Efficient Memory Management for Uniform and Recursive Grid Traversal

Tomasz Toczek and Stéphane Mancini

Abstract This chapter presents the usefulness of predictive and adaptive caching methods for the traversal of both uniform and recursive 3D grid structures. Recursive data structures are used in several image processing kernels and their efficient management is one challenge to save silicon area and reduce the power consumption due to the data transport. The described architectures greatly reduce the needs in term of bandwidth by exploiting the spatial and temporal locality of memory accesses during ray shooting in uniform and recursive grids. To maximize the cache efficiency, the original kernel is transformed to a "phase locked" ray-packet based propagation algorithm. Our results show that well-suited caching strategies can indeed yield significant performance gains during the traversal of both uniform and hierarchical grids. This emphasizes the relevance of semi-general purpose multi-dimensional predictive caches.

1 Introduction

The management of high quantities of data is a challenge for many digital systems. This problem is getting more and more complex with the increase of the quantity of memory embedded in digital integrated systems such as System on Chip (SoC). As an example, the International Technology Road-map for Semiconductors Consortium plans that memory will occupy 90% of a circuit in the next years. Then, to alleviate the well known memory wall, it is mandatory to provide efficient memory hierarchies that are optimized together with a cache friendly optimization of applications.

T. Toczek (✉)
GIPSA-lab, INPG-CNRS, 961 rue de la Houille Blanche Domaine Universitaire-B.P. 46, 38402, Saint Martin d'Heres, France
e-mail: tomasz.toczek@gipsa-lab.inpg.fr

G. Gogniat et al. (eds.), *Algorithm-Architecture Matching for Signal and Image Processing*, Lecture Notes in Electrical Engineering 73,
DOI 10.1007/978-90-481-9965-5_2,

Several image processing algorithms are using multi-resolution images to perform tasks such as vision [4], video compression [3] and 3D rendering [1]. Multi-resolution images are used for vision algorithms to integrate some global information at the pixel level: the results at the low resolution are used to constrain the computations at the more detailed levels. To compress video, the motion estimation step also benefits from a multi-resolution pyramid of the input images: the coarse grain motion vectors at the low resolution are used to guide the computations at higher resolutions, preventing the algorithm to "fall" in some local minima. Mip-Map images are used in real-time 3D rendering to apply textures at a level of resolution that fits the raster scan sampling of the scene triangles. These multi-resolution textures allow to speed-up the rendering and increase the quality of the rendered images by preventing subsampling (aliasing). The efficient management of multi-resolution and recursive data structure is one of the challenge to design embedded systems for image processing.

Multi-resolution 3D grids are widely used for applications such as realistic rendering ray-tracing, medical visualization, volume rendering and tomographic reconstruction. These applications extensively use the *ray shooting* kernel to simulate the propagation of light in a 3D volume. Ray shooting based algorithms are widely considered as both computationally and bandwidth hungry. They have however the advantage of producing high quality results while being conceptually simple, thereby being a good example of what modern architectures may be used for. The principle is to compute ray paths through diversely structured scenes.

In this chapter we will focus on the traversal of both uniform and recursive grids, which can be seen as either space indexing means or direct volume representations. On the one hand, space indexing means are used to store a primitive-based scene (typically a set of textured triangles) in an easily traversable structure, commonly known as an acceleration structure (AS). The AS traversal returns a list of primitives or even a more complex sub-scene to which some further computations are performed. Examples of this approach include SAH-based *kd-trees*, bounding box hierarchies, and so on [7]. Direct volume representations, on the other hand, involve partitioning the space into small cubic fragments called voxels, and storing a meaningful value for each voxel, such as color, density, and so on. This is especially useful in medical visualization and imaging. With no loss of generality, we choose the later direct volume representation as our performance case study.

This chapter demonstrates the usefulness of the nD-AP Cache architecture (n-Dimensional Adaptive and Predictive Cache) [12, 15] and the associated methodology for some ray-shooting based applications. The results demonstrate the versatility of the nD-AP Cache, which performs well for the uniform as well as the recursive grid traversal. To tackle the later aspect, we used a set of small communicating nD-AP Caches to cache part of the scene tree.

To fit the prediction mechanisms, the original ray (line) propagation kernel needs to be transformed to exhibit more temporal and spatial locality. That for a "phase-locked" ray-packet based propagation algorithm is proposed. The idea is to enable a virtual loop to synchronize the propagation of a set of rays. Hence, this transformation improves the very low data-reuse ratio of the original iterative algorithm for which a grid cell is traversed only once.

The proposed memory hierarchies and "phase-locked" propagation algorithm, both for uniform and recursive grids, are validated by performance measures performed both on an emulation platform and through CABA (Cycle Accurate, Bit Accurate) simulations. The experiments lead to some improvements of the cache features so as to adapt the nD-AP Cache to the ray shooting kernel and any processing with similar patterns of memory references.

In Sect. 2, we briefly describe some of the hardware designed so far for memory management for ray shooting. In Sects. 3 through 6, we present our architecture, which includes the above mentioned cache and traversal pipelines. Finally, we discuss its performances in Sect. 7 and conclude in Sect. 8.

2 State of the Art

2.1 Dataset Traversal

The iterative parametric methods for scene traversal are the most popular ones. They are conceptually easy and their implementation is often efficient. Most of them are variants of the DDA (Digital Differential Analyzer) algorithm adapted to ray tracing [2], which parametrizes the ray and performs all the computations in the parameter space thereafter. Figure 1 illustrates the variables used by the DDA on a 2D example.

The algorithm iterates on the cells of a uniform grid, storing the parameters of the intersection between the ray and the yet-uncrossed cell faces' planes along each axis (called t_x and t_y in Fig. 1). In 3D, it comes the next cell is the neighbors of the current one, sharing the face corresponding to the smallest of the (t_x, t_y, t_z) parameters (that is, t_y in our example). This smallest parameter t_l (where $l \in \{x, y, z\}$) is updated by being added Δt_l, the parameter difference between the intersection points of the ray and the two faces of a cell orthogonal to a given axis l. In our example, it comes the next cell is the one just above the dark gray one, and the new face intersection parameters are $(t'_x, t'_y, t'_z) \leftarrow (t_x, t_y + \Delta t_y, t_z)$. The Δt_x and

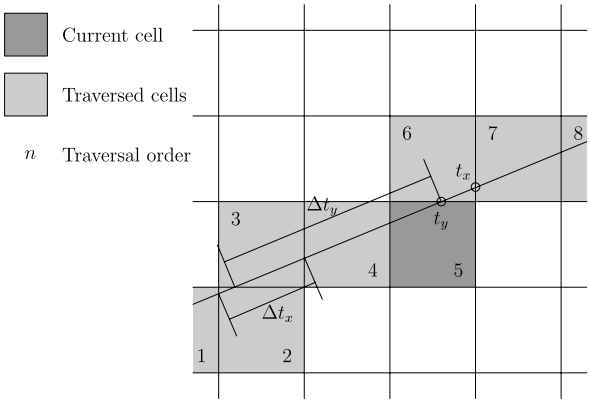


Fig. 1 Geometrical meaning of the variables used by the DDA algorithm

Δt_y parameters are computed once for all for a given ray at the initialization on the DDA algorithm. The resulting traversed nodes are shown in light gray in Fig. 1.

It should be noted that the DDA algorithm can be adapted to use projective geometry [14], which permits a higher traversal accuracy, and is a good substitute for floating-point arithmetics. It is the method we used in the architectures described hereafter.

When performing ray casting, the contribution[1] of each traversed cell is taken into account for the resulting pixel value computation. This is called compositing, and can be implemented in a variety of ways; for our tests, we used voxel-based volume rendering, considering each voxel as having uniform density, and integrating this density along the ray as a compositing rule. We could have used virtually any front-to-back compositing method instead. Also, the compositing unit can be turned into a primitive intersection test unit when the grid is used as a space indexing structure instead of a direct volume representation.

Parametric methods are quite versatile, and very similar techniques can be used for recursive grid traversal [9], the most well known special case being the traversal of octrees [20].

Amongst the non-parametric traversal and compositing methods, a very visually satisfying one consists in sampling the volume along the rays at regular intervals [8, 19]. It is especially efficient in the case of regular grids, where an element can be accessed in constant time. Aside from the oversampling/undersampling issues that may arise, this kind of approach tends to perform poorly quality-wise when used for the projection step of iterative tomographic reconstruction.

2.2 Memory Management

Most of the ray shooting dedicated hardware design with an emphasis on efficient memory management was done in the field of volume rendering. This can be explained by the fact that volume rendering naturally involves very large data sets, up to 1024^3 grids with recent medical appliances for instance. This is why memory bandwidth is likely to become the performance bottleneck of such systems. Therefore, most memory management strategies put a strong emphasis on data reuse and access locality through diverse means.

The cache efficiency of software systems can be optimized thanks to multi-threaded software [6, 11]: each thread deals with a small set of rays and the speed-up comes from the fact that some grid data used by each thread are in the cache memory. This implies rays are shot by coherent packets. Multi-threading based techniques is limited by task context switch, cache trashing and by the available computing power.

[1] Which may be emitted or re-emitted light (rendering), density (PET reconstruction), attenuation (X-ray based reconstruction), ...

The *Cube* series is an example of regular-grid-sampling-based volume rendering hardware designed to spare this very bandwidth. While *Cube-3* [18] is a regular ray parallel ray tracing architecture, *Cube-4* [19] is based on object-order ray-tracing. The *Cube-4* hardware is designed so as each voxel is fetched exactly once per frame from the central memory. Therefore, it is optimally efficient in terms of memory bandwidth usage, if we admit that every voxel accounts for each frame. However, *Cube-4* has a number of severe limitations, one of them being a scene size limited to 256^3 equally sized voxels. Moreover, the very principle of object-order ray tracing upon which *Cube-4* was built makes perspective rendering implementation impractical; that is why VolumePro, a commercial implementation of Cube-4, only supports isometric rendering. Several other problems were underlined by [17].

VoxelCache [8] is a cache specifically crafted for sampling based ray casting, as well. It is small enough to be implemented on reconfigurable hardware. It uses only a single external memory bus, but has an internal 8 memory bank organization. This makes possible to fetch a tri-linearly interpolated sample every cycle. VoxelCache also has a prefetching mechanism, requiring that beams of 4×4 coherent rays are being shot. Despite the fact it was designed for uniform grids, VoxelCache was successfully used for full-octree-based volume rendering as well [22]. This however required the use of off-chip SRAM to keep the performances high, and the caching strategy was shown inefficient for large scenes due to cache trashing. On the contrary, we tried in our approach to use as much as possible inexpensive components found on most prototyping boards, while focusing on arbitrary large sparse octree-like acceleration structures, much more flexible than full octrees.

Since hardware systems benefit of uniform memory accesses, a solution is to allow only parallel rays, possible rendering a given volume slice by slice to produce an illusion of perspective. Such a strategy underlies volume rendering on commodity PC textured rasterization hardware [10, 21]. Some parallel rays are sampled at the same interval to provide "plane parallelism". Perspective volume rendering is then simulated by a perspective transformation of the resulting image. This solution is very popular for visualization because it is fast but it suffers of too low accuracy for tomographic image reconstruction.

More recently, programmable graphics hardware has been used for volume rendering. It is quite suitable for brute force ray casting through uniform grids [16]: the built-in texture cache can handle 3D scenes efficiently, and the Single Instruction, Multiple Thread (SIMT) programming model is suited for casting rays in beams, ensuring reference locality. There are also other ways to perform volume visualization on GPGPUs, for instance through the well known marching cubes algorithm [13], which converts a volumetric scene into polygons before displaying it. Nowadays, GPUs can afford not only to display the generated polygons, but also to build them from the volumetric data in the first place [16], which used to be done on the CPU. The drawbacks of GPUs include a high power consumption, small per-core on-chip memory quantities making them unsuitable for recursive algorithms with high stack-space requirements, and a loss of efficiency in case of diverging code paths between threads, bad load distribution, and so on. Implementations of ray tracing algorithms are rather prone to such pitfalls.

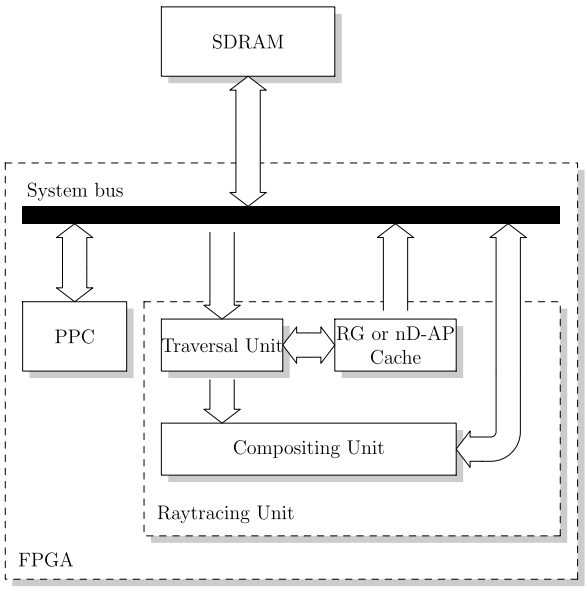


Fig. 2 An overview of a complete rendering system

3 System Architecture

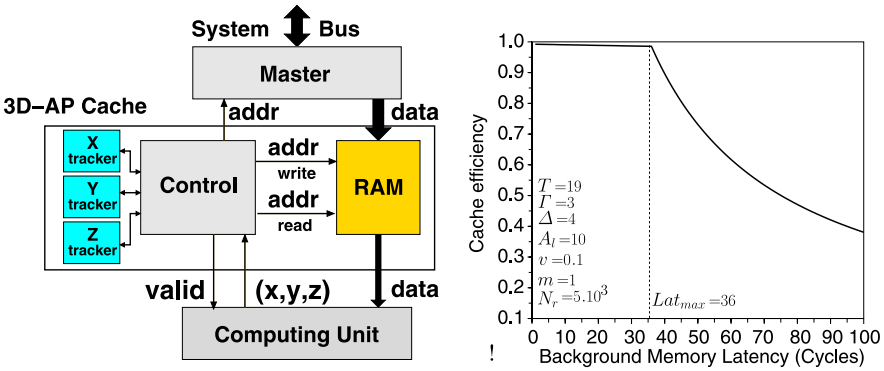
The architecture we propose is composed of two main parts: an adaptive and predictive cache for either uniform or recursive grids and a traversal unit, capable of determining the sequence of grid cells traversed by each ray of a coherent beam.

The generated sequences are meant to be communicated to a compositing unit. Since we chose visualization as an application, once a ray of the beam ends, its accumulated value may be written to a frame buffer. Figure 2 presents an overview of a whole rendering architecture as implemented on a prototyping board.

The traversal unit and the cache were designed and synthesized with the Xilinx Virtex 4 technology as a target. Depending of the number of units and the size of the FPGA, the results were obtained either by actual on-board runs, or CABA simulations.

4 The *n*D-AP Cache

The 3D-AP Cache is an instance of the *n*D-AP Cache which aims at caching multi-dimensional data as originally described in [15]. Also it performs pre-fetching by estimating the future zones of data the processing unit is supposed to fetch. As shown Fig.3(a), the *n*D-AP Cache provides a virtual interface to the computing unit that issues multi-dimensional indexes in the data structure. The *n*D-AP Cache performs both the mappings between indexes and the external memory addresses and the internal memory addresses.



(a) The 3D-AP Cache architecture

(b) The 3D-AP Cache performance model

Fig. 3 The 3D-AP Cache aims at performing prefetching in multi-dimensional arrays

The pre-fetching mechanism relies on a tracking of indexes on each dimension. The trackers try to estimate the zones of indexes the computing unit may fetch from an analysis of the past fetches and a prediction model. The *n*D-AP Cache architecture is modular and several kinds of trackers are available. In the following, we consider a first order SC tracker which prediction model makes the hypothesis that the indexes of the fetch sequence evolve as a compound of a fast displacement around a low speed displacement. The trackers compute the estimated means of the indexes on each axis and request the control unit to grab a new zone of data when the estimated means cross a guard zone. This mechanism ensures that the new zone will be uploaded before the references reach the cached zone boundaries as demonstrated in [12] (see Fig. 3(b)). This prefetching is enabled when the 3D-AP Cache is tuned in such a way that the cut-off frequencies of the estimators and the different thresholds (T, Γ, Δ) fit the average speed of the indexes (v), their local amplitude (A_l) and the background memory latency (see [12] for more details). Above a threshold latency the cache efficiency drops but the cache parameters may be set to fit a given latency.

5 Uniform Grids

5.1 Uniform Grid Traversal

The traversal pipeline for regular grids is shown in Fig. 4. The RCPG-U unit (Ray Casting in Projective Geometry-Uniform grid) is made of a traversal unit connected to a 3D-AP Cache. The Line integral unit implements the compositing processing to compute sinograms in tomography applications. The traversal unit gets parameters from the PowerPC processor and manages the phase-locked propagation of a beam of rays. As can be seen, the memory references to the volume do not depend on

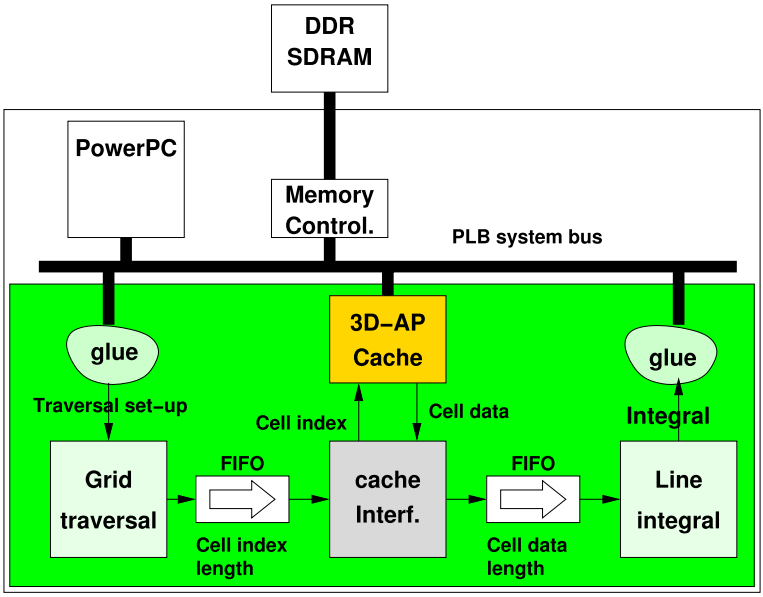
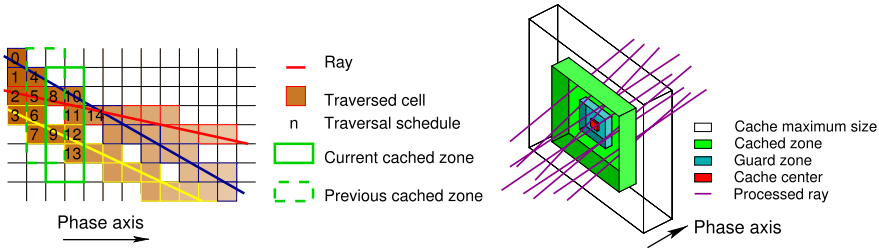


Fig. 4 RCPG-U pipeline and cache interface

the fetched data, and could theoretically be determined ahead of time. We chose nonetheless not to use this specificity, not to increase the hardware complexity of the traversal unit. This allows us to test the behavior of the cache while providing it with no clues regarding the future accesses.

In order to obtain good performances even if caching very small parts of the volume, we rely on "phase-locked" traversal. The phase-locked propagation enables a virtual loop over the traversed cells. Indeed, as shown in Fig. 5, the RCPG-U



(a) The phase-locked propagation (slice)

(b) A slice of volume is cached along the phase axis

Fig. 5 The phase-locked propagation of a beam of rays improves the spatial and temporal locality of grid traversal

algorithm is synchronized over a set of rays to propagate them together along a main direction, orthogonal to a phase plane. This direction is collinear to the major axis which is the closest to the overall direction of the beam of rays, and is pre-computed at the same time the initial states of the rays of the beams are generated (typically, by a hard or soft processor). For each ray, a step of the RCPG-U algorithm returns the next cell to cross. This is iterated while the resulting cell remains in the phase plane. When all the rays of the set are out of the phase plane, then the phase is updated to the next one. The process loops until all the rays exit the volume. The phase acts as a virtual loop, the innermost one being the ray index. Since it is known that the *phase* moves in only one direction and that a lot of consecutive accesses will request cells sharing the same phase, we can afford to cache only a very narrow slice of the scene along the phase axis. Along the two other axes, the cached zone needs to be just broad enough to contain all the rays of the beam.

5.2 Uniform Grid Caching

The experiments on Ray Casting have raised the need of more efficient tracking mechanisms and new cache behaviors. To manage different classes of fetch sequences, multi-mode trackers implement together several tracking mechanisms that may be dynamically selected. The mapping of fetch indices to the internal cache addresses can also be dynamically chosen to fit different sizes of cache. The trackers are joined together to allow more complex cache zone displacements: a displacement request from a tracker makes the other trackers to center on their estimated center. Furthermore, the user module can now select the priority of misses over cache updates.

The dynamic selection of the priority of misses is efficient especially when the misses are faster than the uploading of new zones into the cache. At a first sight, updates of the cached zone have a higher priority than misses because it prevents the user module to always request misses if the trackers are too slow. But fetches can evolve differently on each dimension and misses can have different priorities depending on the cache geometry. At the time the user module requests a high priority miss, the cache update is interrupted, the miss is served and the update continues over.

The phase locked propagation of rays benefits from these improvements. The multi-mode tracking is necessary because the virtual loop on the propagation direction can be efficiently tracked by a linear tracker while the other dimensions are tracked by statistical trackers. The misses have high priority along directions perpendicular to the phase direction and have low priority along the later. Indeed, the phase increases (or decreases) uniformly faster than the other directions in the average case. The worst case is when the rays have a 45-degree angle with all or some of the main axes.

6 Recursive Grids

6.1 Caching the RG Data Structure

We generalize the above described pipeline and cache to the traversal of a generalization of octrees [5] we call recursive grids (for the lack of an established name). As their name somewhat implies, recursive grids are sparse hierarchical structures, and therefore induce a dependency between the memory access sequence and the data fetched from memory. Predictive prefetch mechanisms become therefore unavoidable in order to reach acceptable performances.

After a description of recursive grids, we will show how 3D-AP Caches can handle them, and which access patterns should be used to maximize caching efficiency.

6.1.1 Recursive Grids

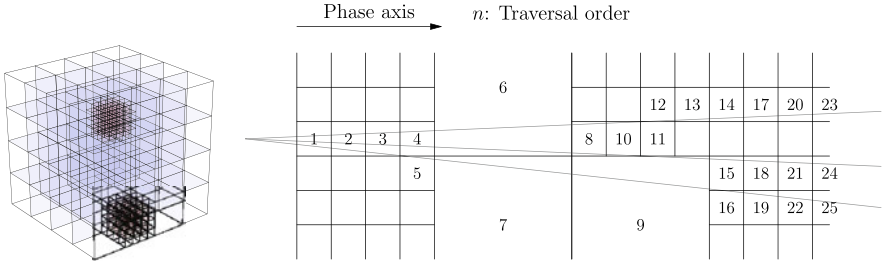
A $2^n \times 2^n \times 2^n$ recursive grid is a tree with the following characteristics:

- each node is a cube
- each node has either a datum, or 2^{3n} equally-sized children

As a consequence, every node of the recursive grid has the same size as any other node on the same depth. We call the leaf nodes voxels. Figure 6(a) shows a $4 \times 4 \times 4$ recursive grid, with two non-voxel children at the root node. It is clear that, under our formulation, octrees are $2 \times 2 \times 2$ recursive grids.

$2^n \times 2^n \times 2^n$ recursive grids, especially for "moderately large" values of n such as 2 or 3, hold several interesting properties few other hierarchical acceleration structures do. While still having the benefits of sparse hierarchical structures, they also offer more regularity than most of them, allowing more efficient caching. It can be noticed that recursive grids are a subset of adaptive grids [9], and hence inherit of most of their advantages when it comes to their traversal.

For our tests, we chose to encode a $4 \times 4 \times 4$ recursive grid by an array of at most 32768 nodes, the node 0 being the root. The exact coding of a node is itself



(a) An example $4 \times 4 \times 4$ recursive grid

(b) Expected voxel access order for a phase-locked tracing of a beam of 3 rays in a 4×4 recursive grid (2D example)

Fig. 6 Recursive grids and their phase-locked traversal

that of an array of 64 words of 16 bits, each coding for a child (15 bit datum and a 1 bit flag, determining if the child is subdivided or not). The datum a child contains is either a density, or index of the child node.

6.1.2 RG Cache

The Recursive Grid (RG) Cache, described in Fig. 7, aims at caching a part of the recursive grid by exploiting the spatial coherency of references. Furthermore, it provides a virtual interface to the processing unit. This means that the processing unit issues a 3D coordinate (x, y, z) together with a resolution level and the cache provides the corresponding datum, preventing the processing unit to manage the tree structure. The RG cache uses the nD-AP Cache as a basic block.

The proposed strategy is to cache each level of resolution with an nD-AP Cache and to perform prefetching in the tree. Indeed, when a reference occurs, it is likely that the next reference is at a close coordinate, either at the same, upper or lower resolution. Each nD-AP Cache is in charge of tracking references at a resolution and the neighboring ones.

The tree manager (TM) unit returns parts of the scene requested by the nD-AP Caches and maintains a coherent state of the caches at different resolutions. Each time a cache requests a part of the scene, the TM reads the corresponding data at the upper level to determine if the requested block is a leaf or a child node. In the later case, the TM fetches the data at the obtained address to fill the nD-AP Cache. As a consequence, the nD-AP Cache is slightly modified to allow the TM to read an nD-AP Cache concurrently with reads at the processing unit interface. Also, the cached zone at level n has to be inside the cached zone a level $n-1$ to maintain cache coherency. Without this constraint, when the cache n would request a part out of the zone in the $n-1$ cache, it would be too slow to get the data by traversing the tree from the root node.

Each of the cache is optimized to manage data at its level of resolution. The size of embedded cache memories fits the level of resolution to save space. For example, because the level 1 cache contains only $4 \times 4 \times 4 = 64$ data, it doesn't need trackers and has a simplified control management.

6.1.3 Improving Reference Locality

Just like we did for uniform grid traversal, we use a phase-locked propagation principle in order to keep the accesses coherent and the cached zone minimal.

More specifically, once the phase axis chosen, we propagate rays in a way that ensures the cell accesses during traversal are ordered by their coordinate on the phase axis. An example of such an access sequence is given in Fig. 6(b). Section 6.2.2 gives more details as how this can be implemented. Just like previously, the cached zone should be narrow along the phase axis.

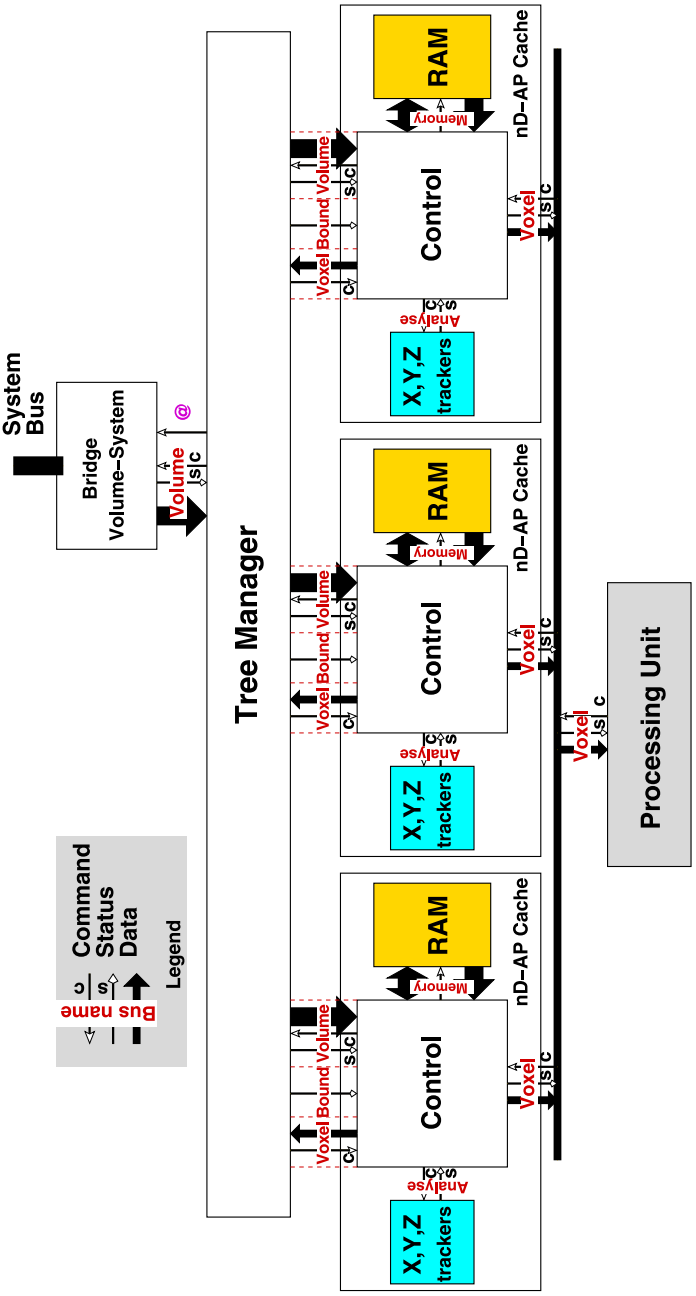


Fig. 7 Recursive Grid (RG) Cache

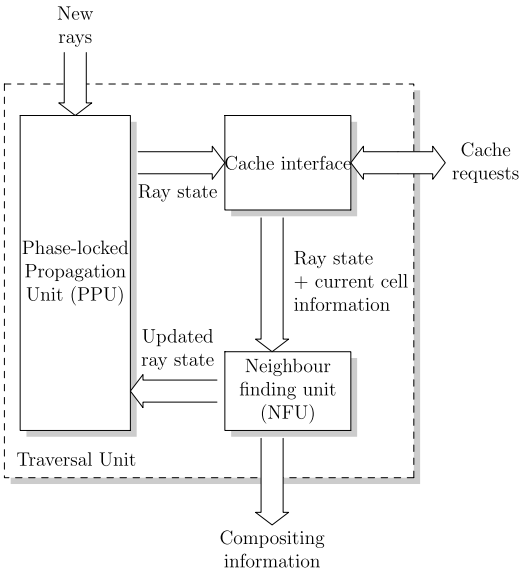


Fig. 8 Architecture of the recursive traversal unit

6.2 Recursive Grid Traversal

We implemented a hardwired traversal unit for efficient ray shooting through recursive grids. It works on beams of up to 256 rays. Figure 8 shows its structure. Its three main parts are:

- the phase-locked beam propagation unit, which determines the order in which grid nodes are fetched so as to minimize the probability of cache misses; do to so, it holds the states of all the rays of the current beam in such a way it is able to ensure they all propagate with the same speed along the phase axis
- the RG Cache interface, which performs cache requests while pipelining ray parameters linked to the request
- the neighbor finding unit, which determines the next node a ray should access and its updated parameters, based on the response from the cache and the former ray parameters

The cache interface simply contains two ray state holding FIFOs, which minimize the impact of small pipeline bubbles. Therefore, in the rest of this section, we will focus on the two other components.

6.2.1 Neighbour Finding Unit

We use a slightly modified version of the DDA algorithm for neighbor finding, in a fashion very similar to that presented in [2]. As we need to be able to vertically traverse the tree as well, we adapted the principles exposed for octrees in [20] to $2^n \times 2^n \times 2^n$ recursive grids, by considering each of our nodes as an n-level perfect

```
1  ray_state.r⃗ ← unitary direction vector of our ray
2  ray_state.t ← current cell entry point parameter
3  ray_state.(t_x, t_y, t_z) ← border intersection parameters
4  ray_state.(Δt_x, Δt_y, Δt_z) ← parameter increments
5  ray_state.(p_x, p_y, p_z) ← current cell absolute position
6  ray_state.depth ← depth of the current cell
7  max_depth ← maximum depth of the tree
```

Listing 1 Variables characterizing a ray state

octree. On a side note, this approach works for adapting pretty much any algorithm working on octrees to recursive grids, and without increasing its asymptotic cost.

To sum up everything, let us consider that a ray propagation state is fully characterized by the variables of Listing 1. The "parameters increments" are the differences between the parameters of the intersection points between the ray and two opposite faces of our cell, for each of the three possible such pairs of faces. The absolute position of the current cell is given in units corresponding to the size of cells located at a given maximum depth (the constant max_depth). If this depth is 6, for instance, the maximum detail level of a $4 \times 4 \times 4$ recursive grid will be the same as that of a 4096^3 uniform grid.

Let's suppose without loss of generality that our ray propagates in the positive direction along each axis.[2] Our neighbor finding algorithm is then summarized by the pseudo code of Listing 2. Basically, our traversal unit advances to the next cell if the current cell is a leaf (this takes one cycle), of dives further if it is an internal node (n cycles for 2^{3n} grids). Figure 9(a) presents the unit's organization. As the diving and advancing modes are mutually exclusive, there is hardware reuse between them which does not appear on this figure for the sake of clarity.

One should pay attention that the input and output data of the neighbor finding unit may grow moderately large depending on its exact coding. What call *ray propagation state* a structure composed of the variables seen in Listing 1, at the exception of the direction vector of the ray (which does not matter for the traversal assuming all the other variables are known). Assuming max_depth $= 5$, $n = 2$, and 32 bits per parameter, we need 260 bits per ray state. For each ray, those parameters need to be initially computed from the ray geometry before they are fed to the propagation unit. It involves obtaining the intersection points parameters between the ray and each of the faces of the root node cube.

6.2.2 Phase-Locked Ray Beam Propagation

The phase-locked propagation unit (PPU), as shown in Fig. 9(b), manages the indexes of rays to enter the propagation pipeline. The indexes of active rays are stored

[2] Indeed, if it is not the case along one or more axes, we can bring ourselves back to the case where it is by taking as absolute cell position the one's complement of the actual cell position along those axes. Of course, the "correct" position must still be used for the memory accesses. This strategy is suggested in [20], where the reader may find extensive detail of such an approach.

```
1  if (current cell is a voxel) then
2        send ray and cell data to the compositing unit
3
4        // We need to find the next cell (on the same depth or above);
5        // determining which direction the next cell is:
6        k | t_k = min(t_x, t_y, t_z)
7
8        // When exiting a cell, undiving as far as necessary:
9        while p_k[(h-1)..(h-n)]='1..1' with h=n*(max_depth-depth)
10             if depth > 0
11                   depth ← depth - 1
12             else
13                   traversal is over for current ray
14             for m in 0 to (n - 1)
15                   h ← n * (max_depth - depth) - (m + 1)
16                   foreach l ∈{x, y, z}
17                         if p_l[h] = '1' then
18                               t_l ← t_l - Δt_l
19                         Δt_l ← 2 * Δt_l
20
21       // Finally, advancing to the next cell:
22       p_k ← p_k + 2 ** (n * (max_depth - depth - 1))
23       t_k ← t_k + Δt_k
24 else
25       // We need to dive further
26       for m in 0 to (n - 1)
27             h ← 2 * (max_depth - depth) - (m + 1)
28             foreach l ∈{x, y, z}
29                   Δt_l ← Δt_l / 2
30                   t_l ← t_l - Δt_l
31                   if t > t_l then
32                         t_l ← t_l + Δt_l
33                         p_l[h] ← '1' // h^th bit of p_l to 1
34                   else
35                         p_l[h] ← '0' // h^th bit of p_l to 0
36       depth ← depth + 1;
```

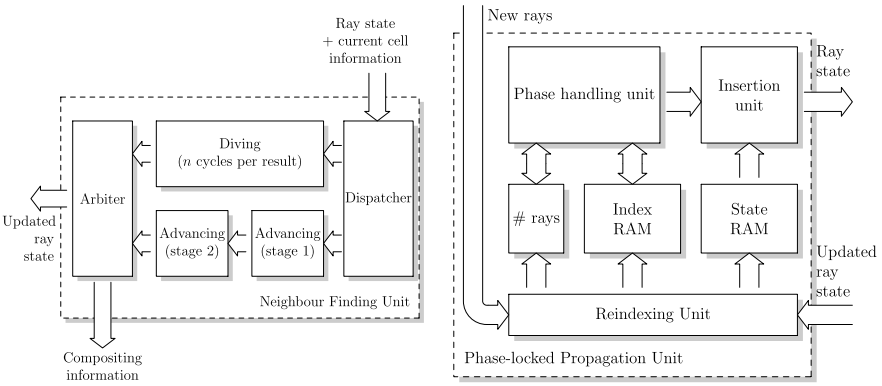
Listing 2 Neighbour finding algorithm

in the "Index RAM" in a way to manage efficiently the synchronization of propagation along the beam phase. Hereinafter a "ray" stands for its index in the "State RAM". The "Index RAM" is divided in several ranges to manage the phase synchronization over the different resolutions. The diving of rays being one of the most tricky behavior to deal with.

The beam phase is memorized in the PPU. It is updated when all the ray phases[3] are further than the beam phase. To that end the memory is divided into in-phase rays and out-of-phase rays.

An in-phase ray is sent to the "Insertion Unit" to update its next state. On its way back, the ray is either still in-phase, if the propagation occurred on an other axis, or out-of-phase in other cases. When all the rays are out-of-phase, the PPU increments the beam phase and swaps the out-of-phase rays into the in-phase. This is actually done when all the rays sent to the propagation pipeline are back. That for, the number of processed rays is counted. Rays exiting the scene are counted back but not inserted in the "Index RAM".

[3] A ray phase is the ray coordinate along the phase axis.



(a) Neighbour finding unit (NFU) (b) Phase-locked Propagation Unit (PPU)

Fig. 9 Architecture of the two main elements of the recursive grid traversal pipeline

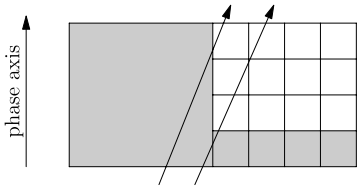


Fig. 10 In some circumstances, diving may out-phase some of the rays of a beam, while keeping others in-phase (*gray voxels* are on the beam phase)

Also, the phase synchronization have to be performed on all the levels of resolution. So, the "Index RAM" in-phase and out-of-phase parts are again divided in sub-parts for each level of resolution. Higher depth rays are first sent to the propagation pipeline and the lowest one are then sent, until there are no more in-phase rays at any resolution.

At last, rays diving into a higher resolution have to be sorted according to the beam phase. Indeed, the diving may cause a ray phase to be higher than the current one because the entry point in the higher resolution node may be anywhere on the child border (see Fig. 10). To speed-up the phase sorting, rays are stored in the "Index RAM" according to their relative phase in a node. This relative phase is simply the n bits starting at the $n(\text{max\_depth} - d)$ bit of the ray phase, where d stands for the current depth of the ray.

Hereof, we conclude that the "Index RAM" is a memory of $(\text{max\_depth} + 1).2^n.\text{max\_rays}$ words of $\log_2(\text{max\_rays})$ bits. A more tolerant phase synchronization allowing a 2^{dn} phase deviation would need a 2.max_rays "Index RAM" but it would increase the cache's memory.

7 Results

This section provides some results about the cache efficiency for a set of applications of the phase-locked ray tracing in uniform (RCPG-U) and recursive grids

Table 1 Area of the RCPG-U unit and 3D-AP Cache for a Xilinx Virtex IV technology. Percentages are relative to the V4FX60 device capacity. A group-level pipeline of 4 RCPG-U units shares a single 3D-AP Cache

	Logic cells	DFF	DSP48	BRAM (KB)
RCPG-U pipeline	1739 (3%)	996 (2%)	6 (4%)	9 (0.7%)
3D-AP Cache	1299 (2.5%)	365 (0.5%)	0	8 (0.7%)

(RCPG-R). We show that in both cases the cache performance highly depends on the geometry of the rays belonging to a beam. The distance between rays directly impacts the data-reuse ratio because a data is little reused when the rays do not cross the same voxels. At the opposite, the cache mechanism of the RCPG-U and RCPG-R systems are more efficient when a grid is visualize with a larger zoom because a voxel is crossed by the rays corresponding to neighboring pixels of the rendered image. The 3D-AP Cache shows to be more efficient when the RCPG-U unit is used to compute a sinogram. This later application computes the volume integral along each Line of Response (LOR) that connects a pair of detectors of a tomographic camera. The LORs of a sinogram form a 4D array. To increase the data-reuse ratio, the computation of the sinogram is split in a set of 4D sub-blocs. The performance of the RCPG-R system is the most difficult to measure because it also depends on the structure of the recursive grid. The RG Cache behavior depends both on the geometry of a beam of rays but also on the traversed levels. It happens that a few levels are traversed where the grid data is uniform. Then, the RG Cache efficiency may fall because the set-up time to initialize the traversal is higher but the total time to traverse the recursive grid is much smaller than a uniform grid traversal.

The RCPG-U system was implemented in a Virtex II Pro prototyping board and some details of its area occupancy and 3D-AP Cache performance are provided. The provided measures may be extrapolated to any FPGA or ASIC technology though. As a proof of concept, the RCPG-R is implemented in VHDL-RTL and the RG Cache performance is measured by simulation.

7.1 Hardware Complexity

7.1.1 Uniform Grid Traversal

The RCPG-U is designed in VHDL-RTL and implemented in prototyping board with a Virtex II Pro FPGA. The board runs at 30 MHz but this is not an issue as logic synthesis of the VHDL code for the Virtex 4 technology reports a clock frequency up to 200 MHz. A memory simulator allows to measure the 3D-AP Cache performance for different background memory configurations.

Table 1 provides the complexities of the RCPG-U pipeline and of the 3D-AP Cache. These results are obtained in fixed point arithmetic, with the bit widths set

to reach the accuracy needed by a tomographic reconstruction application. The 3D-AP Cache seems of the same complexity as the RCPG-U unit but a group-level pipeline up to 4 RCPG-U units shares a single 3D-AP Cache. Indeed, the RCPG-U unit has a pace of 3 to 4 clock cycles between each fetch. In this later configuration, the 3D-AP Cache occupies the third of the whole system complexity and the highest throughput is reached when the 3D-AP Cache releases a datum each clock cycle.

An interesting point is that only the BRAM size depends on the maximum size of the cached zone. The area of the control unit of the 3D-AP Cache is almost independent on the cache memory.

7.1.2 Hierarchical Grid Traversal

The RCPG-R unit is implemented in VHDL-RTL and has been validated has a proof of concept. The main objectives were to show in one hand that the phase-locked synchronization is efficient and on the other hand that the most complex part of the architecture are integrable. Table 2 gives the complexity of some parts of the RCPG-R. Most of the area is used by the index RAM and the ray state RAMs. Their sizes are directly linked with the quantity of rays in a beam and the levels of the recursive grid.

7.2 Cache Efficiency

Several criteria are used to measure a cache efficiency. In the following, we focus on the time performance and the cache efficiency is measured as the ratio between the number of fetches divided by the number of clock cycles to get all the data. Other measures such as the system bus occupancy, the re-use ratio and others are available but will not be discussed to gain in clarity. The measures performed by simulation or with the prototyping board are equivalent, but the prototype is faster.

Table 2 Area of some parts of the RCPG-R recursive grid traversal unit for a Xilinx Virtex IV technology

	Logic cells	DFF	DSP48	BRAM (KB)
Hierarchical Neighbour Finding	5639 (11%)	1474 (2.9%)	6 (4%)	0
Hierarchical Phase-locked Propagation	1035 (2.1%)	727 (1.4%)	0	14.6 (2.7%)
Tree Manager	2246 (4%)	286 (0.5%)	0	0

In order to study the effectiveness of pre-fetching, the efficiency is measured for a set of memory latencies. This later is the time to get a burst of data from the background memory. The latency includes the arbitration time of the system bus, the latency of the memory controller and the latency of the external memory device.

As a worst case hypothesis, we suppose that the latency is paid for each burst request (no pipeline). An update of the 3D-AP Cache is split in a set of bursts corresponding to lines in the x axis. Better results are achievable when the latency is paid only once for a 3D-AP Cache update.

The measures show that the pre-fetch mechanism is efficient for a large set of latencies depending on the application (visualization, sinogram computing), the viewpoint and the quantity of rays in a beam.

7.2.1 Cache Efficiency of the Uniform Grid Traversal

Visualization For the visualization application, ray casting is used to visualize a 3D grid on a 2D focus plane: each pixel of the resulting image is obtained by composing the voxels traversed by the ray issued from the pixel and passing through a viewpoint.

From Fig. 11 we clearly see that the pre-fetching mechanism allows to overcome the memory bottleneck: the efficiency is high before a latency threshold and

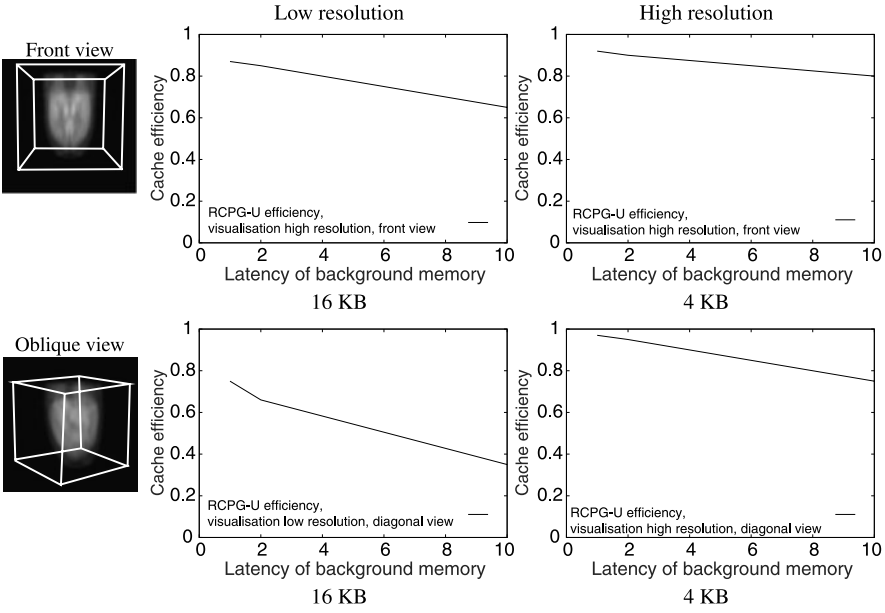


Fig. 11 RCPG-U visualization efficiency: the average 3D-AP Cache efficiency of the group level pipeline for the visualization application; The numbers provide the cache size

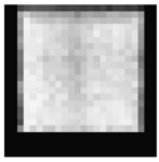


Fig. 12 RCPG-U visualization efficiency: the 3D-AP Cache efficiency for each beam of rays. The cache efficiency exceeds 90% for some tiles (white is 100%)

drops above this threshold. Figure 11 provides the average cache efficiency along the memory latency for two viewpoints and two resolutions. The computed images are divided in tiles of 12×12 pixels, to form beams of 144 rays: at low resolution there are 22×22 tiles and 44×44 tiles at the high resolution. The figure also provides the size of the cache memory. The background memory is 64 bit wide and a word contains a $2 \times 2 \times 2$ part of the volume.

The efficiency corresponding to the computation of each beam is provided in Fig. 12. Each block of that image corresponds to a beam and gives the efficiency at which are performed the fetches of all the voxels traversed by all the rays in the beam. A lower efficiency on the borders is due to the fact that these beams contain fewer rays and a small part of the volume is traversed when the rays hit the volume's edge.

The RCPG-U pipelines can be parallelized at two levels: within a group parallel level, some traversal units share a 3D-AP Cache, and at the cluster level, a set of group level pipelines is connected to a main 3D-AP Cache. In the later configuration, there is a cache hierarchy, with leaf 3D-AP Caches grabbing data from the main cache. Measures on a cluster-level parallel architecture show that two groups of pipelines sharing a higher level cache enables a 1.5 speed-up at the low resolution and 1.8 at the high resolution. The speed-up also depends on the view-point, the geometry of rays and the size of the tiles.

From these measure we conclude that the 3D-AP Cache allows to exploit the data-reuse and performs pre-fetching efficiently.

Sinogram Computing Because of a higher data-reuse it is more efficient to compute a sinogram than to visualize a volume, as shown in Fig. 13. The measures are

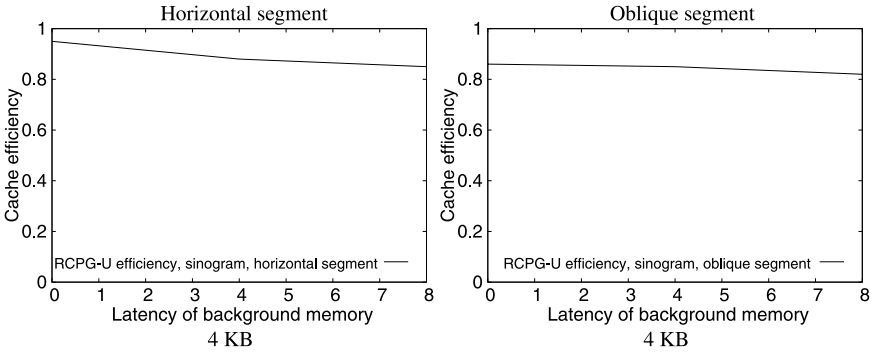
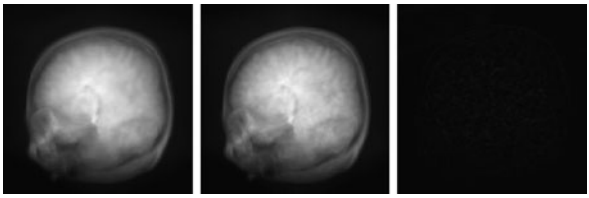


Fig. 13 RCPG-U sinogram efficiency: due to a high data-reuse ratio, the cache efficiency to compute a sinogram is higher



(a) Recursive grid rendering (b) Uniform grid rendering (c) Difference image

Fig. 14 Rendering of a recursive grid at 128 × 128 resolution; The PSNR is 46 db but uniformly distributed

performed to compute a sinogram simulating an "ECAT EXACT HR+" PET camera: it is a cylinder of 32 rings and each ring has 576 detectors. The RCPG-U unit is used to compute the integrals of the lines blending two detectors. The measures are performed in two scenarios: the start and end detectors belong to a single ring (horizontal segment) and in different rings (oblique segment). The rays to compute a sinogram are split in beams of 256 rays.

The pipeline has an efficiency higher than 80% with a typical background memory latency (3 to 4 clock cycles). The simulations show that a cluster of two group-level pipelines still has an efficiency about 85% and provides a speed-up of 1.9 compared to a single group-level pipeline. These very good results are due to the fact that the rays of a 4D tile are crossing together in a "tube" and their density is higher than in the case of visualization.

7.3 Cache Efficiency of the Recursive Grid Traversal

The RCPG-R architecture is evaluated by visualizing a reconstructed 256^3 MRI data from the PET-SORTEO database.[4] Since the data is provided as a uniform grid, a $4 \times 4 \times 4$ recursive grid is built by merging adjacent voxels which are about the same density (i.e., within 37% of the dynamic range). Since each node has 64 children, this criterion does not lead to severe losses in quality.

Figure 14 shows a render of this recursive grid performed by the RCPG-R unit. The resolution of the rendered image is 128 × 128. The quality is enough for a visualization application and the number of traversed cell is 6 times less.

Similarly to the regular grid traversal the RG-Cache efficiency increases with the image resolution because the data-reuse ratio is higher then. But, as some rays do not hit the highest depth in the recursive grid, the data reuse ratio of the intermediate depths is higher than the uniform grid ones, even for a low resolution image.

The plots in Fig. 15 show the efficiency to render a 128 × 128 image for different memory latencies. The efficiency is above 80% for a 4 clock cycle latency for most of the tiles and drops when the latency increases. The latency now corresponds to the clock cycles to wait before getting all the data of a $4 \times 4 \times 4$ sub-grid. Then the

[4] http://sorteo.cermep.fr/.

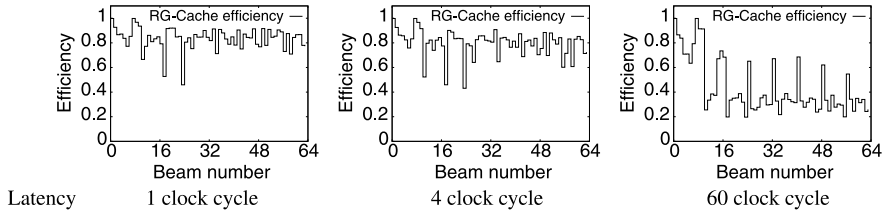


Fig. 15 RCPG-R visualization efficiency: the efficiency is plot for each tile of the rendered image

128 bytes (64 data) of the sub-grid arrive in 16 clock cycles thanks to a 64 bit width background memory.

These results show that the RG-Cache efficiently pre-fetches parts of the tree on time for a typical memory. Furthermore, the virtual interface to the cache prevents the computing unit from computing the effective addresses of the tree data. Hence, considering a latency of four, the RG-Cache allows to fetch a datum each 1.25 clock cycle in average. Eventually, the recursive grid traversal needed six times less memory references that the uniform one, which results in as much speed-up.

7.4 Discussion

Comparison with other solutions is not straightforward because all the architectures we can find in the literature implement a grid sampling algorithm, whereas the proposed algorithm implements an exact grid traversal ray casting. Furthermore, most of the systems implement early-ray termination, which is useful for visualization because unseen voxels do not need to be rendered, but our algorithm performs a complete grid traversal. Early ray termination could be added to our system but it is likely that stopping some rays would lead to too-much instability for a low global gain.

Reference [8] reports simulation results of a cache designed for a volume rendering hardware architecture. For each sample, the pipeline fetches 8 data from the VoxelCache which is a full associative cache that contains blocs of voxels. The VoxelCache holds 512 lines each of 64 voxels to make a 32 KByte memory. The simulation results of [8] show a 90% pipeline utilization but the size of the rendered volume is limited to 128^3 voxels because the VoxelCache is trashing when it cannot hold all the voxels along a line.

Our solution reaches an equivalent throughput with 4 times less memory and without associative memory: it is space-saving and has a better usage of FPGA resources. Also, a high level of parallelism is allowed, depending on the geometry and the density of rays. At last, the 3D-AP Cache and the "phase locked" RCPG allow any size of grid and the system is scalable.

To our knowledge there is no equivalent to the recursive grid traversal strategy presented in this chapter. The closest hardware architectures presented in the literature addresses full octrees [22]. The later are different from sparse octrees in the

way that nodes are always data and never pointers to a higher resolution. They are much like multi-resolution volumes and their traversal is done at a given resolution. Hence, the traversal of full octrees looks like a regular grid traversal, the resolution staying constant along a ray.

7.5 Improvements

Getting higher performances could be done by some improvement of the presented concepts and some new strategies may overcome some bottlenecks.

First, many of the architecture's parameters could be set to fit a particular use of the architecture or to take into account some integration constraints. As an example, the ability to pipeline the burst requests on a system bus would improve the performances dramatically. The placement of the volume data in the memory is also of high importance to reduce the relative latency by enabling longer bursts. As an example, a memory data word may contain a sub-volume and larger parts of volume may be stored at some contiguous addresses. The level of cluster-parallelism could be increased by tuning the bandwidth between the higher level cache and lower level caches. This should allow a better overlap of the computation with the update of the lower level caches.

The recursive grid traversal could be improved by a better synchronization of the different cache depths. Due to the recursive grid structure, the $n-1$-depth cache stores the pointers needed to update the n-depth cache and the two levels have to agree on a common zone to keep the cached tree coherent. At the moment, the $n-1$-depth cache constrains the n-depth cache and a back-pressure mechanism would allow a better pre-fetching mechanisms along several levels of the grid.

The automatic setting of the cache parameters is also a major challenge. Some of the cache parameters should be set dynamically to get higher performances, especially to manage special cases such as small set of rays at the border of the image. Currently the cache parameters are set manually, for a given beam of rays and a memory latency, and are expected to fit all the beams. Dynamically setting the cache parameters should allow a better cache efficiency for the different ray casting applications. Also, a dynamic setting of the parameters would allow a higher efficiency for different memory latencies.

8 Conclusion

In this chapter, we have presented a typical example of Algorithm Architecture Matching focused on the optimization of the memory hierarchy. It illustrates clearly that a co-design of the algorithm, the IP and the memory hierarchy leads to better results than a basic addition of standard IPs.

In a first step, the original grid traversal algorithm is transformed to exhibit more spatial locality than its theoretical expression. Indeed, a "virtual" loop is enabled

by the phase-locked propagation of blocs of rays. Then the memory references are coherent along this phase due to the initial geometrical coherence of the rays belonging to a beam. Splitting the computation of the result in blocs (or tiles) leads to coherent memory references. The phase-locked propagation is efficient both for the regular and recursive grid traversal.

In a second step, this transformation is shown to fit the 3D-AP Cache reference model. The 3D-AP Cache shows to be efficient at prefetching the volume data. The measures performed on an emulation board and by simulation have shown that the cache efficiency highly depends on the target application and the density of rays. For example, in the case of the visualization application, the efficiency drops when the volume is sub-sampled. This is not an issue as a volume of lower resolution could be used to render sub-sampled images.

The measures are performed in a worst-case situation. Indeed, on one side the background memory is supposed not to pipeline burst requests and, on the other side, the processing unit is able to perform a reference each clock cycle. An actual implementation may relax these constraints by allowing pipelining thanks to modern multi-bank memories. Also, the frequency of the references may be lower when the grid traversal would be implemented by software. Even better efficiencies would be reached in a less constrained implementation.

More generally speaking, the transformation applied to the original algorithm leads to a compromise between the efficiency of the memory hierarchy and the IPs area. Indeed, the phase-locked propagation needs some internal memories to store the intermediate results of the propagation. The phase-locked recursive grid traversal algorithm is the most memory hungry because it needs also to store the stacks of traversal of each rays. The proposed architecture is highly configurable to meet different target optimization criteria and fit integration constraints.

Further improvements and trade-offs are available to reach a better efficiency of the recursive grid traversal. Some new strategies to pre-fetch parts of the recursive grids are needed. The difficulty is to manage coherently the different levels of resolution.

We believe that the management of recursive data structure is of a great importance for many applications. For example, multi-resolution images are used in computer vision, video compression and 3D rendering. As the data traffic is one major of source of power consumption, optimizing the management of such data structures would enable complex image processing algorithm now restrained to desktop appliances to be implemented in embedded systems.

References

1. Akenine-Möller T, Haines E, Hoffman N (2008) Real-time rendering, 3rd edn. AK Peters, Natick
2. Amanatides J, Woo A (1987) A fast voxel traversal algorithm for ray tracing. In: Eurographics '87. Elsevier, North-Holland, Amsterdam, pp. 3–10
3. Ang S-S, Constantinides GA, Luk W, Cheung PYK (2008) Custom parallel caching schemes for hardware-accelerated image compression. J Real-Time Image Process 3(4):289–302

4. Felzenszwalb PF, Huttenlocher DP (2006) Efficient belief propagation for early vision. International Journal of Computer Vision 70(1)
5. Glassner AS (October 1984) Space subdivision for fast ray tracing. IEEE Comput Graph Appl 4(10):15–22
6. Grimm S, Bruckner S, Kanitsar A, Meister EG (October 2004) A refined data addressing and processing scheme to accelerate volume raycasting. Comput Graph 28(5):719–729
7. Havran V (November 2000) Heuristic ray shooting algorithms. Ph.D. thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague
8. Kanus U, Wetekam G, Hirche J (July 2003) VoxelCache: a cache-based memory architecture for volume graphics. In: Eurographics/SIGGRAPH workshop on graphics hardware, pp. 76–83
9. Klimaszewski KS, Sederberg TW (January–February 1997) Faster ray tracing using adaptive grids. IEEE Comput Graph Appl 17(1):42–51
10. Krüger J, Westermann R (2003) Acceleration techniques for GPU-based volume rendering. In: Proceedings IEEE visualization 2003
11. Köse C, Chalmers A (July 1997) Profiling for efficient parallel volume visualisation. Parallel Comput 23(7)
12. Larabi Z, Mathieu Y, Mancini S (June 2009) Efficient data access management for FPGA-based image processing socs. In: Proceedings of the 2009 IEEE/IFIP international symposium on rapid system prototyping, pp. 159–165
13. Lorensen WE, Cline HE (1987) Marching cubes: a high resolution 3d surface construction algorithm. SIGGRAPH Comput Graph 21(4):163–169
14. Mancini S, Desvignes M (2006) Ray casting on a SoPC platform: algorithm and memory tradeoff. In: IEEE conference on computer information technology, Seoul, Korea. IEEE, Los Alamitos
15. Mancini S, Eveno N (November 2004) An IIR based 2D adaptive and predictive cache for image processing. In: DCIS 2004, p. 85
16. nVidia. Cuda sdk. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html
17. Osborne R, Pfister H, Lauer H, McKenzie N, Gibson S, Hiatt W, Ohkami T (1997) EM-Cube: an architecture for low-cost real-time volume rendering. In: 1997 SIGGRAPH/eurographics workshop on graphics hardware. ACM, New York
18. Pfister H, Kaufman A, Chiueh T-c (1994) Cube-3: A real-time architecture for high-resolution volume visualization. In: Kaufman A, Krueger W (eds) 1994 symposium on volume visualization, pp. 75–82
19. Pfister H, Kaufman AE (1996) Cube-4 – a scalable architecture for real-time volume rendering. In: VVS, p. 47
20. Revelles J, Ureña C, Lastra M (2000) An efficient parametric algorithm for octree traversal
21. Strengert M et al. (2004) Large volume visualization of compressed time-dependent datasets on GPU clusters. Parallel Comput 31(2)
22. Wetekam G, Staneker D, Kanus U, Wand M (2005) A hardware architecture for multi-resolution volume rendering. In: HWWS '05: proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware. ACM, New York, pp. 45–51

http://www.springer.com/978-90-481-9964-8

Algorithm-Architecture Matching for Signal and Image Processing

Best papers from Design and Architectures for Signal and Image Processing 2007 & 2008 & 2009

Gogniat, G.; Milojevic, D.; Morawiec, A.; Erdogan, A. (Eds.)

2011, XII, 296 p., Hardcover
ISBN: 978-90-481-9964-8