GigaVoxels DP: Starvation-Less Render and Production for Large and Detailed Volumetric Worlds Walkthrough

Antoine Richermoz, Fabrice Neyret

GigaVoxels DP : Starvation-Less Render and Production for Large and Detailed Volumetric Worlds Walkthrough

ANTOINE RICHERMOZ and FABRICE NEYRET, INRIA / LJK (CNRS, UGA, INP-G), France

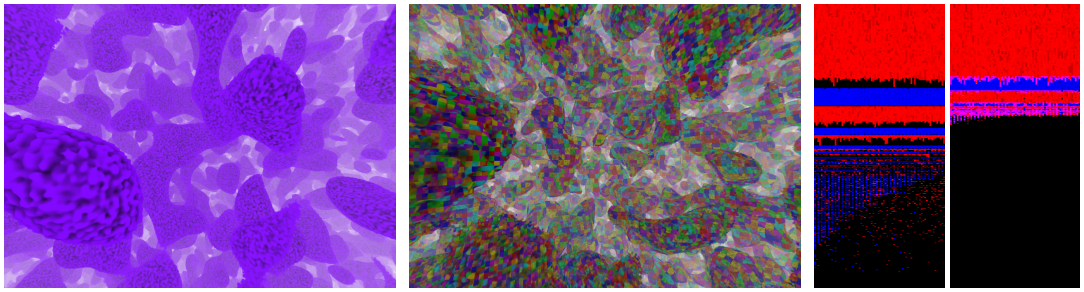


Fig. 1. Left: *a complex world rendered in* 12.8*ms with our approach compared to* 20.2*ms with GigaVoxels original approach (at* 1500×1000 *on an Nvidia 4080).* Middle: *colors show the bricks of* 8^3 *voxels. Thanks to MIPmap like LODs they all have near-identical screen-space size, so that each voxel corresponds to one pixel and the amount of required bricks for a given frame is near constant.* Right: *our GPU-cores timeline tool on GigaVoxels style tasks scheduling vs our scheduling (*horizontal: *core index,* vertical: *time,* red: *render task,* blue: *production task).*

Using voxel hierarchies as a generic 3D scene representation makes ray marching, antialiasing, and LOD easy. The drawback is the huge amount of memory required to store voxels, even with empty space compression. Still, GigaVoxels [Crassin et al. 2009] showed that by using a ray-guided cache to produce and store only visible voxels bricks on demand, it is possible to walk through very large and detailed worlds with real-time performance in bounded GPU memory. However, on-demand production of data during rendering is still challenging in terms of synchronization and starvation of GPU cores. We propose a new GPU-driven algorithm using dynamic parallelism (DP) to minimize these, and a "GPU-cores timeline" profiling tool to analyze them. We validate our model with timings (2× gain) and we illustrate it on various scenes.

CCS Concepts: • **Computing methodologies** → **Rendering**; *Volumetric models*; **Parallel computing methodologies**.

Additional Key Words and Phrases: Computer Graphics, Real-Time Rendering, Volume Ray-Marching, GPU-Driven, On-Demand, Dynamic Parallelism

Authors' address: Antoine Richermoz, Antoine.Richermoz@inria.fr; Fabrice Neyret, Fabrice.Neyret@inria.fr, INRIA / LJK (CNRS, UGA, INP-G), Grenoble, France.

1 INTRODUCTION

Our general goal is similar to GigaVoxels [Crassin et al. 2009]: we want to walk through very large and detailed worlds made of mostly opaque objects in real-time with high quality rendering (i.e., no popping artifact, near and far antialiasing). Voxel grids offer a total order which allows efficient ray marching, easy access to neighborhood information (e.g. near antialiasing is provided by hardware interpolation), and signal-processing makes LOD trivial as a simple MIPmapping operation. Volumes of voxels are easy to tile in bricks, which allows to compress empty space and eases the on-demand streaming or building of data and its caching.

The GigaVoxels method proposed a full scheme based on an octree of voxels bricks stored in an LRU cache, produced on-demand when a ray hits a missing brick, at the LOD resolution determined from the differential ray cones footprint just as for MIPmapping. Data production was thus triggered by rendering, but still, the algorithm was driven from the CPU: Each unfinished ray pass was followed by a stream-reduction pass providing a list of missing bricks, from which the CPU launched a brick content production pass, then relaunched unfinished rays from where they stopped. The iteration of these three passes yielded a lot of efficiency loss caused by these synchronizations and the frequent GPU-core starvation while treating small batches of variable-duration tasks - see Figure 1 (right).

Our main contribution is a new system where production and rendering tasks run in parallel, asynchronously, using CUDA Dynamic Parallelism [Adinets 2014].

To analyze and compare the impact of various possible choices on performances, GPU profiling tools are often tedious to exploit. We developed a "GPU-cores timeline" display tool showing compactly on the fly when each GPU core is working on a render task, a production task, or starving. We believe this tool can be very useful to understand what's going on on the GPU beyond the classical GPU profiling tools such as Nvidia Nsight [NVIDIA 2018].

This allows us to obtain a 2-fold gain compared to the GigaVoxels approach, and our profiling tool shows that GPU core occupancy is near-optimal. We analyze this and show results in section 4 and the companion video.

2 RELATED WORKS

2.1 Encoding large volumetric data

After seminal J.Carmack and J.Olick work on *Sparse Voxel Octrees* (SVO) [Carmack 2008; Olick 2008], several recent voxel-based games and papers took an interest in binary (i.e., either empty or opaque) voxels fields, either high resolution or coarse "boxels" (big voxels) like MineCraft. "Boxels" are often managed as meshes, while true binary voxels lead to near and far aliasing, and popping if LOD are used. As GigaVoxels [Crassin et al. 2009] we are rather interested in encoding "usual" CG scenes, using continuous density and voxel interpolation for antialiasing and seamless LOD transition - even if contrarily to scientific visualization our objects are generally meant to be globally opaque, which allows occlusion, space skipping and early exit. Still, we share the problem of finding efficient 3D structures to encode large scenes containing a lot of empty space, plus an LOD hierarchy: *octrees* have been proposed for real-time volume editing [Careil et al. 2020], DAGs can be used to compress SVO [Kämpe et al. 2013] (but this is mainly fitted for architectural flat & aligned content). GigaVoxels relies on an octree of bricks (cf *Brickmaps* [Christensen and Batali 2004]), containing density and auxiliary data.

Another useful representation for this task is *clipmaps* [Tanner et al. 1998], originally introduced for 2D terrain textures: for each LOD a relevant grid of data is stored around the camera, possibly focused in the view frustum [Losasso and Hoppe 2004]. As discussed by Panteleev [Panteleev 2014], this representation is simpler to implement, more memory coherent, and gives simple access to

neighborhoods, while octrees are more compact but require systematic traversal from the root to access bricks.

More generally, *virtual textures* [Lefebvre et al. 2004; Mittring and GmbH 2008] - for instance ID Software's *MegaTextures* - is a trend to emulate larger-than-memory data, via a page table and a data pool. Khronos *hardware sparse textures* [Obert et al. 2012] manage this totally transparently as a regular huge and possibly MIPmapped nD texture, but the maximum size is limited and we experienced that the updating performances were disappointing. Emulations like MegaTextures have to manage a strategy to handle the interpolation across tile borders. Rather than imposing complex and costly manual bi or tri-linear interpolation to shader programmers, the classical solution is to embed - and thus duplicate - the borders in each data brick. Still, the MIPmap interpolation has to be done manually. In practice, besides terrain texture [Cornel 2012], Virtual Textures seem to be mainly used for shadow maps [Lefohn et al. 2007].

In our implementation, we rely on clipmaps for the reasons mentioned above. But to recover some of the octree efficiency in empty space compression, we use clipmaps with indirection - i.e., as a coarse grid of block IDs plus a brick pool. Marking empty bricks also allows hierarchical empty space traversal. Still, virtual memory is not free lunch: As for GigaVoxels, for the real-time exploration of very large and detailed scenes to be possible we need all the necessary data for a given frame to fit the GPU memory. This is ensured by the "scene is mainly made of opaque objects in transparent space" hypothesis: thanks to the MIPmapping-like LODs, the screen will be covered with un-occluded bricks of near-constant screen-space size, as shown in Figure 1 (middle), and will thus occupy a near-constant space in the pool. The extra available memory and LRU management then reduce the need for production in case of camera meandering in the same neighborhood.

2.2 Managing on-demand data

Most methods assume that all the data is resident, which is not adapted for the walk-through of very large data on the GPU, even with empty space compression. Some consider out-of-core streaming and caching the data just in time when getting visible, but this is generally scheduled for the next frame, with temporary artifacts or lower resolution.

[Beyer et al. 2015] surveys several approaches there, while GigaVoxels [Crassin et al. 2009] triggers the production of missing bricks during the ray-marching of a given frame to always show seamless frames.

When the on-demand data is treated during the process using them, this leads to a dependency between two (or more) different tasks. GigaVoxels was alternating partial rendering tasks stopping on missing bricks and production tasks building them - with a stream compaction task in between to get a request list - then iterating on the unfinished rays. But this causes many synchronizations from the CPU, yielding starvation of GPU cores at the end of each task, and which can result in catastrophically low GPU occupation when iterating on the last chunks of unfinished rays - see Figure 1 (right); we analyze this in section 4.3. Tools like *Cuda Streams* [Harris 2015] allow the CPU to schedule a list of tasks with dependencies, but this is not dynamic, and only made to manage a handful of concurrent streams. Oppositely, *Dynamic Parallelism* [Adinets 2014] let GPU tasks totally manage each other. Performances and overhead of CUDA Dynamic Prallelism version 1 (CDP1) are analyzed in [Wang and Yalamanchili 2014]. The *persistent thread* strategy settles the dependent tasks as producers-consumers connected via FIFO arrays. For example the *Nanite* system [Brian Karis 2021] uses persistent threads to decide which LOD of meshlet should be shown.

In our implementation, we used Cuda Dynamic Parallelism version 2 ([Adinets 2014] Section 9.5), but it would also be possible to use OpenCL Device-Side Enqueue [Khronos 2013] or compute shader with the persistent thread model.

2.3 GPU profiling tools

Profiling parallel algorithms is not an easy task. Global statistics even on a single frame can hide very different unoptimal configurations, while seeing them eases their optimization. Existing GPU profiling tools such as Nvidia Nsight [NVIDIA 2018], AMD Radeon GPU Profiler [AMD 2017], or Microsoft PIX [Microsoft 2016] are widley used, but they show separate timelines for hardware counters and logical tasks, making it difficult to link the two. In addition, they work off-line rather than in real-time, and none of them support CUDA Dynamic Parallelism. So we developped the profiling tool we needed, showing in real time the synthetic timeline of which task is executing on which GPU cores.

3 OUR MODEL

3.1 Overview

Data structures. Like GigaVoxels and BrickMaps we rely on constant-size voxel bricks stored in a pool to compress the empty space and manage the dynamic caching of visible content, and we rely on differential ray cones to choose the LOD for which brick resolution matches pixel resolution (as for MIPmapping). To benefit from the native trilinear interpolation, our bricks also store borders. But we opted for clipmaps of bricks rather than octrees of bricks as an efficient dynamic data structure to store the cached bricks and the LOD hierarchy - which is also required for efficient space-skipping. We justify and detail our structure in section 3.2.

Task scheduling system. Like GigaVoxels we have a rendering task and a production task. But to avoid the loss of efficiency and GPU-core starvation due to synchronizations between iterated alternate invocations - especially when they are small batches -, we want to be able to run them concurrently. For this we rely on the Cuda Dynamic Parallelism [Adinets 2014] : each rendering thread reaching a missing brick launch its production task, and each finished production task relaunch rendering threads from where ray were interrupted. A difficulty is that the parallel domains - neighbor pixels on screen vs neighbor voxels in bricks - are not the same for the two tasks, so that a rendering warp can touch several bricks while a brick can be requested by several rays. We detail the two tasks and our scheduling system in section 3.3.

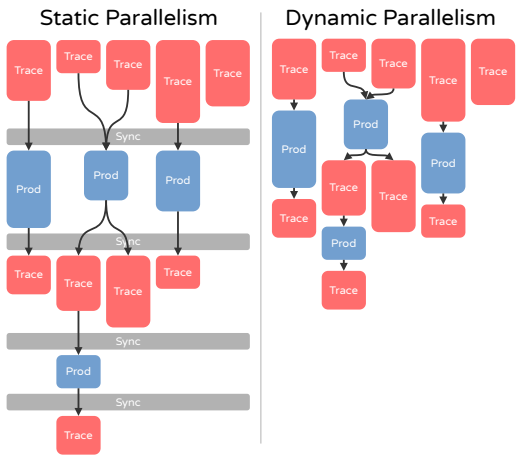


Fig. 2. Comparison of task scheduling with static parallelism (i.e., Gigavoxels-like) vs dynamic parallelism when applied to our problem.

Starvation analysis. We develop a very convenient "GPU-cores timeline", showing in real time the synthetic timeline of all the physical GPU cores, see Figure 1, right. Note that it closely mimics the information of Figure 2). It was instrumental in the creation of our model and the development of our prototype. We study how the different scheduling choices behave in terms of GPU-cores occupancy in section 4.3, using tool we developed.

3.2 Data Structures

In our method, we employ the following data structures:

- *Brick Pool*: A large 3D texture atlas that is used as a brick cache. Each texel has an opacity plus any other required material parameter (color, normal, ...). Bricks are 8^3 voxels in our implementation.
- *Indirection Table*: It maps each brick position (XYZ + LOD) to either a slot in the brick pool or a special value indicating that this bricks is *empty*, *missing*, or *in-production*. It is implemented as a clipmap. As for a 2D clipmap, $clipmapWidth = \left\| \begin{matrix} \frac{screenWidth}{screenHeight} \\ 1 \\ \frac{1}{\tan\frac{fov}{2}} \end{matrix} \right\| \cdot \frac{screenHeight}{brickWidth}$ (300^3 in our examples), for each LOD (8 were sufficient in our examples).
- *Inverse Indirection Table*: It maps each brick pool slots to the position (XYZ + LOD) of the corresponding brick.
- *Timestamp Buffer*: A buffer the size of the brick pool that stores the last timestamp at which the corresponding brick in the page pool was accessed.
- *Free Slot List*: A fixed size list (stored as a continuous buffer) of available brick pool slots not mapped to anything (either never used, freed by the LRU, or by falling out of the clipmap).
- *Ray Payload Buffer*: A screen size buffer that stores the state of rays. Each entry contains the distance traveled, the RGBA color accumulated, and a boolean lock to avoid concurrent execution of the same pixel.

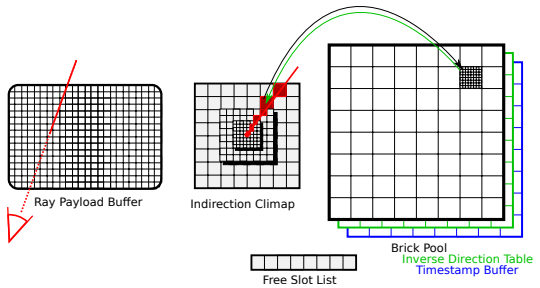


Fig. 3. Our data structure. For a ray in red, the clipmap entries at various LOD, pointing to the allocated voxel bricks.

3.3 Task scheduling system

3.3.1 Rendering task. At first, we start by launching a ray-tracing thread for every pixel (by tasks of 32×16 pixels in our implementation, and variable size at later relaunch). When a ray tracing thread encounters a missing brick (at the required position+LOD), we save its state in the *Ray Payload Buffer* and we release its lock. Then we try to launch a production task for the missing brick. To do so, we use an atomic *compare-and-swap operation (atomicCAS)* to try to change the brick's state from *missing* to *in-production*. If it succeeds, we can safely launch the production task using dynamic parallelism. Then the rendering thread can stop. This is detailed in Algorithm 1.

3.3.2 Production task. A Brick production task starts by computing the data of each voxel in parallel, with one thread per voxel. Once this is done, we have to update our data structures, which is done by the single thread #0 of the production task.

If the brick is empty, we just have to mark it as such in the indirection table. Otherwise, we need to find a slot in the brick pool to store it and update the indirection table. We obtain a slot using the *Free Slot List* (see Section 3.2) and an atomic index. Voxels can now be stored in the brick pool

Algorithm 1 Rendering task

```
Input: rayIndex                                                    ▷ threadIndex
  ray ← getRay(rayIndex)                                           ▷ ray origin and direction
  payload ← payloadBuffer[rayIndex]                                ▷ color, t, lock
  if atomicCAS(payload.lock, false, true) then
      return
  end if
  while payload.color.a < 1 do
      p ← ray.origin + ray.direction * payload.t
      brickIndex ← getBrickIndex(p)
      brickSlot ← atomicCAS(indirectionTable[brickIndex], missing, inProduction)
      if brickSlot = missing then
          payload.lock ← false
          payloadBuffer[rayIndex] ← payload
          launchBrickProduction(brickIndex)                        ▷ Dynamic Parallelism Call
          return
      else if brickSlot = inProduction then
          payload.lock ← false
          payloadBuffer[rayIndex] ← payload
          return
      else if brickSlot = empty then
          skipBrick(brickIndex, ray, payload)                      ▷ Updates t
      else
          d ← sampleBrick(brickSlot, p)
          c ← computeShading(d)                                    ▷ Optional uncompress or shading
          payload.color ← payload.color + (1 − payload.color.a) * c
          payload.t ← payload.t + getVoxelSize(p)
      end if
  end while
  frameBuffer[rayIndex] ← payload.color
```

and the indirection table can be updated with the given slot. The indirection table now has a valid address so the brick is not *in-production* anymore (since the flag is stored in the same field).

Rays that were waiting for this brick can now be restarted. We use Dynamic parallelism once again to launch a rendering task re-starting a ray-tracing thread for every pixel in the brick's screen-space bounding box. (In an earlier implementation we recorded a precise list of rays to relaunch, but the extra structures and locks proved less efficient).

As this will probably contain rays that were not waiting for this brick, ray tracing threads start by testing their legitimacy by trying to acquire their respective ray lock using an *atomicCAS*. If they succeed, they can reload their saved state and restart ray tracing accordingly. If not, they are stopped immediately (since the ray is already alive elsewhere or finished). This is detailed in Algorithm 2.

3.3.3 Possible data race. There is still a possibility for a data race here. In the very short time between a ray-tracing thread seeing that a brick is *in-production* and that ray releasing its lock, it is possible for brick production to end and attempts to restart the same ray happening. If that is the case, the thread in charge of restarting that ray will not be able to acquire the lock. This ray will thus never terminate, resulting in an empty pixel. In practice this was an exceedingly rare event in our tests (we measured about 1 data race every 1000 frames). But if 100% sanity is required, it can be prevented by relaunching an extra full screen render pass until no pixel is empty.

Algorithm 2 Production task

```
Input: voxelIndex                                              ▷ threadIndex
Input: brickIndex                                              ▷ destination
  p ← getPosition(brickIndex, voxelIndex)
  d ← computeVolumeData(p)                                     ▷ Custom voxel production
  if threadBlockAnd(isEmpty(d)) then
      if threadIndex = 0 then
          indirectionTable[brickIndex] ← empty
      end if
  else
      if threadIndex = 0 then
          brickSlot ← freeSlotList[atomicAdd(slotListIndex, 1)]
      end if
      brickPool[brickSlot, voxelIndex] ← c
      if threadIndex = 0 then
          indirectionTable[brickIndex] ← brickSlot
      end if
  end if
  if threadIndex = 0 then
      brickBbox ← getBrickBbox(brickIndex)
      launchRayTracing(brickBbox)                              ▷ Dynamic Parallelism Call
  end if
```

3.3.4 Free Slot List. The *Free Slot List* is maintained as follows: Each time a brick is accessed, its corresponding slot in the *Timestamp buffer* gets updated with the timestamp of the current frame. At the beginning of every frame, brick pool slots are sorted by their timestamp value from the timestamp buffer. This is done very fast thanks to GPU radix sort. The *Free Slot List* then gets filled with the first entries from that sort, i.e. the brick slots that were the least recently used (LRU), or never used, or freed by the clipmap after a camera motion. The bricks from the list then get freed by setting their state to *missing* in the indirection table. For this we need to use the inverse indirection table to go from slots in the brick pool to brick positions in the indirection table.

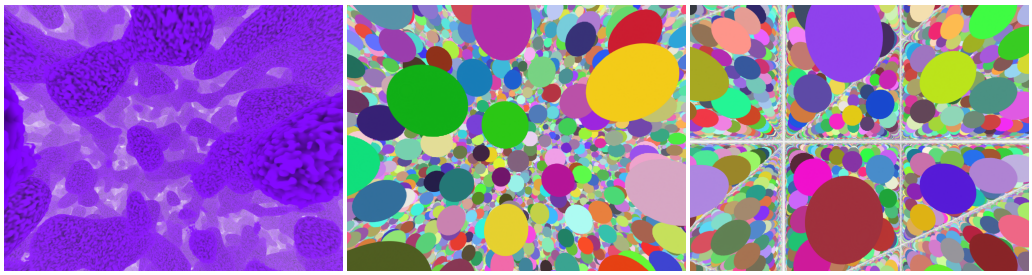
In the degenerate case where the index exceeds the list's capacity, all production is stopped, the list gets regenerated, and a full screen ray tracing task is relaunched, similar to GigaVoxels scheduling. However, this is only a back-up solution and we found that using a list capacity of 10% of brick pool capacity results in no list regeneration during normal traversal (in our examples, moving back through a wall covering the full screen would require $\tilde{2}5\%$ of that).

4 RESULTS AND ANALYSIS

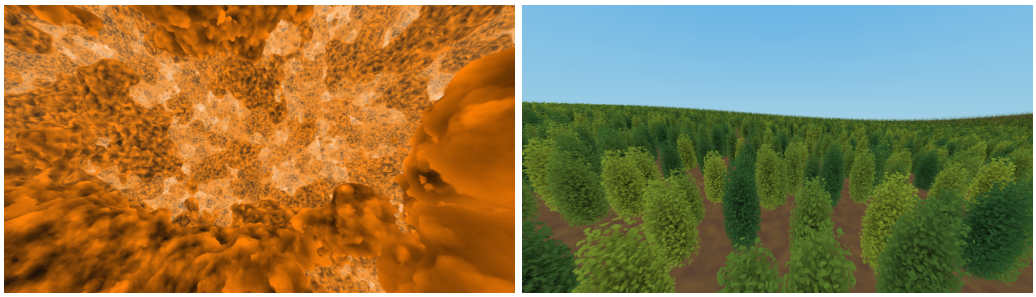
4.1 Testing methodology

The goal of our performance analysis is to measure the speed improvement of our new task scheduling system, as well as to demonstrate the viability of high resolution voxel caching for real-time rendering of generic scenes. In order to compare ourselves to the GigaVoxels approach, we created a modified version of our prototype to run with static (as opposed to dynamic) parallelism task scheduling.

In our prototype bricks are 8^3, and voxels only store opacity + color with pre-backed shading. Our test scenes use procedural brick producers, productions tasks consist of evaluating a (costly) procedural function. We designed five test scenes with differing requirements (see Figure 4):



(a) Multi-layer Perlin noise (b) Random spheres (c) Aligned spheres



(d) 10 Layer FBM (e) Forest

Fig. 4. Test scenes used for performance analysis

- A two-layer Perlin noise scene to simulate a somewhat credible environment with lots of disocclusion and a costly to evaluate procedural function.
- A up-to-10 layers Perlin noise scene showing both a very detailed first stage and large world. The fractal depth is adapted to the LOD.
- A forest scene where tree foliage is passing-through so that only aggregated occlusion from several trees can stop rays. We used a very naive costly procedural producer: basically a field of leaves is potentially computed everywhere in space (16 leave SDF evaluations per voxels, including many sin-based random numbers) and truncated by the closest tree ellipsoid SDF.
- An extreme uncorrelated disocclusion scene consisting of randomly positioned spheres.
- An extreme correlated disocclusion scene consisting of sphere aligned on a grid (causing free sight up to the horizon and many simultaneous production requests caused by the synchronised disocclusions).

Note that in all these examples we used no oracle or bounding box to pre-register empty space or lighten the procedural evaluation. The cache and hierarchical production alone naturally concentrate the calculation where and when useful. Conversely, we didn't implemented view-dependant shading so the stress in higher on the production tasks. For the sake of demonstration and benching we rely on 100% voxels rendering, but note that in a real application it would be easy to mix ordinary rasterized scenes with GigaVoxels via the Z-buffer, so that hero first stage objects could be rendered with regular polygons if preferred - and animated object can be added to the scene the same way.

In order to stress the scheduling systems in different ways, we tested five movement types: simulated walkthrough, moving forward, moving backward, moving sideways, and turning on the spot.

All testing was done on a Linux system equipped with an NVIDIA 4080 GPU and an AMD 7900X CPU, at a resolution of 1500×1000. GPU memory utilization was as follows: 2GB for the brick pool, 1GB for the indirection table, and 1GB for the data structures allocated by CUDA for Dynamic Parallelism, the rest being negligible, resulting in a total memory utilization of a little bit more than 4GB.

4.2 FPS Results

Table 1. Measured average FPS for various scenes and movement type, comparing Static Parallelism (SP) "GigaVoxels" and our Dynamic Parallelism (DP) scheduling, and showing the speedup (SU) from SP to DP.

Scene	Perlin noise			FBM			Forest			Random spheres			Aligned spheres			Average
Scheduling	SP	DP	SU	SP	DP	SU	SP	DP	SU	SP	DP	SU	SP	DP	SU	SU
Walkthrough	49.5	78.1	**1.6**	20.0	29.4	**1.5**	41.0	48.3	**1.2**	22.2	46.6	**2.1**	14.1	35.9	**2.5**	**1.9**
Forward	57.0	85.9	**1.5**	19.5	29.0	**1.5**	63.5	69.1	**1.1**	23.8	51.4	**2.2**	19.6	35.1	**1.8**	**1.7**
Backward	41.4	84.3	**2.0**	10.8	26.1	**2.4**	59.5	65.8	**1.1**	15.6	50.0	**3.2**	8.3	36.3	**4.4**	**2.9**
Sideways	65.7	104.4	**1.6**	42.6	65.8	**1.5**	80.1	96.7	**1.2**	22.5	54.2	**2.4**	14.4	47.5	**3.3**	**2.2**
Spinning	64.8	95.0	**1.5**	33.5	48.5	**1.4**	50.9	75.7	**1.5**	37.2	69.3	**1.9**	24.2	41.1	**1.7**	**1.7**
Average	55.7	89.5	**1.6**	25.3	39.8	**1.7**	59.0	71.1	**1.2**	24.3	54.3	**2.3**	16.1	39.2	**2.7**	**2.1**

All our FPS results can be seen in Table 1. We get a speedup ranging from **1.1** to **4.4**, with an average of **2.1**. We also see that scenes with more disocclusion, especially when it is correlated, perform worse for all systems, but show a better speedup. Similarly, movement types that produce more disocclusion, such a moving backward or sideways, also show a better speedup.

We also measured the 1% low FPS (average FPS of the slowest 1% of frames, indicating performance consistency), as we initially thought that our scheduling system would show a better speedup because of its ability to eliminate cases of extreme underutilization. But in the end the overall speedup was exactly the same at **2.1**, with no clear pattern of which test showed a better improvement for 1% lows or average FPS. In absolute term, 1% FPS values were about 60% of average FPS values.

4.3 GPU-cores Timeline Tool

To better understand how task scheduling affects GPU utilization all along the frame evaluation, we developed a custom profiling tool showing a timeline of the activity of each GPU core (SM) during a frame, with time on the vertical axis (starting from the top), core index on the horizontal axis, and a different color for each task that the core is currently executing. We used red for rendering tasks and blue for production tasks, black indicating that the core is idle, and purple that it is running both tracing and production at the same time (see Figure 5).

Our custom timeline tools work by recording a timeline entry in a buffer for each ThreadBlock executing a kernel. A timeline entry consists of: a kernel color, a GPU-core index, a start time and an end time. In order to be

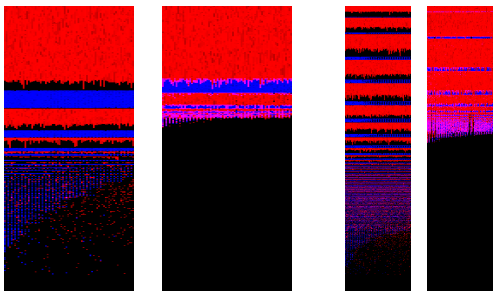


Fig. 5. Example of GPU-cores timelines for a short frame (5a) and a long frame (5b), comparing static parallelism scheduling (5a1, 5b1) and dynamic parallelism (5a2, 5b1). *horizontal:* core index, *vertical:* time, *red:* render task, *blue:* production task.

recorded to the timeline, kernels must be instrumented with a special function call at their beginning and end. The "begin" function call does a ThreadBlock-wise atomicMin with the current time to get the start time of the kernel, allocates a timeline entry from the timeline entry buffer using an atomic index, and records to this entry the kernel color, GPU-core index, and start time. The "end" function call does a ThreadBlock-wise atomicMax with the current time to get the end time of the kernel, and records it to the timeline entry. At the end of the frame, timeline entries are processed and display to the screen. All of this introduces a negligible but measurable overhead (about 2%), so we deactivated it for performance analysis.

In Figure 5, we can see that fixed scheduling leads to a lot of idle time. Idle GPU cores are present at the end of every ray tracing pass, as some rays take longer to trace than others. Similarly, idle time would also be present at the end of each production pass if the brick's producers were heterogeneous. There is also idle time in between each pass, caused by CPU-GPU synchronizations. Finally, there is a lot of idle time at the end of a frame - the "tail" regime, which can sometimes represent more than half of the total time -, when tasks are unable to saturate the GPU. We can see that dynamic parallelism is able to eliminate almost all idle time, at least as far as our timeline tool can measure it. Most of the gain come from compacting the tail of the frame, but significant improvement is also extracted at the beginning just by better overlapping tasks.

5 LIMITATIONS, FUTURE WORK AND CONCLUSION

By using dynamic parallelism, we were able to dramatically increase the effective GPU utilization for GigaVoxels-like applications, resulting in a more than 2× gain on average. This enabled traversal of various scenes with real-time frame rates (30 to 100 fps), at a resolution of 1500×1000, and with perfect filtering at one voxel per pixel. For now, the drawback of our solution is the high memory cost of the data structures associated with dynamic parallelism. However, we believe that by using a custom implementation of dynamic parallelism instead of the one built into CUDA, that memory cost could be reduced. Moreover, using a custom dynamic parallelism implementation could result in better performance, by tailoring it to our specific needs. In addition, a Vulkan-compatible version would open a way broader applicative world such as gaming. Also, relaunching all the rays covering a whole brick bounding box yields several incomplete warps: a better packing should improve rendering task performance. Production-wise, it would be interesting to study some strategies to smooth out the peaks of production, such as relying on a prefetching oracle (e.g. when a voxel fetch was close to the max current LOD) or deferring a brick LOD change to the next frame so as to prioritize the production of totally new content.

We wanted to experiment clipmaps as an alternative to octrees (or more generally, n^3-trees). While it does simplify the implementation, it consumes more memory (where a subtree would be clamped) and is not proven faster - which may depend on scenes. So it is mostly a programmer's choice.

The GPU-cores timeline profiling tool we developed to aid in our comprehension of GPU scheduling proved extremely convenient and fruitful. By working in real-time and showing physical cores instead of logical streams, it provides crystal-clear information that existing solutions don't show explicitly, while supporting dynamic parallelism. This tool could be further improved and made into an API so as to be integrated into other projects.

ACKNOWLEDGMENTS

We thank Cyril Crassin for his answers to our numerous questions. We also thank Thibault Tricard for proofreading and comments on the paper.

REFERENCES

Andy Adinets. 2014. Adaptive Parallel Computation with CUDA Dynamic Parallelism. https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/

AMD. 2017. Radeon GPU Profiler. https://gpuopen.com/rgp

Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. 2015. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum* 34, 8 (2015), 13–37. https://doi.org/10.1111/cgf.12605

Graham Wihlidal Brian Karis, Rune Stubbe. 2021. Nanite - A Deep Dive - SIGGRAPH 2021. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

V. Careil, M. Billeter, and E. Eisemann. 2020. Interactively Modifying Compressed Sparse Voxel Representations. *Computer Graphics Forum* 39, 2 (2020), 111–119. https://doi.org/10.1111/cgf.13916

John Carmack. 2008. talk at id Tech 6. https://pcper.com/2008/03/john-carmack-on-id-tech-6-ray-tracing-consoles-physics-and-more/

Per H. Christensen and Dana Batali. 2004. An Irradiance Atlas for Global Illumination in Complex Production Scenes. In *Eurographics Workshop on Rendering*. https://doi.org/10.2312/EGWR/EGSR04/133-141

Daniel Cornel. 2012. Texture Virtualization for Terrain Rendering. https://www.semanticscholar.org/paper/Texture-Virtualization-for-Terrain-Rendering-Cornel/b8824c1361455e199e61d5e907efc3a00404c39e

Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. 2009. GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009-02-27) *(I3D '09)*. 15–22. https://doi.org/10.1145/1507149.1507152

Mark Harris. 2015. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

Khronos. 2013. OpenCL Device-Side Enqueue documentation. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#device-side-enqueue

Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. 2013. High resolution sparse voxel DAGs. *ACM Transactions on Graphics* 32, 4 (2013), 1–13. https://doi.org/10.1145/2461912.2462024

Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. 2004. *Unified Texture Management for Arbitrary Meshes*. Research Report RR-5210. INRIA. 20 pages. https://inria.hal.science/inria-00070783

Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. 2007. Resolution-matched shadow maps. *ACM Transactions on Graphics* 26, 4 (2007), 20. https://doi.org/10.1145/1289603.1289611

Frank Losasso and Hugues Hoppe. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.* 23, 3 (aug 2004), 769–776. https://doi.org/10.1145/1015706.1015799

Microsoft. 2016. PIX. https://devblogs.microsoft.com/pix/introduction

Martin Mittring and Crytek GmbH. 2008. Advanced virtual texture topics. In *ACM SIGGRAPH 2008 Games* (2008-08-11). 23–51. https://doi.org/10.1145/1404435.1404438

NVIDIA. 2018. NVIDIA Nsight. https://developer.nvidia.com/nsight-compute

Juraj Obert, J. M. P. Van Waveren, and Graham Sellers. 2012. Virtual texturing in software and hardware. In *ACM SIGGRAPH 2012 Courses* (2012-08-05). 1–29. https://doi.org/10.1145/2343483.2343488

Jon Olick. 2008. SIGGRAPH 2008: Beyond Programmable Shading Class. In *ACM SIGGRAPH 2008 Courses*. ACM. https://www.jonolick.com/uploads/7/9/2/1/7921194/olick-current-and-next-generation-parallelism-in-games.pdf

Alexey Panteleev. 2014. Practical Real-Time Voxel-Based Global Illumination for Current GPUs. https://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf

Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. 1998. The clipmap: a virtual mipmap. In *Proceedings of SIGGRAPH '98* (1998). 151–158. https://doi.org/10.1145/280814.280855

Jin Wang and Sudhakar Yalamanchili. 2014. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014-10). 51–60. https://doi.org/10.1109/IISWC.2014.6983039