A Flexible Architecture for Ray Tracing Terrain Heightfields

Steve Dübel[1], Lars Middendorf[2] Christian Haubelt, and Heidrun Schumann

[1] University of Rostock, Institute for Computer Science, D-18059 Rostock, Germany
steve.duebel@uni-rostock.de
[2] University of Rostock, Institute of Microelectronics and Data Technology, D-18119 Rostock, Germany lars.middendorf@uni-rostock.de

Abstract. High-quality interactive rendering of terrain surfaces is a challenging task, which requires compromises between rendering quality, rendering time and available resources. However, current solutions typically provide optimized strategies tailored to particular constraints. In this paper we propose a more scalable approach based on functional programming and introduce a flexible ray tracer for rendering terrain heightfields. This permits the dynamic composition of complex and recursive shaders. In order to exploit the concurrency of the GPU for a large number of dynamically created tasks with inter-dependencies, the functional model is represented as a token stream and is iteratively rewritten via pattern matching on multiple shader cores in parallel. A first prototype demonstrates the feasibility of our approach.

Key words: graphics hardware, terrain rendering, ray tracing

1 Introduction

With today's continuously growing amount of data and increased demand of quality, rendering complex terrain surfaces is a difficult task. Current heightfields can consist of hundreds of megabytes of raw data. To render them efficiently on current hardware, the heightfield needs to be triangulated, and appropriate level-of-details have to be defined. This results in data structures that easily increase the volume of raw data by an order of magnitude.

To considerably decrease this high memory consumption that in many scenario exceeds hardware capability, the heightfields can alternatively be rendered through ray tracing. In doing so, surface details are generated on-the-fly. Moreover, ray tracing allows for global illumination effects that significantly improve image quality. To enhance performance, ray tracing solutions use auxiliary data structures, but that increases memory consumption [1], or they compute approximations that decrease quality on the down side [2].

However, guaranteeing interactive frame rates requires an appropriate hardware support. This is mainly achieved by using multiple parallel processing units, e.g. clusters or many-core-systems [3, 4]. Common GPUs, above all, have been utilized for ray tracing. Tracing millions of individual rays in parallel by thousands of cores increases the performance of ray tracers significantly.

Since close interrelations between rendering quality, rendering time and available resources do exist, rendering approaches have to take these dependencies into account. In this way, changing requirements can be addressed. For example, quality can be prioritized before performance or vice versa: For instance, fully illuminated objects in the front need to be rendered in high quality, while objects in the dark or far away can be rendered with less effort to decrease rendering time.

Implementing a more flexible ray tracer comes along with problems for both options; GPU-based and CPU-based solutions. The recursive nature of ray tracing and the desired scalability of our approach do not fit well with the pipeline-based programming model of the GPU. Optix [5], a powerful and easy-to-use, general purpose ray tracing engine for the GPU, grant a better flexibility, but does not allow the user to fully customizing acceleration structures, buffer usage and task scheduling. On the other hand, the CPU permits high flexibility, but lacks high data parallelism. Thus, the performance of such CPU-based approaches is hardly sufficient. Hybrid solutions that run on CPU and GPU mostly suffer from the bottleneck of efficient communication. Hence, a novel approach is required.

In this paper, we propose a new rendering architecture for terrain visualization. The terrain is modeled as a mathematical function $f : \mathbb{R}^2 \to \mathbb{R} \times \mathbb{R}^3$ with $(x, y) \mapsto (z, (r, g, b))$ which provides an elevation ($z \in \mathbb{R}$) and a color value $(r, g, b) \in \mathbb{R}^3$ for each pair $(x, y) \in \mathbb{R}^2$ of terrain coordinates. The rendering architecture consists of three stages (Fig. 1).

The first stage (top of Fig. 1) is a flexible ray tracer that is made of two parts: *i*) a fixed, high efficient ray tracer kernel for terrain heightfields and *ii*) a modular extension unit. The ray tracing kernel utilizes beam tracing and fast intersection techniques to achieve real time frame rates. The output is a simple, colored image. The flexible, modular extension unit provides a set of enhanced rendering operators, which can be activated on demand to improve the image quality. Here, the generation of surface details (interpolation, microstructures, antialiasing) and advanced shading can be dynamically complemented by a set of appropriate modules. In this way, a trade-off between render time, render quality and available resources can be achieved.

The second stage provides a functional model (center of Fig. 1), defined as a network of executable tasks, including parallel or recursive parts. The basic ray tracing module, as well as the enhanced rendering operators are described as individual tasks, i.e. as nodes of a functional model. The functional model provides benefits like dynamic composition and recursion. However, it does not fit well into the data parallel execution model of current graphics hardware.

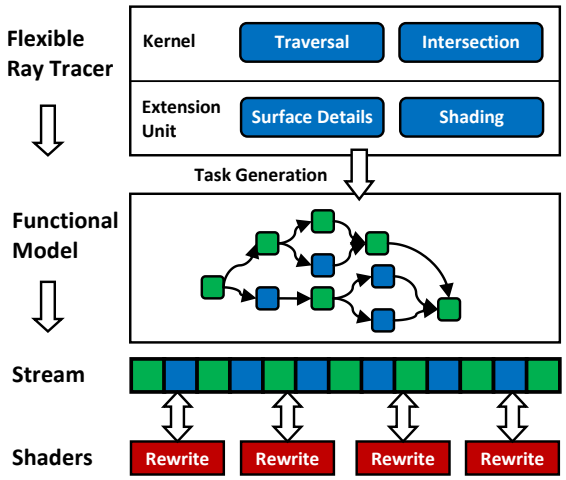


Fig. 1. Rendering architecture for flexible terrain ray-tracing.

Hence, with the *third stage* (bottom of Fig. 1) and in order to build a more flexible ray tracer, we propose a novel approach for dynamic task management of functional programs on the GPU. For this purpose, the functional model is encoded as a token stream and iteratively rewritten by several shader cores in parallel. In particular, invocations are represented by specific patterns in the stream that are replaced by the result of the corresponding functions. Therefore, all types of indirect and recursive functions can be evaluated in parallel.

In summary, the main contribution of this paper consists of a novel execution model for scheduling dynamic workload on the GPU and its application to the problem of terrain visualization. The most significant difference to related approaches like [6], [7], and [8], is the usage of pattern matching to resolve dependencies between tasks through local rewriting operations on the stream.
The remainder of this work is structured as follows. In section 2 we present related work for interactive ray tracing and dynamic task scheduling. Next, the concept of the flexible terrain ray tracer will be introduced in section 3. The formal model and the parallel implementation for scheduling functional programs is subject of section 4 and 5, before we present a first prototype and respective results in section 6. Finally, section 7 concludes this report by a summary and hints for future research directions.

2 Related Work

Before going into the detail of our approach, we will briefly present related, state-of-the-art concepts of interactive terrain ray tracing on the one hand, and dynamic task management, on the other.

2.1 Interactive Ray Tracing

[9] describe three aspects to support interactive ray tracing: accelerating techniques, approximation and hardware.

Accelerating techniques To achieve acceptable frame rates, auxiliary data structures are necessary. Especially Bounding Volume Hierachies (BVH), kd-trees and grids [1] are used to accelerate the ray traversal and reduce intersection tests. BVH trees can be updated very fast, but do not adapt to the geometry as good as kd-trees do. In contrast to both, grids do not provide a hierarchy and no adaptability, but a fixed, uniform subdivision of the space [10]. However, this structure fits well to heightfield data. Our approach is based on [11], who extend the grid by a hierarchy, creating a so-called maximum mipmap data structure. This is similar to a quad-tree and allows for interactive ray casting on heightfields.

Other techniques accelerate ray tracing by exploiting ray coherence. That means, a number of rays are simultaneously traversed as ray packets [12] or beams [13]. Our approach utilizes a beam tracing based fast start, where a chunk of rays are traversed through the heightfield as one beam to calculate a map of starting points for the actual ray tracing (cf. Section 3.1).

Approximations Typically, intersection points and shading information can be determined by proper approximations. For instance, [14] proposed an efficient surface-ray-intersection algorithm for heightfields that is based on combination of uniform and binary search instead of an exact calculation. This might cause visible artifacts, but this problem is later solved through relaxed cone stepping [15, 16].

Moreover, global illumination can be approximated, since in outdoor scenes reflection and transparency do hardly contribute to shading. Such techniques were originally used to introduce global illumination effects to rasterization-based terrain rendering. A very simple approximation is ambient occlusion [17] that mainly estimates the locally limited distribution of ambient light by sampling the hemisphere through ray tracing. The average direction of unoccluded samples can additionally be used to include incident radiance e.g. through a look-up-texture (environment map). [18] extend this concept by additionally defining a cone, which aperture angle and alignment are based on the unoccluded samples and limits the incident light, considering a single light source. This technique is well-suited for outdoor terrain scenes, where the sun is the only light source, and considerably decreases render time. Our approach also supports different approximated illumination.

Hardware Numerous customized hardware solutions are developed [19, 20] to provide capable ray tracers. [21, 22] propose ray tracing approaches that are completely realized on the GPU. On the other hand, specialized hardware solutions address particular tasks of ray tracing, such as traversal and intersection, e.g. [20]. However, none of these architectures support flexible scaling between quality and performance or provide dynamic scheduling of tasks.

2.2 Dynamic Task Scheduling

There already exist several concepts for dynamic task scheduling for graphics processing.

Software Implementations Although the hardware architecture of modern GPUs is optimized towards data parallelism [23], dynamic scheduling of heterogeneous tasks can be implemented in software [6] and utilize work stealing for load-balancing [7]. Usually, a single kernel runs an infinite loop that consumes and processes tasks from queues in local or global memory [8]. However, if the tasks are selected via dynamic branching, irregular workloads can interfere with the single-instruction multiple-thread (SIMT) execution model of modern GPUs [23].

Our concept is compatible to these existing approaches, but additionally performs a pattern matching step to determine the readiness of a task. In particular, the relative execution order is controlled by data dependencies, which permit to efficiently embed complex task hierarchies into the stream, while both the creation and the completion of tasks are light-weight and local operations. [24]

Hardware Architecture Similar to our technique, the graphics processor proposed by [25] also stores the stages and the topology of a generic rendering pipeline as a stream. However, the scalability of the presented hardware implementation remains limited because the stream is decoded and reassembled sequentially. For comparison, our scheduling algorithm performs an out-of-order rewriting of the stream and keeps the tokens in the fast shared memory of a multiprocessor.

Programming Languages In addition, purely functional languages like *NOVA* were proposed for GPU programming [26] due to their applicability for automatic optimization techniques [27]. Predecessors like Vertigo [28] or Renaissance [29] are based on Haskell and allow composing complex objects from parametric surfaces and geometric operators in the shader. Similarly, the language *Spark* [30] introduces aspect-oriented shaders to permit a compact and modular description in the form of classes. While these approaches statically translate the source language into shaders, we introduce a runtime environment for the GPU, which retains the flexibility of functional programming at the expense of dynamic scheduling.

3 Flexible Terrain Ray Tracing

Our flexible rendering approach consists of two major components: A fixed ray tracing kernel to traverse the rays through a discrete terrain heightfield and a modular extension unit to control surface generation and shading. While the ray tracing kernel generates a simple, colored image, the extensions support flexibility to balance between rendering quality, rendering time and memory consumption. Both components are described in the following.

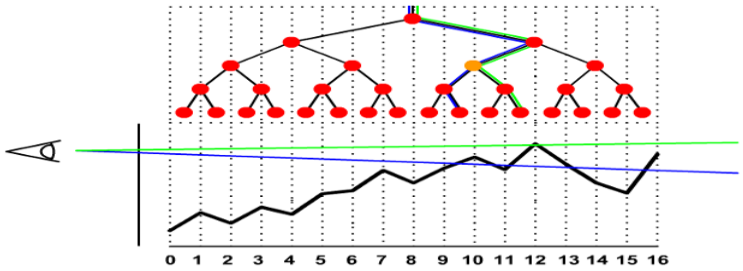
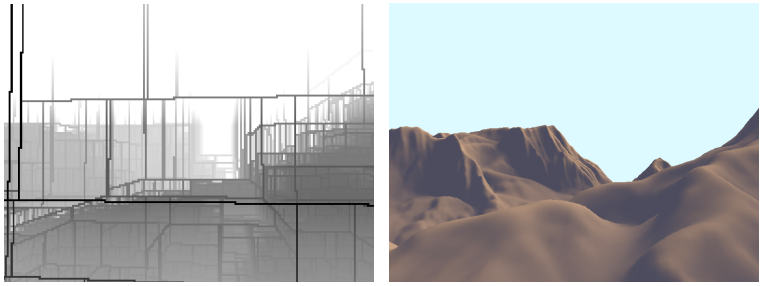


Fig. 2. Ray traversal in 2D for beam tracing. As long as the rays on the corners follow the same path through the tree, the whole beam visits the same node (orange). Otherwise, the beam needs to be subdivided.



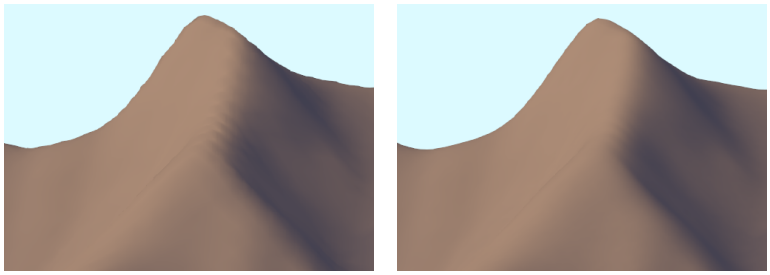
(a) (b)

Fig. 3. Illustration of our beam tracing based fast start. (a) The result of beam tracing is the starting position of individual ray traversal encoded as a depth map. The grid structure reflects the split of them at the BV. (b) The final rendered image.

3.1 Ray Tracing Kernel

To ensure a high performance, we apply and combine sophisticated techniques from literature for both ray traversal and intersection tests. We utilize a grid-based bounding volume hierarchy, in particular the maximum mipmaps (MM) introduced by [11]. The maximum mipmap is structured as a quad tree that stores the maximum of all underlying height values at each node. The root node spans the whole heightfield, while a leaf node stores the maximum of four actual height values of the field. This structure can be constructed very fast. Since spatial information is stored implicitly, the increased memory footprint is very low ($\approx 33\%$).

To decrease rendering time, we apply a beam tracing based fast start. The beams are pyramids with a base area of e.g. 8x8 pixels. The rays defined by the corners of the base area are traversed through the MM-tree. Figure 2 shows the traversal through the tree in 2D. Only if the four rays at the corner take the same path, it is guaranteed, that all rays within the beam will also take this specific path. If the rays visit different nodes or hit different sides of the



(a) (b)

Fig. 4. (a) Bilinear interpolation introduces artifacts, such as discontinuation within the silhouette and at shadow edges, whereas bicubic interpolation (b) provides smoother transitions.

bounding volume, beam tracing needs to be replaced by traversing individual rays. The result of the beam tracing is depicted in Figure 3.

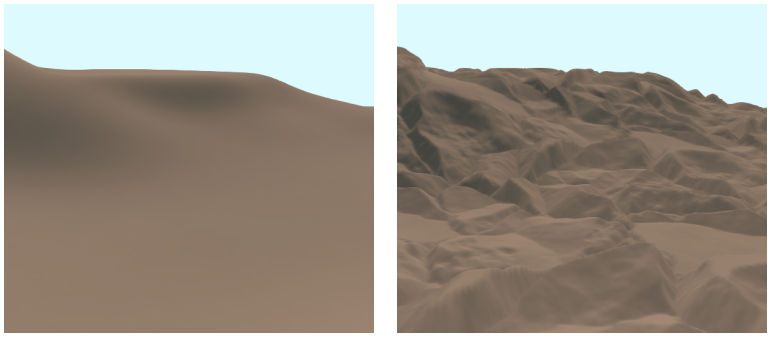
An exactly calculation of the intersection points between the rays and the surface patch of the heightfield is time consuming. Therefore, we use uniform and binary search [14] to get an approximate intersection point. First the ray is subdivided into uniform line segments. The uniform search determines that line segment which intersects the patch of the heightfield. Second the binary search computes the approximated intersection point within this segment. An advantage of this method is the abstraction from the real structure of the patch. The intersection test is based only on the height values (z) at given coordinates (x,y). Therefore, the generation of patches themselves can be encapsulated by operators of the modular extension unit.

3.2 Modular Extension Unit

The modular extension unit consists of a set of operators that allow to improve image quality on demand. In this paper, we suggest enhanced operators for surface generation and shading, but further operators can be easily added.

Surface Generation The ray-patch-intersection computed by the ray tracing kernel is based solely on height values. Now the surface patches are generated by enhanced operators. They can be composed to adjust this part of the rendering process.

Interpolation The surface patches of a heightfield are generated through interpolation by a specific operator of the extension unit. Commonly, the patch is bilinear interpolated between four neighbored height values. However, bilinear interpolation can introduce artifacts at the silhouette and shadow edges (Fig. 4(a)). Hence, a different operator can be used to provide a smoother bicubic interpolation. This results in less artifacts (Fig. 4(b)), but also increases rendering time. A midway solution provides an approximated bicubic interpolation. Here the



(a) (b)

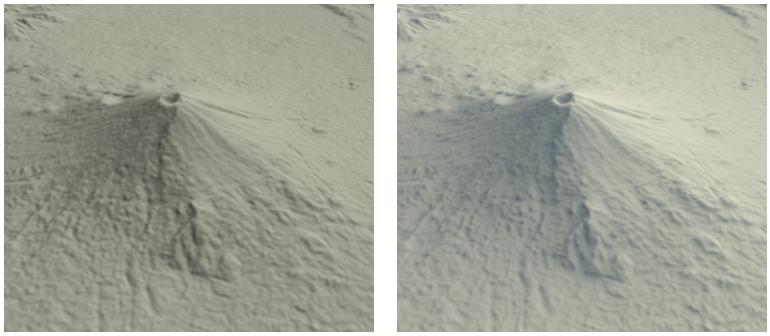
Fig. 5. (a) The smooth interpolated surface of the heightfield lacks fine granular details. To increase realism, the surface can be enriched by microstructures that displace the height values along the z-axis (b).

surface is continuously subdivided by generating points through a bicubic function and is afterwards bilinear interpolated in between. This operator provides both, good quality and good performance.

Microstructures The resolution of heightfields is normally not sufficient to provide fine granular details. Hence microstructures are used to increase image quality. Different operators enrich a base heightfield by extra details. Either noise functions or additional micro-heightfields then describe the displacement along the z-axis (Fig. 5).

Antialiasing If only one ray per pixel is traversed, aliasing artifacts appear in the distance where the surface is under-sampled. Therefore, enhanced operators support antialiasing. On the one hand, an average mipmap allows for trilinear interpolation of the height values depending on the distance. This increases the memory footprint. Alternatively, multiple rays can be traversed per pixel. In this case, enhanced operators invoke supplementary ray tracing for surfaces in the distance. This increases rendering time.

Shading The ray tracing kernel assigns a material color to each visible point. To improve shading quality, the modular extension unit provides enhanced operators. A full recursive ray tracing is the most time consuming method. Simple texturing, e.g. with satellite images, however, may lead to low quality. Further, sophisticated illumination models, tailored to outdoor scenes, are supported. Especially Ambient Occlusion (AO) and Ambient Aperture Lighting (AAL) (cf. Section 2.1) apply well to terrain rendering. While AO is the fastest method, since only the quantity of self-occlusion is measured. AAL is more accurate and produces softer shadows, since the color and angle of incident light and even the diffuse scattered light of the sky are considered (Fig. 6). The higher quality leads to a higher rendering time. However, both techniques are based on preprocessing



(a) (b)

Fig. 6. Using (a) Ambient occlusion or (b) Ambient Aperture Lighting results in different levels of image quality. (Fuji region, elevation data source: ASTER GDEM, a product of METI and NASA.)

to reduce computation time during rendering. But this again increases memory consumption.

The described operators of the extension unit support the configuration of a ray tracer to different constrains. However, the introduced set of operators can easily be extended.

4 Functional Model

The functional model forms the interface between the flexible ray tracer and the dynamic execution model on the GPU. It is composed of individual tasks. Each task represents an operator of the ray tracing stage. The tasks are connected with regard to the processing flow.

They compose a network of sequential, parallel or recursive tasks. The basic configuration consists of a combination of ray traversal, intersection test and additional enhanced operators of the extension unit. The choice of enhanced operators determine the network topology. Recursive ray tracing, for instance, maps to a recursive network structure of the tasks, while the traversal of individual rays in parallel maps to a parallel structure.

To support the decision which enhanced operators should be used, we describe presets that either focus on rendering time, rendering quality or memory consumption. When rendering time is prioritized, simple bilinear interpolation and simple shading, for instance texturing, will be used. Whereas a quality-based configuration utilizes bicubic interpolation, adds details through microstructures and reduces artifacts in the distance through antialiasing. Moreover, high-quality shading can be activated, e.g. full recursive ray tracing. Memory-based set-ups, again, will omit precomputed shading operators and additional micro-heightfields to reduce memory consumption. When antialiasing is used, subsampling will be favored over additional average mipmaps.

The presets reflects a primary focus and define the primary functional model. However, varying data complexity and/or further constrains, such as ensuring minimal frame rates, might require adaptions of the functional model. Supporting dynamic insertion, removal or replacement of tasks on parallel hardware is a challenging issue. In the next section, we introduce a new task scheduling architecture that solves this problem.

5 Dynamic Task Scheduling

In this section, we present a formal model and a parallel implementation for scheduling the functional model (center of Fig. 1) on the GPU through parallel rewriting operations (bottom of Fig. 1). For this purpose, each invocation of a function is described as a task, whose dependencies are encoded into the stream.

5.1 Execution Model

The proposed execution model (Fig. 1) consists of a token stream, storing the current state of the functional program, and a set of rewriting rules, which are iteratively applied to modify the stream. In particular, we assume that the program is given as set of functions $F := \{f_1, \ldots, f_n\}$ and that the stream contains two different types of tokens to distinguish literal values from invocations.

Formally, a stream $s \in S$ can be described as a word from a language S with alphabet $\Sigma := \mathbb{Z} \cup F$, while each function f_i maps a tuple of n_i integers to m_i output tokens $f_i : \mathbb{Z}^{n_i} \to \Sigma^{m_i}$. The rewriting step is specified by a function $\mathit{rewrite} : S \to S$ that replaces the following pattern:

$$\langle a_1, \ldots, a_{n_i}, f_i \rangle \quad \text{with } a_1, \ldots, a_{n_i} \in \mathbb{Z}, f_i \in F$$

by the result of the invocation:

$$\langle r_1, \ldots, r_{m_i} \rangle \quad \text{with } (r_1, \ldots, r_{m_i}) := f_i(a_1, \ldots, a_{n_i})$$

In particular, an invocation pattern takes a list of literal arguments and a reference to the corresponding function $f_i \in F$. If the function token f_i is preceded by at least n_i arguments $(a_1, \ldots, a_{n_i})$, it is evaluated and replaces the original sub-stream. Hence, this scheme is equivalent to the post-order format also used in reverse Polish notations. Most important for a parallel GPU implementation, the rewriting affects only local regions of the stream and can be performed on different segments in parallel.

By starting with an initial stream s_0 and iteratively trying to replace invocations, we can construct a sequence of streams, whose limiting value can be considered as the final result:

$$\begin{aligned} s_{n+1} &:= \mathit{rewrite}(s_n) \\ \mathit{result}(s_0) &:= \lim_{n \to \infty} s_n \end{aligned}$$

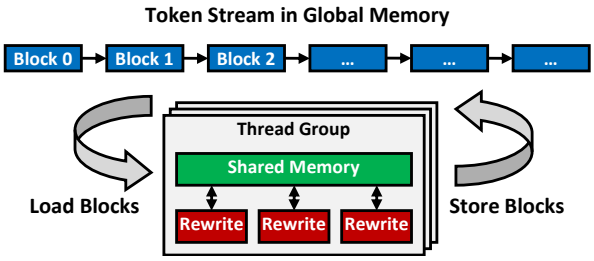


Fig. 7. Global scheduling of the segmented stream.

Due to the iterative rewriting, only at least one pattern must be replaced by the function *rewrite* in order to guarantee the monotony of this sequence. As a result, an implementation is not required to detect every pattern in the stream, so that it can be partitioned more freely for parallel rewriting. The following example illustrates the rewriting sequence for the expression $1 \cdot 2 + 3 \cdot 4$ with $F := \{f_1, f_2\}$, $f_1(x, y) := x \cdot y$ and $f_2(x, y) := x + y$:

$$
\begin{aligned}
s_0 &:= \langle \underbrace{1, 2, f_1}_{f_1(1,2)}, \underbrace{3, 4, f_1}_{f_1(3,4)}, f_2 \rangle \\
s_1 &:= \langle \underbrace{2, 12, f_2}_{f_2(2,12)} \rangle \\
s_2 &:= \langle 14 \rangle
\end{aligned}
$$

In iteration s_0 only the inner multiplications of f_1 can be evaluated in parallel, whereas f_2 waits for the intermediate result to become ready. Eventually, in the rewriting step from s_1 to s_2 the final sum is computed. In addition to literal values ($\mathbb{Z}$), also function tokens ($f_i \in F$) can be emitted to create recursive invocations.

Despite the simplicity of this model, which is entirely based on find-and-replace operations, an efficient GPU implementation has to solve several issues, which are discussed in the next two sections.

5.2 Parallel Implementation

According to the formal definition and the illustration in Fig. 1, the proposed algorithm can be parallelized by letting each core rewrite a different region of the stream. Also important, the partitioning of the stream into regions can be chosen almost arbitrarily. Further, we do not need to consider the contents of the stream because data dependencies are resolved by the pattern matching, and the model does not define an explicit execution order. Instead, control dependencies are also represented by local data dependencies. However, an efficient implementation also has to respect the architecture of modern GPUs, which are optimized for data parallel kernels and therefore require a large number of threads to reach optimal occupancy. In addition, threads are organized into groups, which are

executed on the same processor and communicate via a small but fast shared memory. Since a function pattern creates and deletes a variable number of tokens in the stream, the length of the stream is continuously changing during the rewriting process, so that the data structure must be able to provide random access but also permit the fast insertion and removal of tokens.

As a consequence, the stream is partitioned into blocks of fixed size, which are stored as a linked list in global memory (Fig. 7). In correspondence to the two-level hierarchy of the graphics processor [31], we distinguish between the global scheduling of blocks at the system-level and the local rewriting of individual tokens, which is performed in the shared memory of each thread group. In particular, we utilize the concept of persistent threads, which run an infinite loop executing the following steps:

1. **Load Blocks** Depending on their size, one or two consecutive blocks are fetched from the stream and loaded into the shared memory of the multiprocessor. Due to the coherence of the memory access, the load operations can be coalesced to utilize the available bandwidth.
2. **Local Rewriting** The stream is rewritten locally and the results are stored in the shared memory (see Section 5.3). This process can be optionally repeated several times and requires multiple passes as well as random memory access.
3. **Store Blocks** The resulting tokens are written back into the global stream and up to three additional blocks are allocated. Also, the memory of empty blocks is released if necessary.

Most of these steps can be performed independently by several thread groups in parallel. Hence, especially the local rewriting but also the reading and writing of the stream can benefit from the parallel GPU architecture. However, the selection of a block in the linked list, the allocation, and the release of a block require exclusive access and must be protected by a global lock. The linked list provides the ability for fast insertion and removal of blocks, so that the memory layout of the stream is decoupled from its logical sequence, while the linearity within each block still facilitates coherent access of the global memory.

Formally, a block can be described as a tuple $(addr, next, size, active)$, storing the address of the tokens ($addr$), a pointer to the next block ($next$), the number of tokens in the block ($size$), and a flag ($active$) indicating whether the block is currently rewritten.

Each block has a fixed size in memory, so that the allocation and deallocation can be performed in constant time using a stack of free blocks. However, in order to compensate for varying lengths, the actual number of tokens in the block ($size$) can vary during the rewriting process. Blocks, which are marked as *active*, are currently rewritten by a different thread group and must be ignored. In addition, we have to deal with the two cases of blocks causing *underflow* and *overflow* conditions. An *underflow* is reached if a block is too small, so that no valid pattern can be found, while an *overflow* occurs if the rewritten stream does not fit back into its original block. As a consequence, the system must be able to merge or split consecutive blocks on demand.

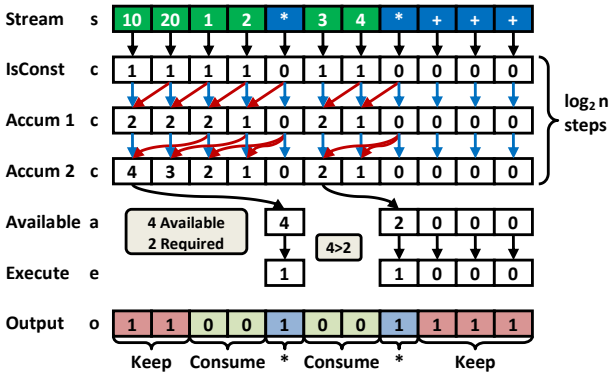


Fig. 8. Parallel decoding of the local stream.

In order to handle underflows, we always try to load two successive blocks as long as both fit into the shared memory. Therefore, small or empty blocks are automatically merged while the stream is rewritten. If the result can be stored back into a single block, the second one is released and removed from the linked list. On the other hand, if it becomes conceivable that the rewritten stream will not fit into the local memory, it is broken into four blocks. Since the next thread group loads at most two of these four blocks, a new overflow is less likely to happen and would subdivide the stream further until the results fit into the shared memory. Hence, the stream of blocks is expanded and reduced by several thread groups in parallel and out-of-order, so that compute-intensive regions of the stream do not delay the rewriting of faster blocks. In the next section, we will discuss the local rewriting in the shared memory.

5.3 Local Rewriting

Unlike the global rewriting process, which is partitioned across several thread groups, the local rewriting of a sub-stream must exploit the data parallelism of a single multiprocessor. Since each function consumes and emits a different number of tokens, a stream must be adjusted by copying it into a new array. In particular, for each of the two token types, we can identify two possible actions:

- **Literal values** are either removed from the stream, if the corresponding function is executed, or they are copied to the next iteration. Hence, a literal always creates one or zero outputs.
- **Function tokens** $f_i \in F$ either produce the specified number of m_i outputs if a sufficient number of literals are available or they are kept on the stream. Thus, a function token is rewritten into one or m_i output tokens.

Since the rewriting in shared memory should employ coherent control flow and data parallelism, it is restructured into three passes:

1. **Decode Stream** The stream is scanned for executable patterns and for each token, the number of outputs is computed according to the four cases above.

2. **Allocate Outputs** Depending on the number of outputs, the new position of each token is calculated using a prefix-sum.
3. **Execute Functions** The functions determined in the 1st step are executed and their results are stored at the positions computed in the 2nd step. Finally, the remaining tokens, which do not participate in an invocation, are copied to the output array.

Decode Stream The decoding pass is illustrated in Fig. 8 and assumes that the stream is given as a sequence of n tokens with $s := \langle t_1, \ldots, t_n \rangle$. In order to decide, if a literal $t_i \in \mathbb{Z}$ is an argument of an executable expression, the distance to the next succeeding function token in the stream is relevant. For this purpose, the number of literals c_i up to the next function token are counted. In particular, literals $t_i \in \mathbb{Z}$ are marked with $c_i = 1$, so that the second line of Fig. 8 (*IsConst*) contains a 1 for each literal and a 0 for each function token:

$$c_i := \begin{cases} 1 & \text{if } t_i \in \mathbb{Z} \\ 0 & \text{else} \end{cases}$$

Next, up to $log_2(n)$ iterative passes are required to converge c_i:

$$c_i := c_i + c_{i+c_i}$$

In this example, only two accumulation steps (*Accum 1, Accum 2*) are necessary to count up to four arguments. Hence, for each literal $t_i \in \mathbb{Z}$, the next function token can be found at position $i + c_i$ in the stream. When assuming that the stream s has a length of n tokens, the maximum number of preceding literals a_i of a token t_i can be computed as:

$$a_i := \max_{j \in [1,n]} \{c_j | c_j + j = i\}$$

As a result, an invocation $t_i = f_j$ with $f_j \in F$ is executable, which is indicated by $e_i = 1$, if the number of available arguments a_i are greater or equal than the number of required arguments n_j:

$$e_i := \begin{cases} 1 & \text{if } \exists j \in \mathbb{N} : (t_i = f_j) \wedge (a_i \geq n_j) \\ 0 & \text{else} \end{cases}$$

Hence, in the example shown by Fig. 8, only the multiplications are executable. Eventually, the number of generated outputs o_i of a token t_i is given by:

$$o_i := \begin{cases} 0 & \text{if } \exists j \in \mathbb{N} : (t_i \in \mathbb{Z}) \wedge (t_{i+c_i} = f_j) \\ & \wedge (e_{i+c_i} = 1) \wedge (c_i \leq n_j) \\ m_j & \text{if } \exists j \in \mathbb{N} : (t_i = f_j) \wedge (e_i = 1) \\ 1 & \text{else} \end{cases}$$

The first case checks if the literal t_i is consumed by the next successor function f_j, which requires the function to be executable ($e_{i+c_i} = 1$) and the literal

to be within its argument list: ($c_i \leq n_j$). Likewise, the second condition determines the execution of the current expression if it is a function ($t_i = f_j$) and executable ($e_i = 1$). Finally, the *else* branch corresponds to unused literals and unmatched functions, which are replicated and thus create exactly one output. In the presented example, each of the multiplications produces one result and their arguments are removed.

Allocate Outputs In order to compute the destination position of each token, the output size o_i is accumulated using a parallel prefix-sum [32]. If the resulting stream fits into the shared memory, it can be rewritten in the next step. Otherwise, it is sub-divided into four blocks and written back into global memory.

Execute Functions In the last pass, the previously collected functions are executed and their results are stored in the shared memory. Similarly, the remaining tokens, which do not participate in an invocation are copied to the resulting stream.

In the next section, we present an implementation of this technique.

6 Implementation

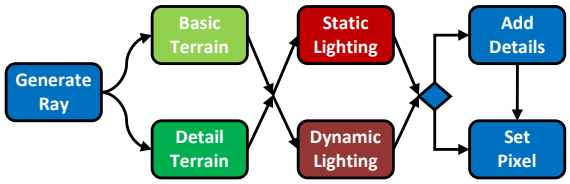


Fig. 9. Example function consisting of different stages for geometric and shading computations.

A prototype of our flexible architecture for terrain rendering has been evaluated on a GeForce GTX TITAN using CUDA 7.0. The rewriting process is started by a single kernel launch and performs the algorithm described in Section 5. For this purpose, the stream is divided into blocks of 512 tokens and each thread block stores at most two of them in the shared memory. In addition, there are 16 to 64 threads per thread block that decode the stream in parallel and execute the detected functions.

The structure of our example ray tracer is shown in Fig. 9 and consists of several stages which are described by the functional model (Section 4). In particular, we can switch between two detail levels of geometry and two lighting modes to vary quality, resource usage, and computation time. First, a fixed kernel generates the initial stream and creates a ray for each pixel. It either emits function calls to the '*Basic Terrain*' or '*Detail Terrain*' stages that compute the intersection point of the ray and the terrain through a combination of linear and

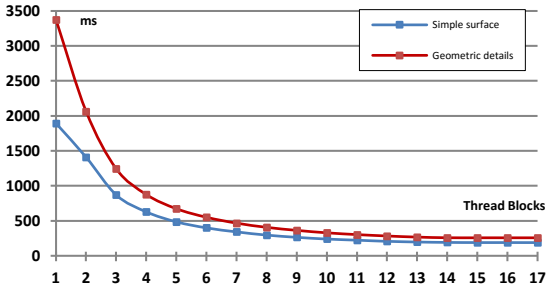


Fig. 10. Rendering time for different number of thread blocks and 16 threads per block on the GeForce GTX TITAN, which has 15 streaming multiprocessors.

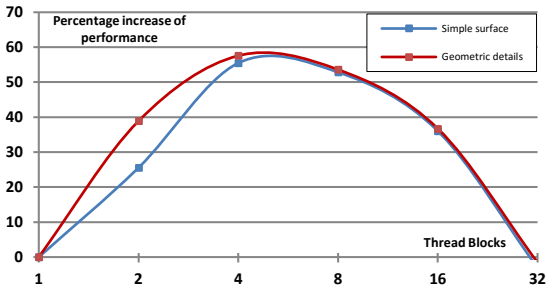


Fig. 11. Increase of performance when the number of thread blocks are doubled (16 threads per block).

binary search. Likewise, in the next stage, we can switch between pre-computed static or dynamic lighting models. In addition, the color is modulated by a microstructure and is interpolated for nearby pixels of the detail terrain ('*Add Detail*'). Finally, in the '*Set Pixel*' stage, the color data is written into the rendering target.

The ray tracer has been evaluated using four different heightmaps (*Terrain1*, *Terrain2*, *Fuji*, *Himalaya*) with a size of 512x512 samples and different configurations for geometry details and lighting models to show, in prinziple, feasability of our approach. Samples of the generated images shown in Fig. 12. For performance comparison, each test setup is used to draw 10 frames and the average rendering times per frames are listed in Table 1. Rendering times are nearly the same for all four data, since data size is equal. Also, the rendering times for static and dynamic lighting are comparable, so that this decision only affects the quality of the image. This would if more complex illumination models are used. In particular, it can be seen that the detailed rendering mode of all terrains requires more computational resources, but also creates more sophisticated images (Fig. 12 b).

Table 1. Rendering time for different configurations. Net Render Time is determined by the avg. render time without the system overhead (65ms) to handle stream rewriting

Terrain	Geometry	Lighting	Avg. Rendering Time	Net Render Time
Terrain1	Basic	Static	99.5 ms	34.5 ms
		Dynamic	101.1 ms	36.1 ms
	Detail	Static	136.5 ms	71.5 ms
		Dynamic	136.9 ms	71.9 ms
Terrain2	Basic	Static	98.8 ms	33.8 ms
		Dynamic	99.8 ms	34.8 ms
	Detail	Static	136.4 ms	71.4 ms
		Dynamic	136.3 ms	71.3 ms
Fuji	Basic	Static	99.6 ms	34.6 ms
		Dynamic	99.7 ms	34.7 ms
	Detail	Static	136.6 ms	71.6 ms
		Dynamic	137.0 ms	72.0 ms
Everest	Basic	Static	99.1 ms	34.1 ms
		Dynamic	100.0 ms	35.0 ms
	Detail	Static	136.4 ms	71.4 ms
		Dynamic	136.6 ms	71.6 ms

Tests have shown that a great portion of the rendering time ($\approx 65ms$) is consumed by the rewriting algorithm itself. Therefore additional tests to evaluate the core rewriting algorithm were performed. The scalability of the rewriting algorithm (Section 5) has been analyzed for the basic and detailed *Everest* terrain by varying the number of launched thread blocks. Since different thread blocks can run on distinct multiprocessors in parallel, a linear speed-up should be expected but there are two possible bottlenecks: First the linked list of blocks represents a global synchronization point and is protected by a mutex. However, each thread holds the lock only for a short amount of time. Second the stream is stored in global memory and must be copied into shared memory for rewriting. Though, it is accessed coherently, so that the available bandwidth can be maximized. As a result, the measurements indicate an continual decreasing render time, which stagnates at 15 thread blocks (Fig. 10). When the number of thread blocks is doubled, performance increases accordingly up to 16 blocks (Fig. 11) Since the GeForce GTX TITAN consists of 15 streaming multiprocessors (SMX) and similar tests on other graphic cards show the same coherence, we conclude that each thread block is mapped to a different multiprocessor and that each multiprocessor executes at most one thread block. Since on the Geforce GTX TITAN one thread block can execute at most 16 threads at once, for this test the number of threads was set to 16. However, the configuration resulting in the best performance, as seen in Table 1 utilized 8 SMX and 256 threads per block. This indicates that internally thread blocks and threads can be mapped differently depending on configuration and driver. Nevertheless our tests show the fundamental scalability of the parallel rewriting algorithm, but hardware limitation on graphic cards currently restricts scalability. Further improvements and optimiza-

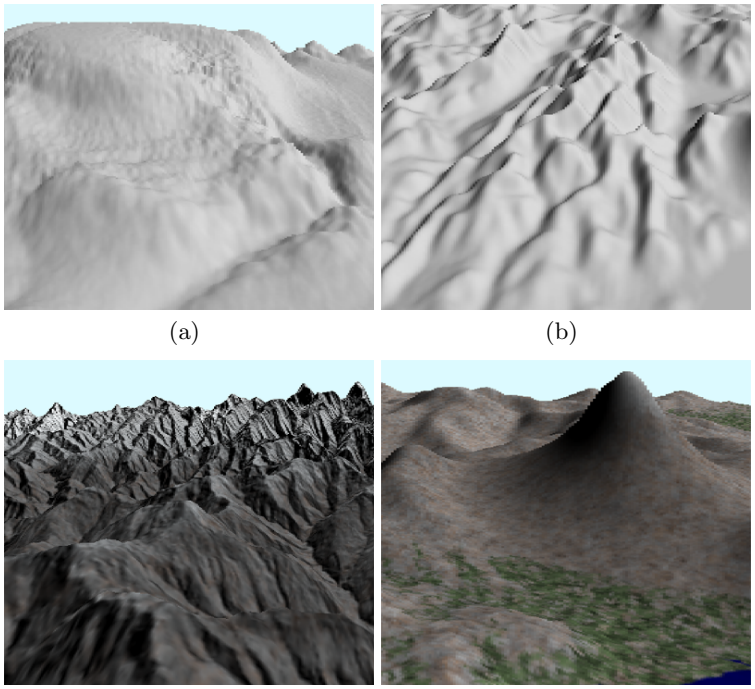


Fig. 12. Images rendered using our flexible terrain ray tracer. (Elevation data source: ASTER GDEM, a product of METI and NASA.)

tion should permit the execution of multiple thread blocks per multiprocessor to further scale our approach.

7 Conclusion

Balancing rendering time, rendering quality and resource consumption for ray tracing terrain surfaces is challenging. For this purpose, we presented a flexible ray tracing architecture. This is composed of a basic, high-efficient ray tracer and flexible, modular extensions to adjust the rendering process on demand with respect to time, quality and memory. To allow such a flexible approach to be mapped on the parallel hardware, we propose a novel execution model for scheduling dynamic workload on the GPU. A first prototype shows the feasibility. However, this is still subject to further development to increase the number of supported enhanced operators and to further facilitate the high parallelism of the GPU. Moreover the rewriting algorithm itself must be further optimized to the GPU to minimize execution time. Additionally, open questions still remain. Currently, manually composed presets determine the structure of the functional

model. However, an automated adjustment by means of restrictions, e.g. minimum frame rate or minimum quality standards, has to be investigated. Moreover, how combinations of operators influence the total quality of an image, which is furthermore mostly subjective, is still subject of ongoing research. In the future, we will also investigate the possibility to support embedded visualization of data, such as movement and weather data. Since the functional model allows for freely adjusting the rendering process and the dynamic task scheduling can parallelize even strongly heterogeneous execution task, embedding data into the terrain in compliance with quality and performance constrains can be beneficial.

References

1. Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007.
2. Lszl Szirmay-Kalos, Barnabs Aszdi, Istvn Laznyi, and Mtys Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum*, 24(3):695–704, 2005.
3. Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st MICRO*, 2008.
4. Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.
5. Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *ACM TOG*, 29(4):66, 2010.
6. Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: dynamic scheduling on gpus. *ACM Trans. on Graphics*, 31(6), 2012.
7. Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *High Performance Graphics*, HPG '10, 2010.
8. Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.
9. Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. *State of the Art Reports, EUROGRAPHICS*, 2001.
10. Thiago Ize, Ingo Wald, Chelsea Robertson, and Steven G Parker. An evaluation of parallel grid construction for ray tracing dynamic scenes. In *Interactive Ray Tracing 2006, IEEE*, 2006.
11. Art Tevs, Ivo Ihrke, and Hans-Peter Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In *I3D '08*, 2008.
12. Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

13. John C Peterson and Michael B Porter. Ray/beam tracing for modeling the effects of ocean and platform dynamics. *Oceanic Engineering, IEEE Journal of*, 38(4):655–665, 2013.
14. Fábio Policarpo, Manuel M Oliveira, and João LD Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D*, pages 155–162. ACM, 2005.
15. J Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. *URL: http://www. lonesock. net/files/ConeStepMapping. pdf*, 2(3):4, 2006.
16. Fabio Policarpo and Manuel M Oliveira. Relaxed cone stepping for relief mapping. *GPU gems*, 3:409–428, 2007.
17. Matt Pharr and Simon Green. Ambient occlusion. *GPU Gems*, 1:279–292, 2004.
18. Christopher Oat and Pedro V Sander. Ambient aperture lighting. In *Proceedings of SI3D*, pages 61–64. ACM, 2007.
19. Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seok-Yoon Jung, Shi-Hwa Lee, Hyun-Sang Park, and Tack-Don Han. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of ACM High Performance Graphics 2009*, pages 109–119, 2013.
20. Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. T&i engine: traversal and intersection engine for hardware accelerated ray tracing. In *ACM Transactions on Graphics (TOG)*, volume 30, page 160. ACM, 2011.
21. Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *High Performance Graphics 2009*, 2009.
22. Christian Dick, Jens Krüger, and Rüdiger Westermann. Gpu ray-casting for scalable terrain rendering. In *Proceedings of EUROGRAPHICS*, volume 50. Citeseer, 2009.
23. Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, 2013.
24. Lars Middendorf, Christian Zebelein, and Christian Haubelt. Dynamic task mapping onto multi-core architectures through stream rewriting. In *SAMOS '13*, 2013.
25. Lars Middendorf and Ch Haubelt. A programmable graphics processor based on partial stream rewriting. In *Computer Graphics Forum*, volume 32, pages 325–334. Wiley Online Library, 2013.
26. Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *SIGPLAN Not.*, 48(9):49–60, September 2013.
27. Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN ARRAY'14*, 2014.
28. Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004.
29. Chad Austin and Dirk Reiners. Renaissance: A functional shading language. In *Proceedings of Graphics Hardware*, 2005.
30. Tim Foley and Pat Hanrahan. Spark: modular, composable shaders for graphics hardware. In *ACM SIGGRAPH 2011*, 2011.
31. Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE MICRO*, 28(2):39–55, 2008.
32. Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.