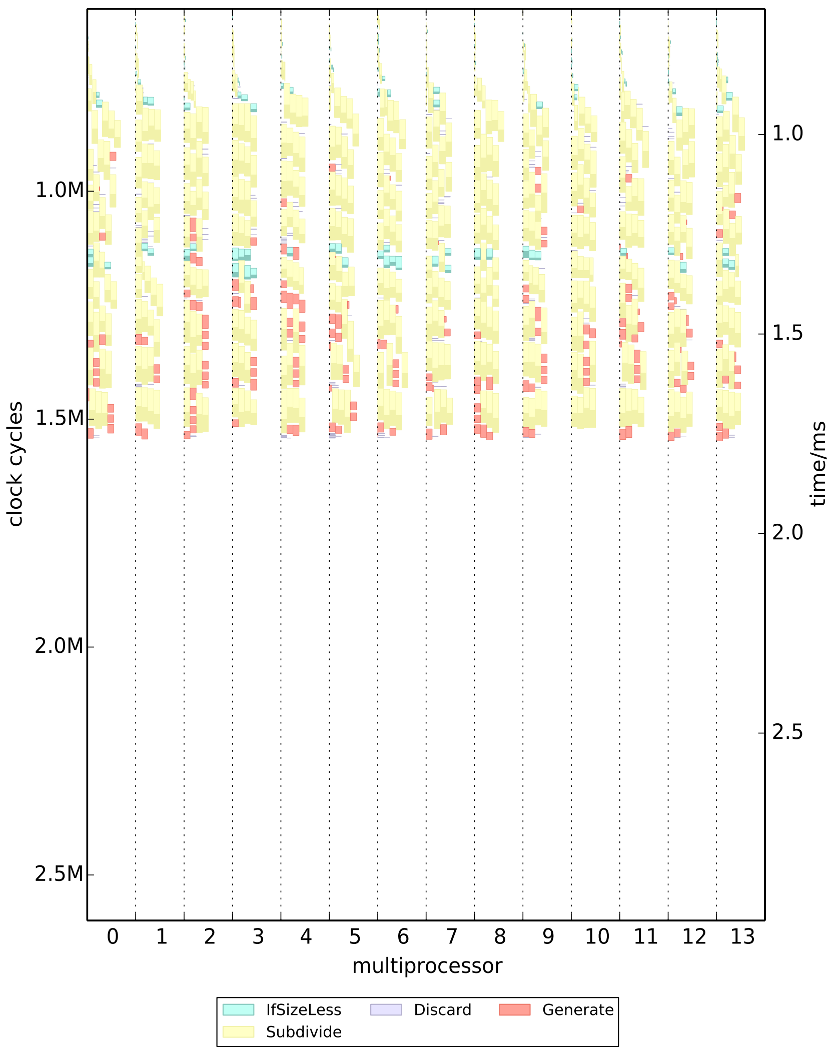


Figure C.26.: *Single Sierpinski Test Case Configuration: IRD PMK HIR*

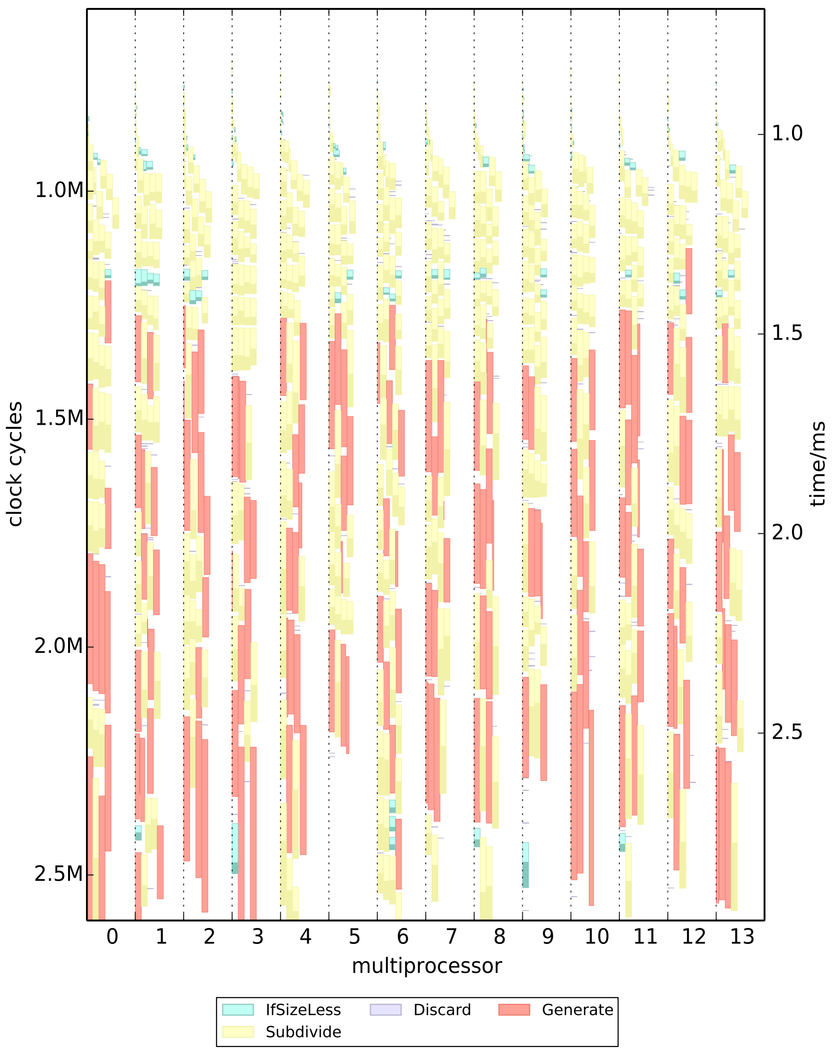


Figure C.27.: *Single Sierpinski Test Case Configuration: IRD PMK NIR*

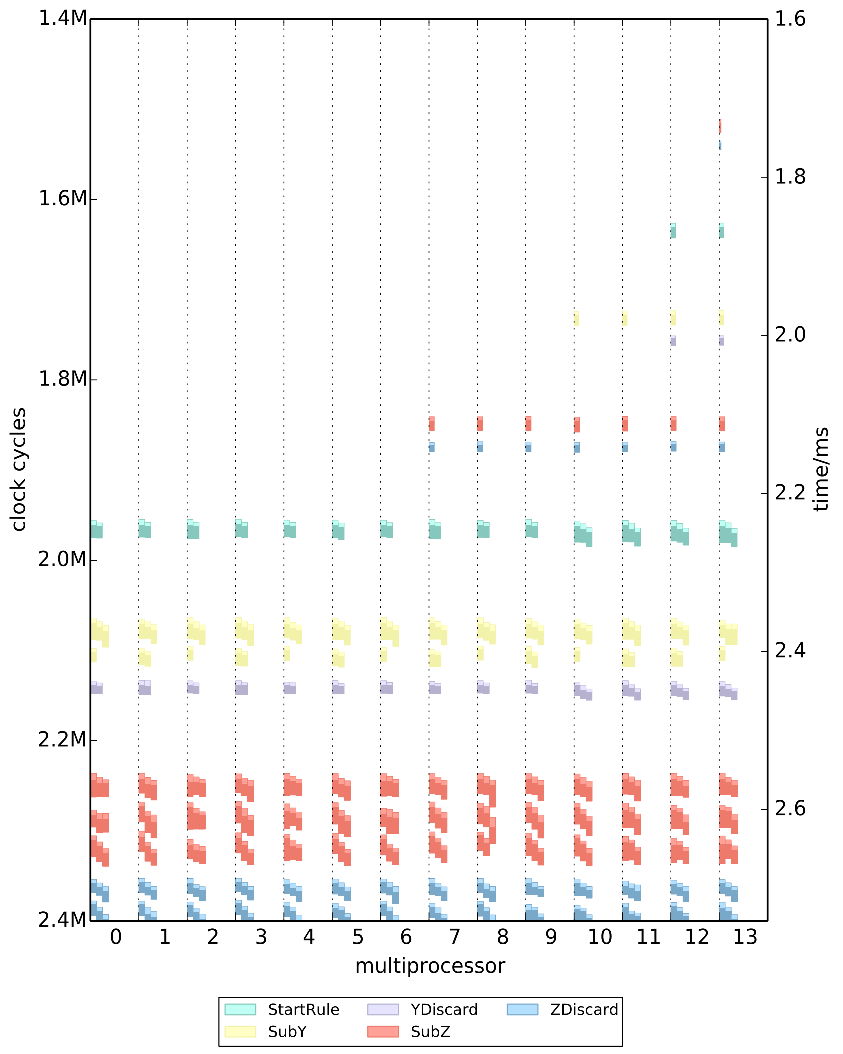


Figure C.28.: *Single Sierpinski Test Case Configuration: PRD KLS HIR*

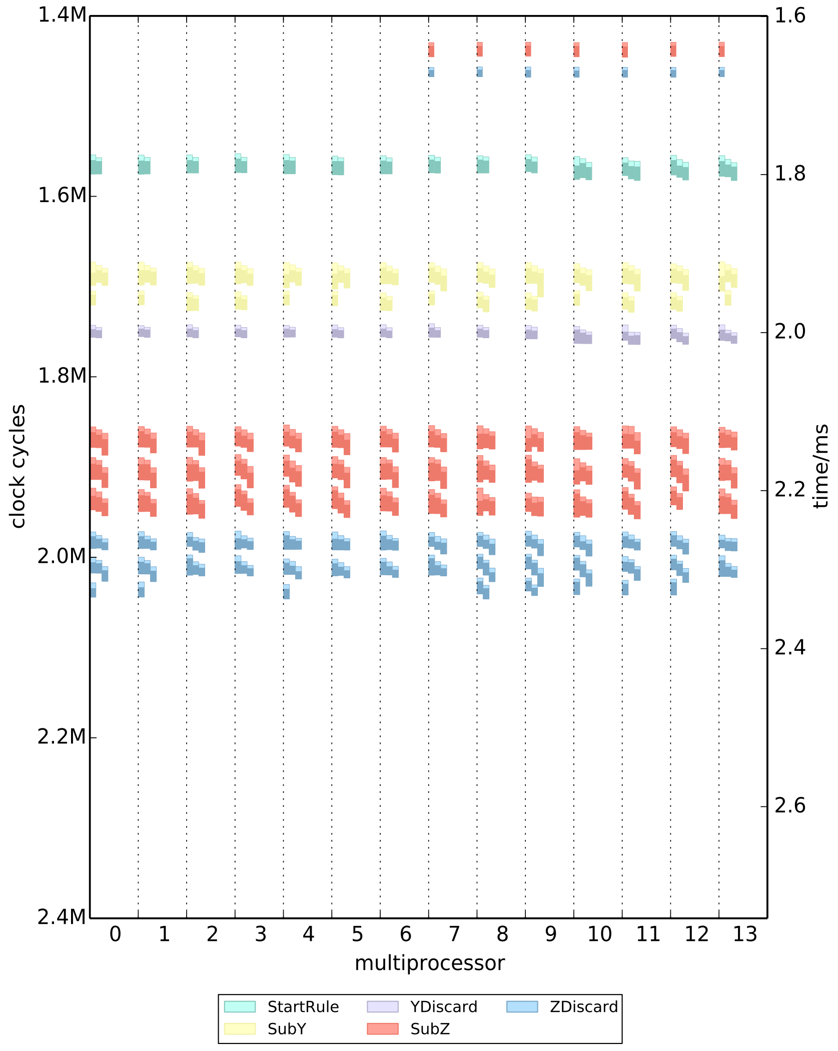


Figure C.29.: *Single Sierpinski Test Case Configuration: PRD KLS NIR*

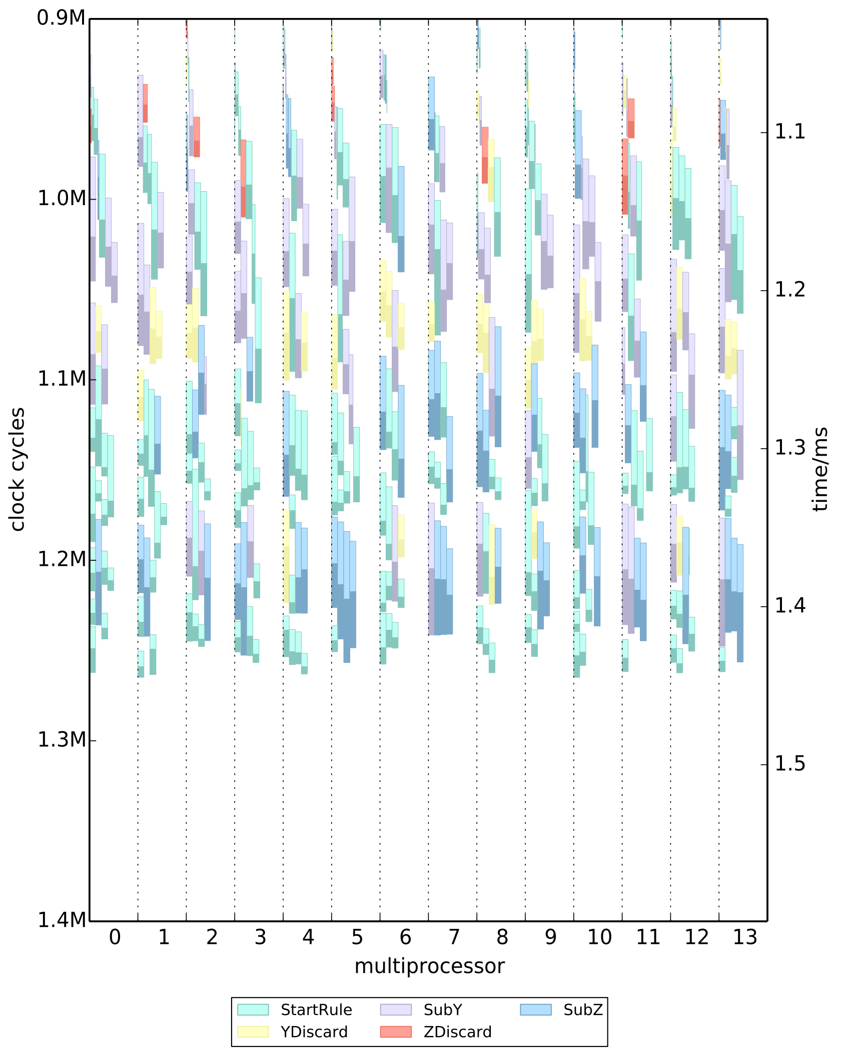


Figure C.30.: *Single Sierpinski Test Case Configuration: PRD PMK HIR*

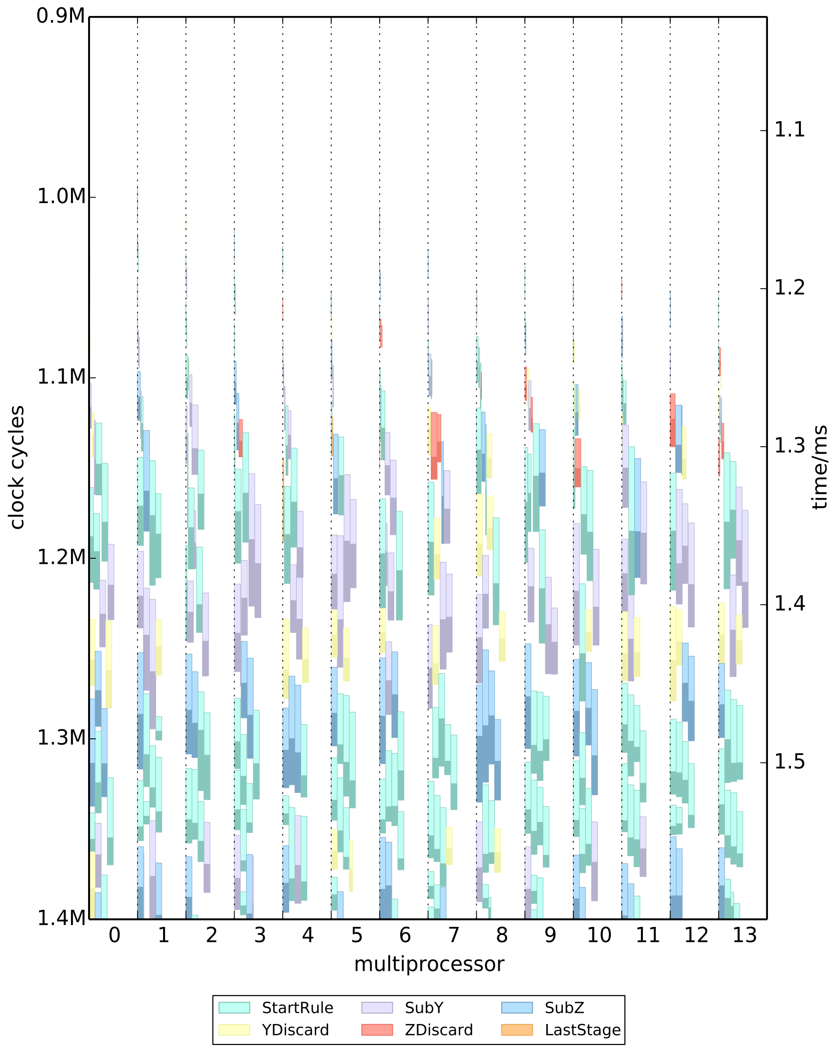


Figure C.31.: *Single Sierpinski Test Case Configuration: PRD PMK NIR*

Bibliography

Aila, Timo et al. (2009). “Understanding the Efficiency of Ray Traversal on GPUs.” In: *Proc. High Performance Graphics*. ACM, pp. 145–149 (cit. on pp. 12, 16, 29).

Cederman, Daniel et al. (2008). “On dynamic load balancing on graphics processors.” In: *Proc. Graphics Hardware*. Sarajevo, Bosnia and Herzegovina, pp. 57–64. ISBN: 978-3-905674-09-5. URL: http://dl.acm.org/citation.cfm?id=1413957.1413967 (cit. on p. 13).

Chatterjee, Sanjay et al. (2011). “Dynamic Task Parallelism with a GPU Work-Stealing Runtime System.” In: *Proc. Languages and Compilers for Parallel Computing* (cit. on p. 13).

Chen, Long et al. (2010). “Dynamic load balancing on single- and multi-GPU systems.” In: *IEEE Parallel Distributed Processing*. DOI: 10.1109/IPDPS.2010.5470413 (cit. on p. 13).

Fung, Wilson W. L. et al. (2007). “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow.” In: *Proc. IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. IEEE, pp. 407–420 (cit. on p. 12).

Gupta, Kshitij et al. (2012). “A Study of Persistent Threads Style GPU Programming for GPGPU Workloads.” In: *Innovative Parallel Computing*. San Jose, CA, p. 14 (cit. on p. 12).

Hargreaves, Shawn (2005). “Generating shaders from HLSL fragments.” In: *ShaderX3: Advanced rendering with DirectX and OpenGL* (cit. on p. 12).

Havemann, S (2005). “Generative Mesh Modeling.” PhD Thesis. TU Braunschweig (cit. on p. 9).

Krecklau, Lars et al. (2011). “Generalized Use of Non-Terminal Symbols for Procedural Modeling.” In: 29 (8), pp. 2291–2303 (cit. on p. 9).

Lacz, P. et al. (2004). “Procedural Geometry Synthesis on the GPU.” In: *Workshop on General Purpose Computing on Graphics Processors*, pp. 23–23 (cit. on p. 10).

Laine, Samuli et al. (2013). “Megakernels Considered Harmful: Wavefront Path Tracing on GPUs.” In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG ’13. Anaheim, California: ACM, pp. 137–143. ISBN: 978-1-4503-2135-8. DOI: 10.1145/2492045.2492060. URL: http://doi.acm.org/10.1145/2492045.2492060 (cit. on p. 27).

Lefebvre, Sylvain et al. (2010). “By-example synthesis of architectural textures.” In: *ACM Trans. Graph.* 29 (4), A84 (cit. on p. 9).

Lin, Jinjie et al. (2011). “Structure-preserving retargeting of irregular 3D architecture.” In: *ACM Trans. Graph.* 30.6, A183 (cit. on p. 9).

Lipp, Markus et al. (2010). “Parallel Generation of Multiple L-systems.” In: 34.5 (cit. on pp. 10, 27).

Luo, Lijuan et al. (2010). “An effective GPU implementation of breadth-first search.” In: *Proc. Design Automation Conference*. Anaheim, California: ACM, pp. 52–55. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837289. URL: http://doi.acm.org/10.1145/1837274.1837289 (cit. on p. 13).

Magdics, M. (2009). “Real-time Generation of L-system Scene Models for Rendering and Interaction.” In: *Spring Conf. on Computer Graphics*. Budmerice, Slovakia: Comenius Univ., pp. 77–84 (cit. on p. 10).

Marvie, Jean-Eudes et al. (2012). “GPU Shape Grammars.” In: 31.7-1, pp. 2087–2095 (cit. on pp. 5, 10).

Merrell, P. et al. (2011). “Model Synthesis: A General Procedural Modeling Algorithm.” In: *Visualization and Computer Graphics, IEEE Trans.* 17.6, pp. 715–728 (cit. on p. 9).

Müller, Pascal et al. (2006). “Procedural Modeling of Buildings.” In: 25.3, pp. 614–623. DOI: 10.1145/1179352.1141931 (cit. on pp. 5, 11).

NVIDIA (2011). *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA (cit. on p. 12).

NVIDIA (2012). *CUDA Dynamic Parallelism Programming Guide* (cit. on p. 12).

Parker, Steven G. et al. (2010). “OptiX: a general purpose ray tracing engine.” In: *ACM TOG* 29.4(66). ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: http://doi.acm.org/10.1145/1778765.1778803 (cit. on p. 13).

Prusinkiewicz, Przemyslaw et al. (1990). *The Algorithmic Beauty of Plants*. New York. ISBN: 0-387-97297-8 (cit. on p. 5).

Steinberger, Markus, Bernhard Kainz, et al. (2012). “Softshell: Dynamic Scheduling on GPUs.” In: *ACM Trans. Graph.* 31.6, A161. DOI: 10.1145/2366145.2366180 (cit. on pp. 11, 13, 29).

Steinberger, Markus, Michael Kenzel, Bernhard Kainz, Jörg Müller, et al. (2014). "Parallel Generation of Architecture on the GPU." In: 33 (cit. on pp. 1, 2, 16).

Steinberger, Markus, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg (2012). "ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU." In: *Proceedings of Innovative Parallel Computing (InPar'12)* (cit. on p. 1).

Steinberger, Markus, Michael Kenzel, Bernhard Kainz, Peter Wonka, et al. (2014). "On-the-fly Generation and Rendering of Infinite Cities on the GPU." In: 33 (cit. on pp. 11, 77, 78).

Steinberger, Markus, Michael Kenzel, and Dieter Schmalstieg (to appear). "The Versatile Megakernel." In: (cit. on p. 12).

Stiny, G. (1975). *Pictorial and Formal Aspects of Shape and Shape Grammars*. Basel: Birkhauser Verlag (cit. on p. 5).

Stiny, G. (1982). "Spatial Relations and Grammars." In: *Environment and Planning B* 9, pp. 313–314 (cit. on p. 5).

Stiny, George (1980). "Introduction to shape and shape grammars." In: *Environment and planning B* 7.3, pp. 343–351 (cit. on p. 6).

Stuart, Jeff A. et al. (2009). "Message passing on data-parallel architectures." In: *Proc. Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161065. URL: http://dx.doi.org/10.1109/IPDPS.2009.5161065 (cit. on p. 13).

Tanenbaum, Andrew S. (2007). *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press. ISBN: 9780136006633 (cit. on p. 11).

Tzeng, Stanley et al. (2010). “Task management for irregular-parallel workloads on the GPU.” In: *Proc. High Performance Graphics*. HPG ’10. Saarbrucken, Germany: Eurographics Association, pp. 29–37 (cit. on p. 13).

Wonka, Peter et al. (2003). “Instant Architecture.” In: *ACM Trans. Graph.* 22.3, pp. 669–677. DOI: 10.1145/1201775.882324 (cit. on pp. 5–7).

Xiao, Shucai et al. (2010). “Inter-block GPU communication via fast barrier synchronization.” In: *IEEE Parallel Distributed Processing*. DOI: 10.1109/IPDPS.2010.5470477 (cit. on p. 13).

Yang, T. et al. (2007). “A Parallel Algorithm for Binary-Tree-Based String Rewriting in L-system.” In: *Proc. of the Second International Multi-symposiums of Computer and Computational Sciences*, pp. 245–252 (cit. on p. 9).