Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

Explicit Cache Management for Volume Ray-Casting on Parallel Architectures

Daniel Jönsson[1], Per Ganestam[2], Michael Doggett[2], Anders Ynnerman[1] and Timo Ropinski[1]

[1]C-Research, Linköping University, Sweden
[2]Lund University, Sweden

Abstract

A major challenge when designing general purpose graphics hardware is to allow efficient access to texture data. Although different rendering paradigms vary with respect to their data access patterns, there is no flexibility when it comes to data caching provided by the graphics architecture. In this paper we focus on volume ray-casting, and show the benefits of algorithm-aware data caching. Our Marching Caches method exploits inter-ray coherence and thus utilizes the memory layout of the highly parallel processors by allowing them to share data through a cache which marches along with the ray front. By exploiting Marching Caches we can apply higher-order reconstruction and enhancement filters to generate more accurate and enriched renderings with an improved rendering performance. We have tested our Marching Caches with seven different filters, e. g., Catmul-Rom, B-spline, ambient occlusion projection, and could show that a speed up of four times can be achieved compared to using the caching implicitly provided by the graphics hardware, and that the memory bandwidth to global memory can be reduced by orders of magnitude. Throughout the paper, we will introduce the Marching Cache concept, provide implementation details and discuss the performance and memory bandwidth impact when using different filters.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Volume Rendering—

1. Introduction

Modern Graphics Processing Units (GPU) are complex parallel devices which allow SIMD processing. To acquire high image fidelity and high rendering performance at the same time, algorithms need to take advantage of several aspects of a GPU's architecture, which itself is subject to change undergoes technological advancement. One aspect of high importance which needs to be considered, is the increasing gap between computational capability and memory bandwidth of modern GPUs [OHL*08]. While compute power grows, memory bandwidth does not grow at the same rate. Algorithms running on modern GPUs can be roughly classified on whether they are compute bound or memory bound. The former spend a major amount of cycles performing computations, while the latter use most of the cycles accessing data usually located in graphics memory. While compute bound algorithms are supported by the high degree of parallelism of modern GPUs, memory bound algorithms suffer from the increasing gap between computational capabilities and memory bandwidth. To reduce this effect, specialized memory is used to grant fast data access through caching. This goes even that far, that the Larrabee architecture otherwise solely build with standard CPUs, integrates dedicated texturing hardware to allow better performance for memory bound algorithms [SCS*08]. However, when providing such dedicated hardware, besides the costs mainly two problems occur. First, the same data caching strategy is used independent of the algorithm executed by the parallel processing units. This is problematic, since data access patterns vary drastically even among algorithms performing similar tasks. For instance, when implementing volume rendering through texture slicing, the 3D texture representing the volumetric data set is accessed in a slice pattern, while ray-casting accesses the 3D texture in a pattern following each viewing ray. It is obvious, that these data access patterns require different caching strategies to minimize cache misses. Second, todays texturing hardware is designed to offer hardware support for linear interpolation only, while in many application

© The Eurographics Association 2012.

DOI: 10.2312/EGPGV/EGPGV12/031-040

delivered by EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

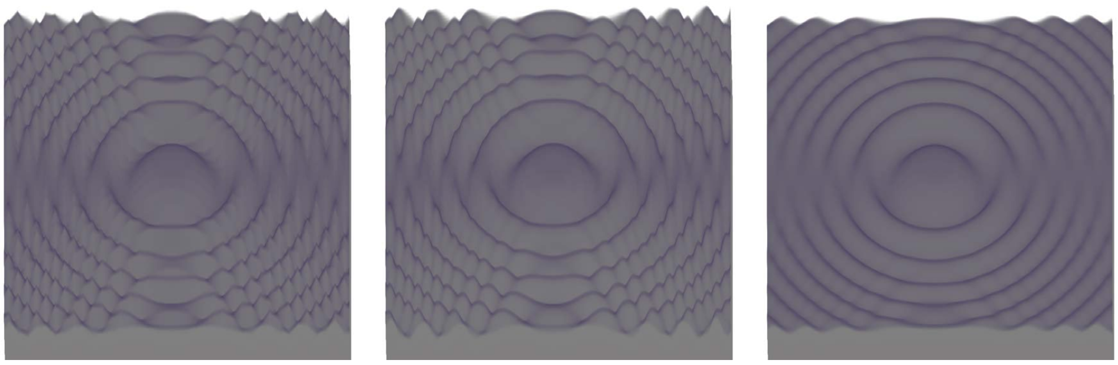
(a) Trilinear interpolation (speed: -2.5x). (b) Catmul-Rom (speed: 3.3x). (c) B-spline (speed: 3.3x).

Figure 1: *Comparison of data reconstruction and performance improvement for the Marshner-Lobb dataset using our method. Hardware trilinear interpolation was used in (a) for comparison, while our method implements it in software which causes the slowdown.*

areas it can be necessary to apply more complex filters when accessing data [HAM11].

Within this paper we address these two problems, by introducing explicit data caching strategies for memory bound volume rendering algorithms. By using our approach, we can increase rendering performance and improve image quality by enabling the application of higher-order reconstruction and enhancement filters when accessing data. We have chosen volume rendering as our application scenario, as it is known to be memory bound by dealing with often large 3D data sets. To reduce the limitations resulting from the memory bound behavior, we introduce the *Marching Caches* concept as an explicit caching strategy for state-of-the-art GPU-based volume ray-casting. Marching Caches improve performance by reducing the amount of volume data read from slow memory. This is done by exploiting inter-ray coherence, whereby computational power is traded for memory bandwidth through neighboring rays sharing fast memory which they incrementally update. Thus, it becomes possible to apply higher-order reconstruction and enhancement filters to generate more accurate and enriched renderings with an improved rendering performance as demonstrated in Figure 1. This is mainly achieved by taking advantage of data locality, which has previously been pointed out as being important by Rixner et al. in the development of the Imagine Stream Processor [RDK*98] [Cor11]. Thus, by moving data closer in terms of the architecture to the thread execution, performance can be improved. Within our approach, we utilize a thread's ability to access fast memory regions and use it as an explicit cache, by moving voxel data into the memory that is shared between the group of threads. By moving the data from slow memory into the fast memory regions this ensures data is available for reuse by several threads all at very low latency, unlike having to read the data again from slow memory or running the risk that data is evicted by the implicit cache.

We have tested our Marching Caches with seven different filters: Catmul-Rom, B-spline, ambient occlusion projection, trilinear, bilateral, gradient and filtered gradient. For some of these filters we were able to obtain a speed up of four times as compared to using the caching implicitly provided by the graphics hardware. Furthermore, the proposed explicit caching reduces the memory bandwidth to global memory by orders of magnitude. Besides these benefits, our explicit caching also allows the programmer who knows the data access pattern, to reorder computations to make maximum use of the available shared memory. Thus, it allows further improvements for specialized rendering algorithms.

While we mainly focus on volume ray-casting, a similar analysis as performed within this paper can be applied to other data intensive algorithms and potentially lead to dedicated explicit caching schemes, which may allow similar gains.

2. Previous Work

The concept of tracing several rays together has been used previously in several contexts including volume rendering [MKW*02]. Meissner et al. ray cast four rays together to ensure enough computation is available to cover the latency of the compositing pipeline. For the generation of realistic images using interactive ray tracing the concept of packet tracing [WBWS01] also involves tracing several rays at the same time. In particular the rays in each packet are forced to travel together, to ensure the same data is used for each ray as it traces its path through the 3D dataset. While originally used in a four-wide configuration to make best use of CPU SIMD engines [WBWS01], it has also been applied to GPU based ray tracing as well [HSHH07].

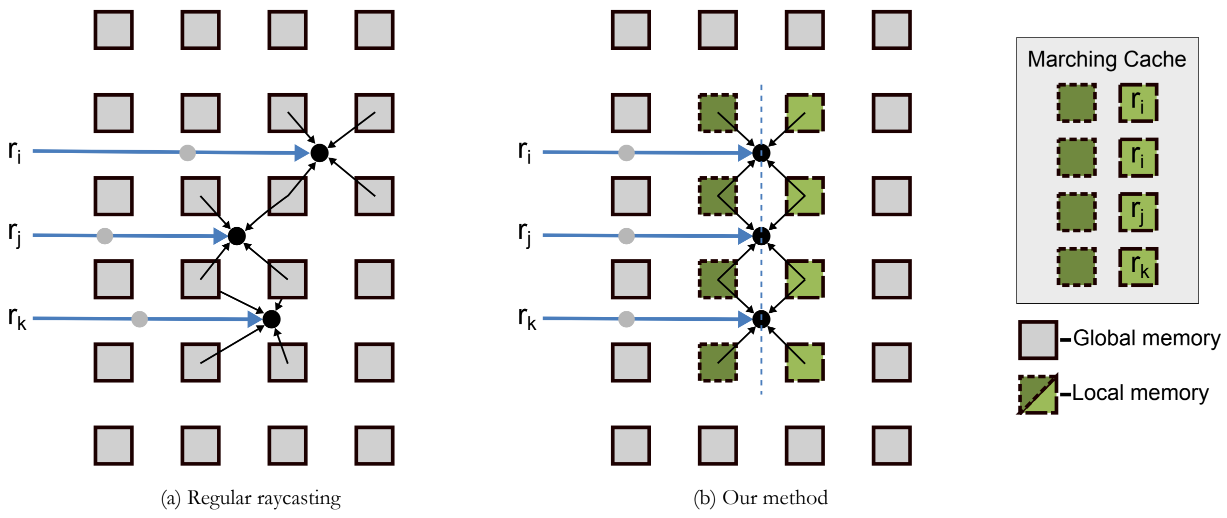

Figure 2: *A vast amount of reads from slow memory, indicated by grey boxes, are necessary when performing filtering for regular ray-casting. Our method works by incrementally updating a cache which marches along a ray front. Dark (short dashes) and light (long dashes) green represent fast memory where only the light green is updated from slow memory. The neighboring rays, r_i, r_j and r_k, work together to fill up the missing values. Using Marching Caches in this example, r_i will read two values from global memory and the other rays one each. For regular ray-casting each ray needs to read four values.*

A vast amount of work has been directed into designing filters, and summaries have been made by Meijering [Mei02] and Unser [Uns00]. A comparative study of prefiltered reconstruction techniques for volume-rendering was made by Csébfalvi [Csé08], which showed that the image quality can be improved considerably when using such techniques. Recently, Hossain et al. [HAM11] showed how to also improve gradient estimation by prefiltering the data. As our approach works for any filter, the work of both Csébfalvi [Csé08] and Hossain et al. [HAM11] could benefit from using our method.

Although more advanced filters can be used to improve image quality, they will not be used for visualization if it prevents interactivity. For 3D meshes, data access pattern where improved by Tchiboukdjian et al. [TDR10] using a cache-oblivous layout which provided coherent traversal. Hadwiger et al. [HTHG01] enabled high quality filtering on graphics hardware by scattering instead of gathering input data. They reconstructed one slice at a time which could possibly be used in a slice-based volume renderer. Sigg and Hadwiger [SH05] and Lee et al. [LYS*10] introduced methods to improve the performance of cubic filters in volume rendering by using combinations of hardware trilinear filtering and virtual samples, respectively. As such their methods are not extendable to other filters. As pointed out in the CUDA C Programming Guide [Cor11], shared memory can be used as a user managed cache. This was utilized by Mensmann et al. [MRH10] which introduced slab-based ray-casting where trilinearly interpolated sample points along rays could be reused by neighboring rays. However, as the cached data in their work depends on distance between rays and step size it could only be used as an approximation for a small set of filters such as ambient occlusion. In this work raw volume data is cached and updated using an incremental scheme which saves bandwidth and enables support for arbitrary filters.

3. Marching Caches Concept

As briefly mentioned in Section 1, the Marching Caches approach relies on two observations in volume ray-casting. First, inter-ray coherence, i. e., neighboring rays access the same data in a volume, and second, that the access to volume data is a bottleneck in current parallel graphics architectures. The data access bottleneck is especially a problem when considering ray-casting, since the amount of data that needs to be read for each ray is high compared to the number of performed computations. Only a few computational operations are performed for each step along the ray, usually consisting of compositing and the computation of the next step along the ray. To take advantage of these observations we propose Marching Caches, which exploits inter-ray coherence to reduce the data access bottleneck. To do so, we create a cache containing the required data around a bundle of rays and incrementally update it in the direction of the rays as the rays traverse the volume (see Figure 2). Thus, the cache marches along the ray bundle. As also can be seen in Figure 2, a bundle of rays is aligned such that the ray front accesses the same data simultaneously. The dark green boxes represent data that has been cached in the previous step and the light green boxes is new data that is read for the current step. Using this caching scheme, we are able to reduce cache misses and thus considerably reduce the bandwidth to global memory. In Section 4 we will explain how to synchronize the ray front and incrementally update the cache as the ray front advances.

4. Algorithm

The Marching Cache is enabled in three steps, as shown in Figure 3. The first step ensures that the ray front of each ray bundle is synchronized. The second step calculates which part of the cache each ray needs to update when the cache becomes invalid, and the third step handles the cache updates during ray-casting and enables filtering the cached data. The first and second step occur before the actual ray traversal, and enable a ray bundle to share data, compute the size of the cache and determine which part of the cache each ray needs to update. The third step occurs during ray traversal and ensures that the cache is incrementally updated, i. e., that it marches along with a ray bundle. To ensure that rays efficiently share data using incremental updates, the following requirements for a ray bundle must be met:

- Ray start and end points are synchronized.
- The positions at which the rays sample must be spatially close.

The first requirement is necessary since the rays fetch data to a common cache and a ray cannot be terminated before the others as this would leave that area of the cache outdated. The second requirement has two implications. First, it ensures that the required size of the cache is as small as possible and second, that as much data as possible is reused between rays. As illustrated in Figure 3, these requirements are met by projecting the entry and exit points of the rays in a ray bundle to be the same distance from the camera as the closest and farthest entry and exit point respectively. This projection step ensures, that all rays within a ray bundle are of the same length, resulting in a minimal ray front which the Marching Cache needs to cover.

The second step in Figure 3 computes which positions in the Marching Cache each ray in a ray bundle should update. To enable incremental updates, only the sides which are invalid are updated, resulting in three different cases based on the direction of the ray, which we will discuss in more detail below.

In the third step depicted in Figure 3, we ensure that the Marching Cache is updated and marches along with the ray bundle. At each step along the ray, the Marching Cache is also advanced and if it has entered another set of voxels, the sides that need updating are updated. We will in the following subsections explain each step of the algorithm in detail.

4.1. Project entry and exit points

To synchronize a ray bundle we make sure that all rays are of the same length and start relatively close to each other. We do this by first finding the start and end points which are closest and furthest away from the camera in texture space within the ray bundle. Once found, the entry point, p_{en}, and exit point p_{ex} is projected along the direction of the ray, $\vec{d}$, using:

$$p'_{en} = p_{en} - (s - s_{min})\vec{d} \tag{1}$$
$$p'_{ex} = p_{ex} + (s - s_{max})\vec{d} \tag{2}$$

where p'_{en} and p'_{ex} are the projected entry and exit points, s is the distance from the point to the camera and s_{min} and s_{max} are the distance to the camera of the closest entry point and the farthest exit point respectively.

4.2. Calculate cache update region

In this subsection we will first explain how to determine the size of the cache, and then how to calculate it's incremental update. For explanatory purposes it is assumed that the filter and cache sizes are cube shaped, even though it is not required or done in practice.

The size of the cache must be large enough to cover the ray bundle wavefront plus additional borders determined by the size of the filter, but small enough to fit inside the cache memory and allow minimal memory access to slow memory. A simple axis aligned bounding box is used as cache because it requires minimal computation for indexing even though it might not have an optimal fit for the ray bundle. The extent of the ray bundle is determined using projected exit points which allows support for perspective projection. For each ray bundle, the extent is first computed and then the maximum extent of all the ray bundles is selected as basis for the size of the cache. Once the size of the Marching Cache has been determined, it is possible to calculate which part of the cache each ray should update.

As mentioned above, there are three such update cases which can occur when the cache marches along the ray bundle. In the first case, one cache side needs to be updated, in the second case two sides and in the third case three sides need to be updated (see Figure 4). Thus, each xyz-axis may need to be updated in the positive, negative or no direction, which results in $3^3 = 27$ combinations. In practice, the case where no update in any direction is necessary is not computed and as such 26 different cases need to be calculated. For each combination, the number of fetches that each ray needs to perform should be minimized. Space filling curves [Sag94] could be used to calculate the region, but in this work a simpler method is used. First, the size of the 3 (7 if rectangular box) possible regions are computed using

$$\begin{aligned} n_1 &= C^2 \\ n_2 &= C^2 + C(C-1) \end{aligned}$$

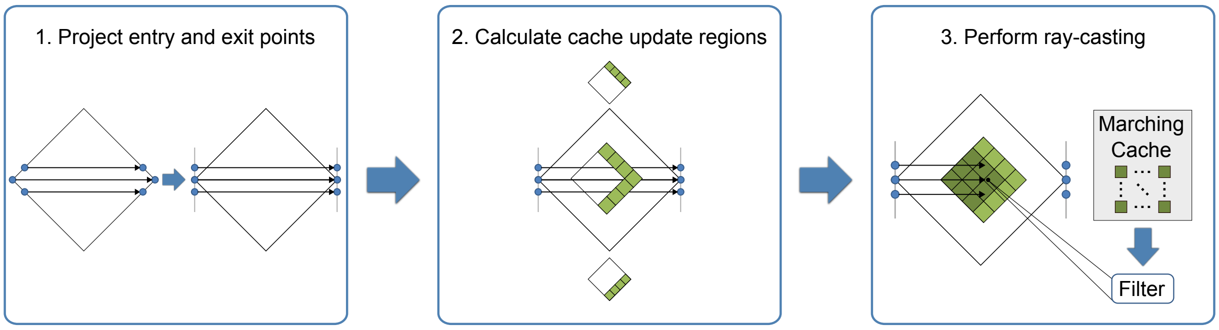
1. Project entry and exit points
2. Calculate cache update regions
3. Perform ray-casting
Marching Cache
Filter

Figure 3: *To allow sharing of data between rays during ray-casting a number of steps are necessary. In the first step, entry and exit points are projected such that the rays are of the same length within a ray bundle. The second step calculates which part of the cache a ray in a ray bundle should update during ray-casting. In the last step, regular ray-casting is extended to incrementally update the cache and use fast local memory instead of slow global memory for filtering.*

$$n_3 = C^2 + C(C-1) + (C-1)(C-1) \quad (3)$$

where C is the size of the cache in each dimensions. Here, n_1 is simply the area of one side, n_2 excludes one overlapping part of the cache and n_3 excludes two overlapping parts of the cache. Then, the optimal number of fetches, f_i, for each case shown in Figure 4 is computed using

$$f_i = \lceil n_i / R_n \rceil \quad (4)$$

where R_n is the number of rays in a ray bundle. For simplicity, each ray is then allowed to fetch data to one row which means that the last one in the row may fetch less than the others. To compensate for this, the number of fetches each ray performs are increased until the area is covered. The 3D index to start and end updating for each ray and case is then stored to be used during ray-casting.

4.3. Ray-casting

The Marching Cache first needs to be setup before the rays begin their traversal, and once they begin, incremental updates can be performed as certain parts of the cache become invalid. We will begin by describing how to setup the cache and then go into how the incremental updates are performed.

First the origin, o of the Marching Cache is found by offsetting the minimum entry point, p_{emin} of the ray bundle with half of the filter size, N, to ensure that no data is missed.

$$o = \lfloor p_{emin} - N/2 \rfloor \quad (5)$$

Then the direction of the cache is computed using the minimum entry and exit point of the ray bundle. Before starting the ray traversal the cache must be filled. Here the optimal number of fetches is used since deciding the indices to update is easy and performing extra computation can be afforded since it is only performed once per frame. Each ray will perform n fetches given by

$$n = \lceil C_n / R_n \rceil \quad (6)$$

where C_n is the size of the cache. The cache can then be filled in parallel using Algorithm 1, where r_i is the index of a ray within the ray bundle and C_i is the size of the cache in dimension i.

Algorithm 1 Fill cache
```
start = $r_i \cdot$ n
end = min(start+n, $C_n$)
for i = start to end do
    position = o + ( i mod $C_x$, i / $C_x$, i / $(C_x \times C_y)$ )
    cache data at position
end for
```

Once the cache has been setup, the ray traversal can begin. The Marching Cache is moved along the ray bundle in a synchronized manner with the actual sampling points. If the origin enters a different voxel, it needs to be updated, otherwise no operation is performed. By using the difference between the previous and current voxel origin of the cache together

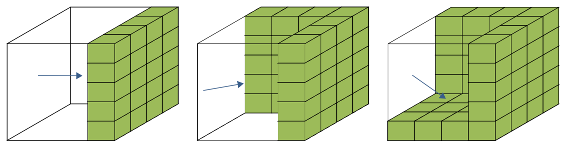
(a) Case 1: One side needs to update (b) Case 2:Two sides needs to update (c) Case 3: Three sides needs to update

Figure 4: *Incremental updating of the cache is performed at each step along the ray instead of updating the whole cache. Depending on the direction and step size of the rays, 0 to 3 sides of the cache may need to be updated.*

with the direction, it is possible to determine which combination of sides are invalidated. In order to avoid branching, an integer representation of the cache direction is computed indicating if the direction is zero (0), negative (1) or positive (2) in each dimension. The index to the data computed in the previous step can then be computed without branching:

$$\begin{aligned} \vec{i} &= \vec{C}_u \times \vec{d}_i \\ k &= (9\vec{i}_z + 3\vec{i}_y + \vec{i}_x - 1)R_n + r_i \end{aligned} \tag{7}$$

where $\vec{d}_i$ is the integer representation of the direction and $\vec{C}_u$ is 1 in the directions the cache needs to be updated and 0 otherwise. In Equation 7, the update combination is first determined in 3D and then flattened to a 1D index. Furthermore, one is subtracted from the 1D index to remove the case where no side has changed as it is handled separately.

To enable the incremental update of the cache, an internal offset o_c is used. If the cache moves in the positive direction, the internal offset is incremented by one and if it moves in the negative direction it is incremented by the size of the cache. The final internal offset, $o_c^{'}$, is then determined by performing the modulo operation:

$$o_c^{'} = o_c \, mod \, C \tag{8}$$

where C is the size of the cache in each dimension. To lookup a value in the cache, the input coordinate p_{in} is transformed using the internal offset and size of the cache using:

$$p = (p_{in} + o_c^{'}) \, mod \, C \tag{9}$$

Coordinate p takes the incremental updates into account and can be used to directly index the data within the cache.

Even though less data is fetched and memory is saved, increased computation is required to handle the Marching Cache. If the computation to handle the Marching Cache is too expensive, it will cancel the effect of decreasing the bandwidth requirements. In the next section we will discuss how to efficiently compute the discussed steps on parallel architectures and introduce some optimizations necessary to make it beneficial.

5. Implementation

To realize the Marching Cache method we have chosen to use OpenCL. Accordingly, we will also use the OpenCL terminology within this section. To familiarize the reader with the concepts of OpenCL we provide a short summary here, which the experienced reader may skip. To begin with, hardware which executes OpenCL code is referred to as a compute device. The compute device has a number of compute units which each can execute a number of work-items (threads). Each work-item has a private memory (usually registers) which no other work-item may access. Each compute unit can share data through local memory, which is utilized for the Marching Cache. Memory which can be accessed by all threads on the compute device is called global memory, or constant if it does not change. Out of the different memories, private memory is the fastest, then local memory and at last global memory which is substantially slower and access should be minimized to. An overview of the memory layout can be seen in Figure 5.

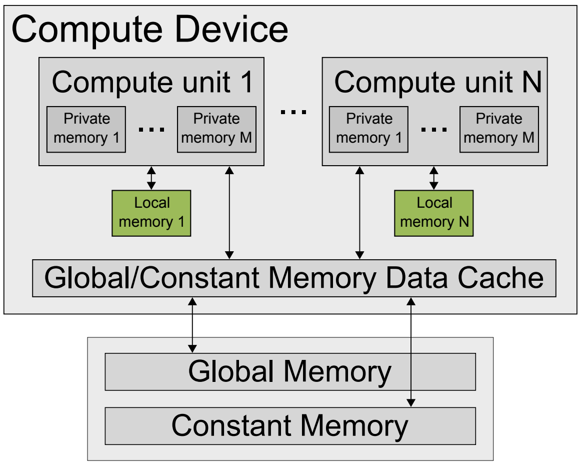

Figure 5: *Memory layout of OpenCL enabled architectures. Marching Caches utilize local memory to share data between rays within a ray bundle.*

As mentioned previously, Marching Caches focuses on using the local memory to minimize the access of global memory. Furthermore, incremental update of the Marching Cache is performed in order to minimize global memory access even more. As illustrated in Figure 3, we perform two steps on the input entry and exit points before performing the ray traversal. During ray traversal the Marching Cache is incrementally updated and used to lookup data for the filters. Using the OpenCL syntax, a ray bundle is mapped to a work-group and a ray is mapped to a work-item. In the following subsections we will follow the order of the pipeline which means that first, implementation details on the projection step is given, then how to efficiently calculate correct cache update locations and last how to optimize lookups in the Marching Cache.

5.1. Project entry and exit points

In order to find the start and end points which are closest and furthest away from the camera within the ray bundle used for equation 2, a parallel reduction technique described in [SHZO07] is used. The distance to the camera in texture space is then computed and equation 2 is used to project the entry and exit points which can then be stored in two buffers.

Some ray bundles will include rays which both miss and hit the bounding box. Since the scheme is dependent on the neighboring rays for fetching values into the cache, we cannot let them terminate because this would result in invalid data. Instead, we also execute them, but along a virtual ray, and discard the results afterwards. One might think that also executing rays which miss, creates a lot of overhead, but the work-items within a work group must wait for the last work-item to finish. Therefore, it does not matter as much if the missed rays are executed or not. If all of the rays within the ray bundle misses, no ray needs to traverse and all can terminate directly.

5.2. Calculate cache update region

Even though one might have a different size for each ray bundle, all ray bundles would have to allocate the same amount of data as it cannot be done dynamically in OpenCL and it would also require different update regions for the incremental update of the Marching Caches. Therefore, the maximum size of all ray bundles are used to determine the size of the cache. Determining the maximum extent is performed in a two step process. First, each ray bundle determines its own extent using the projected exit points and stores it. Then, in a second pass, the maximum extent of all ray bundles is determined. Parallel reduction is used in both steps. The parallelism is also used when computing the 26 update combinations for the cache. Each work-item in a work-group computes the update region of one ray, and each work-group computes one of the 26 different combinations. The use of reduction when computing the cache size and parallel computation of the update combinations enables the whole step to be computed in a small fraction of the total computation time.

5.3. Modulo optimization

Equation 9 will be heavily used, both when applying filters and updating the cache, but the modulo operator is very slow. Therefore, some optimizations need to be performed. By realizing that the size of the cache is constant the whole frame we can utilize the following:

$$x \, mod \, y = x - \lfloor x/y \rfloor y \qquad (10)$$

where $1/y$ can be precomputed and instead of a division, a multiplication can be performed which is much faster. Using the optimization in equation 10 improves the Marching Cache performance by five times on the graphics card used for benchmarking in the next section. Without the modulo optimization, the Marching Cache would actually be slower than regular ray-casting.

6. Performance Evaluation

To evaluate the performance of the Marching Cache we implemented a range of filters which use different amounts of computations and data. Besides using existing filters we also created an artificial filter which allows us to vary the size of it and the amount of computations. All tests where performed on a computer with an Intel Xeon 2.67 GHz processor, 6 GB

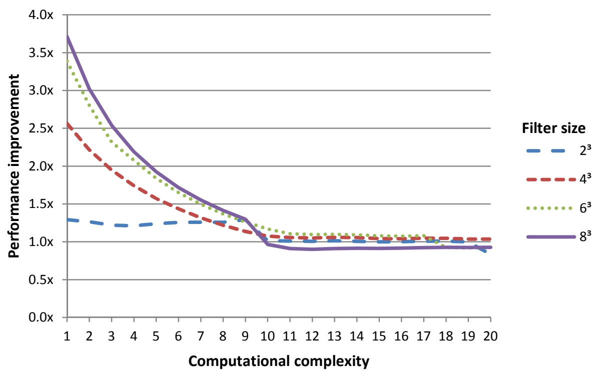

Figure 6: *A speedup of almost four times can be achieved with filters of size 8^3 with low computational complexity compared to a ray caster not using Marching Caches. However, when the computational complexity of the filter increases enough, the latency of fetching the data can be hidden and using the Marching Caches can decrease the performance.*

random access memory and a Nvidia Geforce 570 graphics card. The frames per second (fps) were averaged over 100 frames, using a camera position of (0,0,1.5) looking at the origin and rotating 360° around the origin with a screen of size 1024^2. The Engine dataset (8-bit, 256^3 voxels) was used to benchmark all filters. In order to show the true difference between our method and ordinary ray-casting no acceleration methods were used during performance measurements. However, acceleration methods such as early ray termination can be implemented as described in [MRH10]. Empty-space skipping as described in [KW03] can also be applied, one just needs to make sure that the cache is complete when data has been skipped. To evaluate the performance of the built-in implicit caching, we projected the start and end points and applied filters without using local memory. We found that the performance was equivalent to a standard openCL raycaster and therefore used the standard raycaster in the tests.

Current graphic cards can hide the latency of memory transactions if enough computations are available. Therefore, it is reasonable to expect that a filter which is expensive enough will turn the ray-casting from bandwidth bound to compute bound. When the ray-casting is compute bound the Marching Caches will not increase the performance, it could actually decrease the performance. To show how much computations that can be performed before a raycaster without Marching Caches becomes compute bound an artificial filter was designed. The artificial filter performs three simple operations for each data element, it first computes a weight by dividing one with the current computation iteration number, i, then accumulates the weight times the current data value, x, and at last accumulates the weight, w_i:

$$\begin{aligned} w_i &= 1/i \\ v &= v + x \cdot w_i \\ w &= w + w_i \end{aligned} \tag{11}$$

After performing the operations, the value, v, is normalized with the accumulated weights, w. The artificial filter simulates a simple filter where the filter weights has not been precomputed. As can be seen in Figure 6, the more data used (larger filter size) and fewer computations made, the greater is the benefit of using the Marching Cache. Furthermore, at around 10 iterations of equation 11 per data element the performance improvement levels out and it is not as advantageous to use the Marching Cache. In order to evaluate the amount of unnecessary reads from global memory that is introduced by the simplified algorithm described in section 4.2, the average amount of reads and unused reads for each case was measured. The maximum amount of reads per ray occurs for case 3 in Figure 4(c) which, for the filter sizes 2^3, 4^3, 6^3 and 8^3 are 1, 2, 3 and 4 respectively. Two percent additional reads needed to be performed for the largest filter size but for the other filter sizes no fetches were wasted.

Although the behavior for the artificial filter is interesting, the real value of the Marching Cache is not apparent until real filters are used. The following filters, shown in Table 1, were implemented: Catmull-Rom spline and Cubic B-spline [CR74] for reconstructing data; a 3D version of a Bilateral filter [AW95] for edge preserving denoising; a version of local ambient occlusion [HLY10] and two gradient filters for shading. Hardware trilinear interpolation was used for data reconstruction in bilateral filtering, local ambient occlusion, gradient and filtered gradient as it improved performance slightly (roughly 10 %). To compute the gradient we implemented central difference. Filtered gradient was computed using a weighted sum of the neighbourhood gradients and local ambient occlusion as the the average of the neighborhood contained by the size of the filter.

In Figure 1 we show how the different data reconstruction filters compare using the Marschner-Lobb dataset configured as described in [ML94]. To show how Marching Caches behaves when the user zooms we measured the time it took to render each frame when going from an overview to a close-up of the dataset as shown Figure 7. Two datasets were used, the Engine as before and an artificial 16-bit dataset of size 512^3. The filter was set to ambient occlusion of size 4^3, and the zooming animation was measured over 100 frames. The results can be seen in Figure 8.

7. Results

As shown in the Table 1, the Marching Caches outperforms the built-in texture cache by 1.3-4.7 times for existing filters of size 4^3 or greater. If extended to the best filter on Cartesian Cubic lattices in [HAM11] and cubic filters in [Csé08],

(a) Zoomed out Engine.

(b) Zoomed in Engine.

Figure 7: *Snapshots of start and end camera view of the Engine dataset used for measuring performance during zooming with ambient occlusion projection.*

Table 1: *Performance improvements for filters compared to ray-casting without Marching Caches.*

Filter	Size	Performance improvement
Catmul-Rom	4^3	3.4x
B-spline	4^3	3.4x
Trilinear interpolation	2^3	-3.5x
Ambient occlusion	4^3	2.4x
	6^3	4.7x
	8^3	2.6x
Bilateral	4^3	1.3x
	6^3	1.8x
	8^3	1.2x
Gradient	3^3	-1.3x
Filtered gradient	5^3	2.2x

which are of size 4^3, the Marching Caches should be able to improve the performance by 3-4 times. The filters which requires more computation have less performance improvement which is caused by latency hiding in regular ray-casting and that the Marching Cache is compute bound. Using an artificial filter, it could also be validated that when too much computation is performed the advantage of the reduced bandwidth decreases. Looking at Figure 8 it can be seen that the render time when using Marching Caches is higher if the rays are not subject to a high degree of inter-ray coherence, i. e. not as many voxels are shared between the rays. However, as soon as the zoom factor becomes higher, i. e. the user zooms in, inter-ray coherence increases and thus using the Marching Caches concept has a clear performance benefit. Nevertheless, as can be seen in Figure 8, exceeding a certain zoom factor results in a linear slightly increasing performance. We suspect that the cause for this is two-fold. First, the rays get shorter as the viewpoint enters the volume. Second, the high degree of inter-ray coherence results in only a few voxels being used, which makes the performance behaviour of the Marching Caches concept more sim-

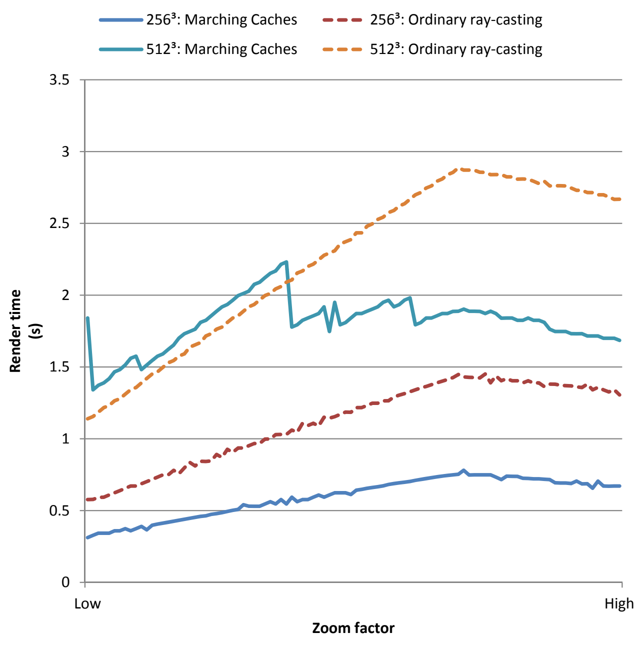

Figure 8: *Render time for each frame during a zooming from overview to close-up using the Engine (256^3) and an artificial (512^3) dataset.*

ilar to the behaviour of the rather local cache incorporated in modern GPUs.

The main advantage of this approach is the amount of bandwidth saved. As an example, consider a three dimensional filter of size N^3, where N is the size of the filter in each dimension. Using a naive approach would require, for each point along the ray, N^3 data values to be fetched. Marching Caches utilize two properties which substantially reduce the amount of data fetched for each ray. First, each ray in a bundle only updates a part of the Marching Cache. Therefore, if the number of rays in a bundle is denoted R_n and the size of the cache in each dimension is C, each ray would perform a maximum of $\lceil C^3/R_n \rceil$ fetches. Second, since only the sides of the Marching Cache that needs updating is updated each ray performs a maximum of $\lceil C^2 + C(C-1) + (C-1)(C-1))/R_n \rceil$ fetches, given by Equations 3 and 4, when three sides are invalid. As an example, a typical case is when the ray bundle consist of $R_n = 64$ rays and the Marching Cache is of size $C = 10$ for a filter of size $N = 4$. Using the naive approach would require 64 fetches per step along the ray while the method presented here would require a maximum of four fetches per step along the ray, 16 times less. By measuring the amount of reads from global memory performed per ray for the Engine dataset it can be concluded that the amount of bandwidth saved ranges from 8-128 times, a difference of several orders of magnitude, depending on filter size. Furthermore, it could be shown that the simple algorithm for calculating which part of the cache each ray should update in section 4.2 performs well in practice; only wasting on average two percent of the reads for the largest filter size and none for the others.

8. Limitations

The main drawback with using the Marching Cache is the limited amount of local memory on current architectures. When the amount of voxels covered by a ray bundle is too large, local memory will not be large enough. The GPU used for performance evaluation has 48kB of local memory which means that a cache of about 23^3 is supported if float data type is used and about 28^3 for half data type. To put this into context, a 512^3 dataset with a filter size of 4^3 uses on average a cache size of about 12^3 for a screen size of 1024^2.

When the amount of physical memory is not enough there are a couple of options available. The work-group size can be decreased, which means that fewer voxels will be covered at the expense of performance. Another possibility is to increase spatial proximity of the rays, for instance by increasing the screen resolution. If the amount of voxels covered is caused by the zoom level selected by the user, a lower resolution of the volume could be used. If none of the mentioned options are viable, one has to resort to ray-casting without the Marching Caches.

9. Conclusions and Future Work

In this paper, we have introduced Marching Caches as an explicit strategy for GPU-based volume ray-casting. By exploiting inter-ray coherence it is possible to share cached data in between neighboring rays and thus allow to increase rendering performance and image fidelity when applying complex filter kernels. We show, that with modern GPUs it becomes possible to get up to 4.7 times increased performance when using Marching Caches compared to regular ray-casting. Additionally, the memory bandwidth to global memory, which is a limiting factor for many GPU algorithms, could be reduced by orders of magnitude, which we show theoretically and demonstrate through profiling. We expect the performance improvement to rise even more with the development of new GPUs, since the expected increased compute power of these models is beneficial for our compute bound explicit caching.

In the future, we would like to use the Marching Caches approach in order to investigate how inter-ray coherence can be exploited to deal with various zoom levels, i. e., how sufficient data reconstruction can be enabled with ray-casting when zooming out such that undersampling occurs. Furthermore, an application to ray-tracing based rendering algorithms seems to be an interesting direction. By proposing Marching Caches, we could show how to modify volume ray-casting algorithms from being memory bound GPU algorithms into compute bound algorithms. In the future, we would like to apply similar strategies to other memory bound GPU algorithms, and thus allow them to take full advantage of the increasing compute power of future GPUs. Ideally, a conceptual approach for transforming general GPU algorithms from being memory bound to compute bound reduces the demand of dedicated data access hardware and thus decreases GPU development costs.

Acknowledgments

The presented method was developed using the Voreen volume rendering engine (www.voreen.org), which is an open source visualization framework. We would also like to thank the anonymous reviewers for their feedback. This work was supported through the Intel Visual Computing Institute, Saarbrücken, Germany, the Excellence Center at Linköping and Lund in Information Technology (ELLIIT) and the Swedish e-Science Research Centre (SERC).

References

[AW95] AURICH V., WEULE J.: *Non-Linear Gaussian Filters Performing Edge Preserving Diffusion*, vol. 17. Springer-Verlag, 1995, pp. 538–545. 8

[Cor11] CORPORATION N.: *NVIDIA CUDA C Programming Guide*, June 2011. 2, 3

[CR74] CATMULL E., ROM R.: A class of local interpolating splines. *Computer aided geometric design. Academic Press.* (1974), 317–326. 8

[Csé08] CSÉBFALVI B.: An Evaluation of Prefiltered Reconstruction Schemes for Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 14*, 2 (march-april 2008), 289 –301. 3, 8

[HAM11] HOSSAIN Z., ALIM U. R., MÖLLER T.: Toward High-Quality Gradient Estimation on Regular Lattices. *IEEE Transactions on Visualization and Computer Graphics 17* (2011), 426–439. 2, 3, 8

[HLY10] HERNELL F., LJUNG P., YNNERMAN A.: Local Ambient Occlusion in Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 16*, 4 (2010), 548–559. 8

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d Tree GPU Raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), I3D '07, pp. 167–174. 2

[HTHG01] HADWIGER M., THEUSSL T., HAUSER H., GRÖLLER E.: Hardware-Accelerated High-Quality Filtering on PC Hardware. In *In Proc. of Vision, Modeling and Visualization 2001* (2001), pp. 105–112. 3

[KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003* (2003). 7

[LYS*10] LEE B., YUN J., SEO J., SHIM B., SHIN Y.-G., KIM B.: Fast High-Quality Volume Ray Casting with Virtual Samplings. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (nov.-dec. 2010), 1525 –1532. 3

[Mei02] MEIJERING E.: A Chronology of Interpolation: From Ancient Astronomy to Modern Signal and Image Processing. In *Proceedings of the IEEE* (2002), pp. 319–342. 3

[MKW*02] MEISSNER M., KANUS U., WETEKAM G., HIRCHE J., EHLERT A., STRASSER W., DOGGETT M., FORTHMANN P., PROKSA R.: VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), HWWS '02, pp. 137–146. 2

[ML94] MARSCHNER S. R., LOBB R. J.: An Evaluation of Reconstruction Filters for Volume Rendering. In *Proceedings of the conference on Visualization '94* (Los Alamitos, CA, USA, 1994), VIS '94, IEEE Computer Society Press, pp. 100–107. 8

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K.: An Advanced Volume Raycasting Technique using GPU Stream Processing. In *Computer Graphics Theory and Applications* (2010), pp. 190–198. 3, 7

[OHL*08] OWENS J. D., HOUSTON M., LUEBKE D., GREEN S., STONE J. E., PHILLIPS J. C.: GPU Computing. *Proceedings of the IEEE 96*, 5 (may 2008), 879–899. 1

[RDK*98] RIXNER S., DALLY W. J., KAPASI U. J., KHAILANY B., LÓPEZ-LAGUNAS A., MATTSON P. R., OWENS J. D.: A Bandwidth-Efficient Architecture for Media Processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), MICRO 31, IEEE Computer Society Press, pp. 3–13. 2

[Sag94] SAGAN H.: *Space-Filling Curves*. Springer-Verlag, 1994. 4

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph. 27* (2008), 18:1–18:15. 1

[SH05] SIGG C., HADWIGER M.: Fast Third-Order Texture Filtering. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, pp. 313–329. 3

[SHZO07] SENGUPTA S., HARRIS M., ZHANG Y., OWENS J. D.: Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), GH '07, Eurographics Association, pp. 97–106. 6

[TDR10] TCHIBOUKDJIAN M., DANJEAN V., RAFFIN B.: Binary Mesh Partitioning for Cache- Efficient Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2010), 1–14. 3

[Uns00] UNSER M.: Sampling-50 years after Shannon. *Proceedings of the IEEE 88*, 4 (apr 2000), 569 –587. 3

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001* (2001), Chalmers A., Rhyne T.-M., (Eds.), vol. 20, Blackwell Publishers, Oxford. 2