360.245 - Selected Topics for Computational Electronics

Mario R. Hiti

September 4, 2022

Abstract

OpenVDB and NanoVDB are data structures which were initially developed for image and movie rendering. However it is possible to use the provided frameworks within the context of semiconductor process simulation. This paper aims to benchmark the raytracing performance of NanoVDB and OpenVDB for said application in a worst-case scenario on a small scientific computing cluster. NanoVDB, which covers a subset of OpenVDB's functionality, is also compatible with common graphics APIs. Therefore its behavior on GPUs and it's accuracy is examined as well. The benchmarks show that NanoVDB performs about 6x faster on a GPU compared to OpenVDB on similarly priced hardware and with similar power consumption under full load. However the accuracy of NanoVDB is limited by the grid's voxel size.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Limitations of NanoVDB . . . 2

2 Methodology **2**
- 2.1 Hardware setup . . . 2
- 2.2 Simulation environment . . . 2
- 2.3 Intersection accuracy . . . 4

3 Results **5**

4 Discussion and Outlook **6**
- 4.1 Performance . . . 6
- 4.2 Potential performance improvements . . . 6

5 Conclusions **7**

6 Appendix **8**

1 Introduction

1.1 Background

The production of semiconductors has been one of the biggest growing markets during the last decades which resulted in significant advances in production capabilities and a wide range of possible production steps. These processes often take place at the scale of nanometers which make both process design, and the verification of results exceedingly difficult. The surface structure of semiconductors is often too complex for analytical models and quality control often requires expensive equipment such as electron microscopes.

Due to the increase of computing power it has now become common practice to simulate many steps of the production process. Many simulations involve rays and ray-casting of some sort which is a very common problem encountered in the gaming and movie industry. These types of computations are in fact so important that many computers include dedicated graphical processing units (GPUs) to accelerate such calculations. However modern hardware is still far from being able to simulate every single atom in a focused ion beam or every single photon from a light source. Therefore simulations usually resemble an approximation using a limited amount of virtual rays or atoms. However in general simulations usually benefit from an increase in computed elements (i.e. the more the better).

An important part of every simulation is the choice of underlying data structure. One option is OpenVDB which can be used to efficiently store high resolution volumes and level-sets [5]. OpenVDB was initially developed for the animation and movie industry. Due to its flexibility it is also possible to adapt it for use in semiconductor process simulation [2].

In a recent article NVIDIA published a benchmark that promises a significant speed-up when using NanoVDB. For process simulations the results for level-set raytracing are of significant importance. According to Tab. 1, NanoVDB on a GPU should be 60x faster compared to a multithreaded implementation using OpenVDB. (Execution time of 2.427ms vs 148.182ms).

Table 1: Benchmark results published by NVIDIA. The benchmark setup and source code are undisclosed [1].

	OpenVDB (TBB)	NanoVDB (TBB)	NanoVDB (CUDA)	CUDA Speed-Up
Level Set	148.182	11.554	2.427	5x
Fog Volume	243.985	223.195	4.971	44x
Collision	-	120.324	10.131	12x

However since the benchmark setup and source code are no published it is not clear if the same increase in performance can be achieved for other applications. Therefore the goal of this paper is to verify these results using a benchmark that is tailored to typical applications within the semiconductor process simulation.

1.2 Limitations of NanoVDB

NanoVDB is still in its early development phase and therefore it is likely that not all features are implemented yet. Therefore NanoVDB currently comes with several limitations when used for simulations:

static grids: Modifications to the tree topology which are required for problems such as surface advection are not supported. However it is possible to perform an intermediate transformations to OpenVDB in order to make changes to the tree.

accuracy: Only the voxel containing the intersection is returned by NanoVDB's raytracing function (`nanovdb::ZeroCrossing`, see Alg. 6). In contrast to OpenVDB, no further steps are performed to locate the intersection point within a voxel.

2 Methodology

The benchmark is designed to be baseline for future applications that are using NanoVDB for narrow-band level-sets and raytracing within the context of semiconductor process simulation. Therefore no specific problem is chosen but the worst-case scenario in a typical application is modelled.

Source code, build instructions and measurement data are available at:

`https://github.com/hitimr/SelectedTopicsCompElectronics`

2.1 Hardware setup

The benchmark is performed on a single node of a scientific cluster operated at the Institute for Microelectronics at the TU Wien. The node consists of the devices listed in Tab. 2 which are both used for the benchmark.

CPUs and GPUs are different platforms in terms of architecture and design which makes a fair comparison with regards to their technical aspects difficult. However both devices are similar in cost of acquisition and operating expenses (i.e. power usage). Furthermore both platforms are marketed towards scientific computing.

Table 2: Hardware used for the benchmark. Prices may fluctuate due to current events. Power consumption represents the absolute maximum ratings according to the vendor

	Price	Power Consumption	Cores
Intel Xeon 6248	€ 3.300	105W	20 Cores; 40 Threads
NVIDIA Tesla T4	€ 2.500 - 3.000	70W	2.560 CUDA-Cores

2.2 Simulation environment

A common problem in Semiconductor process simulation is light being cast into a trench with semi-reflective walls as shown in Fig. 1 (left). To simplify the program and enforce a worst-case scenario the following modifications are performed:

- All calculations are performed using 32-bit floating point numbers (`float`) as single-precision arithmetics have been shown to be sufficiently accurate [2, Chapter 5.1]
- Rays leaving the bounding box (i.e. shooting into the sky) are cheaper to compute but do not contribute to the simulation. In order to prevent these edge cases rays are cast onto the inner surface of a hollow sphere.

- The point source is replaced with a volumetric source. Otherwise every ray would start within the same voxel which would lead to a beneficial memory access pattern.
- Depending on the reflecting angle, rays may cover different distances. Therefore the inner sphere (ray source) is offset to create a distribution of distances.
- Rays are shuffled in memory before being passed to the kernel to prevent beneficial memory access patterns.
- Any ray reflection on the surface is equivalent to having 2 separate rays (inbound and outbound) at the intersection point. Therefore reflections do not need to be modelled.

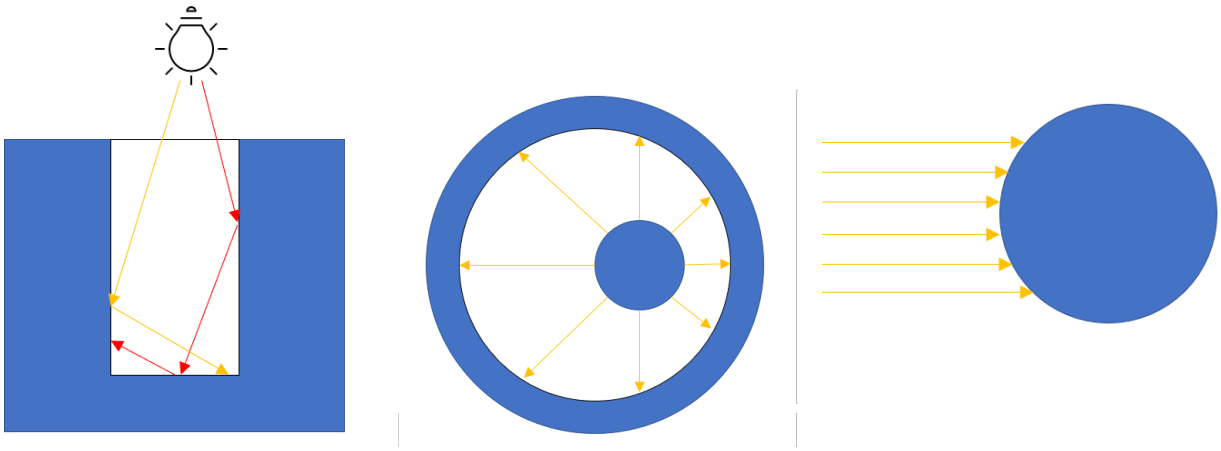

Fig. 1: Benchmark setup. **left** Trench being illuminated by a light source. Every ray has a chance of being reflected or absorbed. **center** 2D cross-section of the modified setup. **right** Setup for determining accuracy of the calculation

Origin and direction of every ray along with a ground truth are precomputed and passed to three different ray intersection kernels:

- OpenVDB (CPU)
- NanoVDB (CPU)
- NanoVDB (GPU)

The OpenVDB kernel serves as a baseline for comparison. Both NanoVDB kernels are identical but launched on different platforms. CPU based benchmarks are performed using all 40 threads.

Only the time required to calculate intersections is measured. Memory management, data transfer, ray generation, result verification, etc. is not included. After the benchmark is complete the number of calculated rays per second is derived using

$$Rps = \frac{ray\ count}{time} = [\frac{1}{s}] \tag{1}$$

The benchmark is repeated as the number of rays is increased until the measured performance starts to plateau. If the benchmark requires more rays than RAM or VRAM can store, then pregenerated rays are recycled. This has the same effect as increasing the memory of a device as the set of rays is big enough (100s of millions) such that no beneficial memory access patterns emerge. The asymptotic behaviour of the resulting performance curve is used to estimate a potential performance gain for switching to NanoVDB or GPUs.

2.3 Intersection accuracy

After each iteration the resulting intersections ($\mathbf{r}_{calc}$) are compared to a pre-computed ground truth ($\mathbf{r}_{gt}$) to assure the correctness of the results. As mentioned in chapter 1.2, NanoVDB does not compute intersections within a voxel. Therefore results are considered correct if the following condition is satisfied for each ray [1]:

$$\|\mathbf{r}_{calc} - \mathbf{r}_{gt}\|_2 \leq \sqrt{3} \cdot VOXEL\_SIZE \tag{2}$$

The difference between OpenVDB and NanoVDB in terms of accuracy can be illustrated by projecting parallel rays onto a sphere as shown in Fig. 1 (right). While OpenVDB is able to accurately find intersections within a given voxel, NanoVDB only returns the position of the voxel center containing the intersection. Therefore the only way to increase accuracy without performing further calculations is to increase the grid resolution i.e. decrease voxel size.

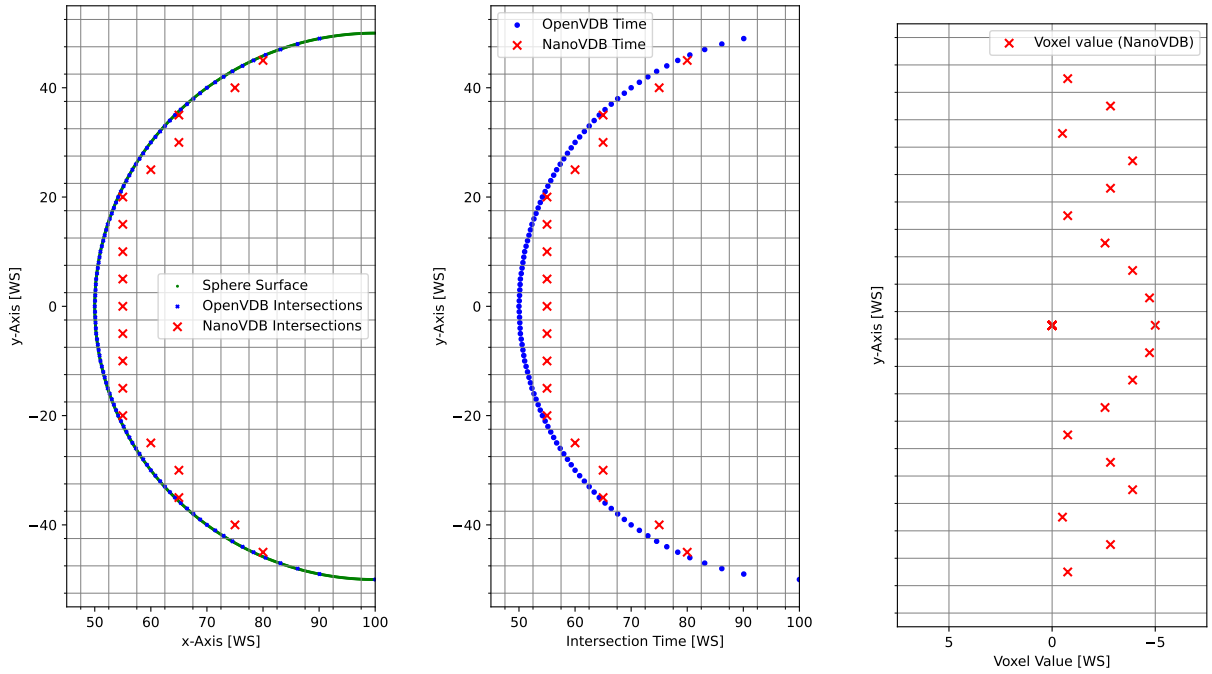

Fig. 2: Difference in terms of accuracy of OpenVDB and NanoVDB. The grid lines resemble voxels. In total 100 rays are used but many results overlap because the accuracy is reduced to voxel-level **left:** Calculated intersection points compared to analytical solution **center:** Intersection time/distance **right:** Voxel value at intersection point. (NanoVDB only)

[1] While the intersection can be located anywhere between a bounding cube, a sphere is used as it is simpler to compute

3 Results

Fig. 3 shows the achieved performance for different problem sizes on all three kernels. The spheres as shown in Fig. 1 are represented using approx. 6.3mio active voxels. NanoVDB achieves overall better results on the CPU compared to OpenVDB. Above one million rays the GPU starts to overtake both CPU kernels.

OpenVDB achieves up to 18.5 MRps [2]. NanoVDB consistently outperforms OpenVDB and reaches up to 29.7 MRps. For problems with 1 million rays or more the GPU kernel overtakes both CPU implementations and achieves up to 117.4 MRps. Therefore the switch from OpenVDB to NanoVDB increased performance by a factor of 6.3.

Furthermore both CPU implementations suffer from random drops in performance while GPU results are more consistent.

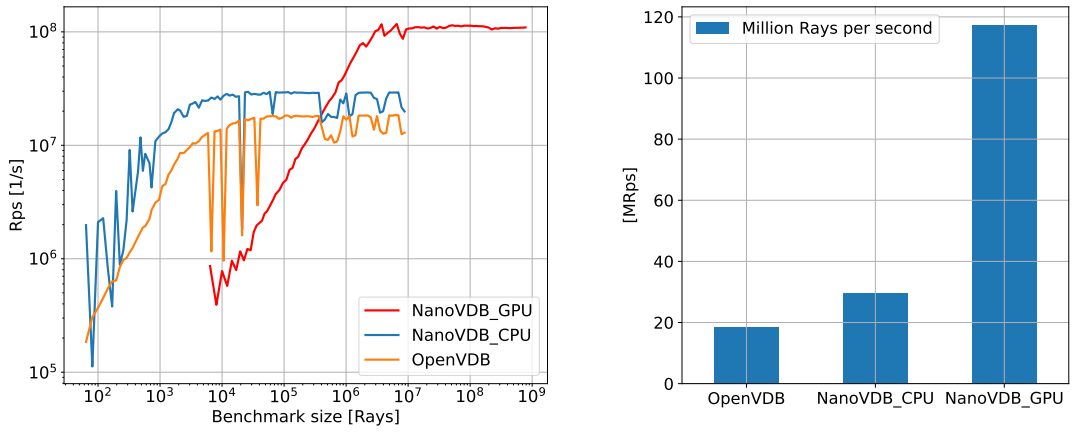

Fig. 3: **left:** measured performance across different problem sizes. **right:** Results for more than 5M rays

[2] 1 MRps = 10^6 Rays per second

4 Discussion and Outlook

4.1 Performance

As shown in Fig. 3 NanoVDB does provide better performance compared to OpenVDB on both platforms. However this is most likely due to the fact that OpenVDB performs additional steps to increase it's accuracy as mentioned in section 1.2.

Only for small problem sizes the CPU delivers better results. Once the benchmark surpasses $10^5 - 10^6$ rays the GPU overtakes the CPU. This behaviour is very common for GPU benchmarks because CPUs usually have significantly higher single core performance and lower latencies. Below a certain threshold the GPU is often not even able to utilize it's full performance as more threads than rays are available. However within the context of semiconductor process simulation the only the high-end spectrum is relevant.

While NVIDIA promises an increase of performance of approx. 60x (see Tab. 1), this benchmark is one order of magnitude slower. This is likely because NVIDIA's benchmark represent a CG[3] scenario while this benchmark aims to be a worst-case scenario for scientific applications.

Since both frameworks use different data structures and algorithms, significant adaptations to the code base are required for a project to migrate to NanoVDB. The expected increase in performance for CPU based systems may be considered too small to justify the amount of necessary work. However for machines with a dedicated GPU a switch to NanoVDB can be very attractive especially for large simulations using multiple GPUs. NanoVDB-Kernels can be used interchangeably on CPU and GPU, i.e. the target platform is determined at compile time. Therefore very little adaptations to the codebase are required to launch kernels on either CPUs or GPUs. This also allows to easily combine both platforms to perform calculations simultaneously thus effectively combining the performance of both platforms.

In its current form NanoVDB only supports static grids. This means that the whole data structure must me re-computed if the level-set changes. Depending on the complexity of the transformation it might be easier to convert the grid to the OpenVDB format before applying the transformation as OpenVDB is a more feature-rich and mature frameworks. For example generating platonic solids in OpenVDB and transforming them to NanoVDB is significantly faster than generating them natively because only OpenVDBs generation functions are multi-threaded. Depending on the complexity of the simulation this process may require significant amounts of time. However within the context of semiconductor process simulation the majority of time is spent on calculating ray intersections as there is no need to finish all calculations within a given frame time.

It should also be noted that a GPU's Video-RAM is usually much smaller than regular RAM. This not only limits the maximum grid size but also the number of rays that can be stored or buffered on GPU. The rays in NanoVDB have a memory-footprint of 56 bytes per ray. The GPU used for this benchmark has 16GB of memory and can therefore theoretically hold up to 285 million rays (omitting grid, kernels, etc.). Using the measured performance of 117 MRps the computation of all rays would complete within 2.5s. In order to go beyond that limit new rays need to be generated during the simulation. Performing this step on CPU would cause an additional load 6.4 GB/s on the PCI bus. This number also increases if more than one GPU is used. Therefore ray generation should be performed on GPU as well. However this is not covered by this benchmark as this problem was not considered in the design phase of the experiment.

4.2 Potential performance improvements

Nvidia GPUs follow a SIMT [4] hardware architecture where threads are grouped in 'warps'. Each warp usually contains 32 synchronized threads which all execute the same instruction. In order to

[3] computer graphics

[4] Single Instruction Multiple Thread

execute `if-else`-statements both branches have to be evaluated by a warp consecutively while a mask is used to disable the the false-path in each thread. [3, Chapter 3.6.3] Therefore every branch is potentially wasteful and should be avoided by using branchless patterns.

Alg. 6 shows the core loop if NanoVDB's raytracing function which contains 2 `if`-branches. Additional branches are present in various subroutines such as `RoundDown()` or `hdda.update()`. Furthermore two `while`-loops are present which in the context of warps behave just like branches. I.e.: Each warp must 'wait' until all 32 threads have finished the loop. The outer loop is used while the ray is outside the narrow band. Once the ray enters a narrow band the inner loop is active and the value of each voxel is evaluated to detect an intersection. Therefore as long as any ray of a warp remains within a narrow-band, the computation of all other rays is essentially halted.

Modern compilers are likely able to optimize simple conditions such as `RoundDown()` to prevent branching. However more complex operations may increase the likelihood of wasteful branches. One way to achieve this could be by separating the nested while-loops such that only rays outside of the narrow band are processed. Once a ray enters the narrow-band it is handled by a different warp or kernel.

As mentioned above, NanoVDBs Ray-Class has a memory footprint of 56 bytes [5]. A ray consists at minimum of a pair of 3D-vectors or a set of 6 floating point numbers. Using single precision floats (4 bytes) each ray requires at least 24 bytes. This means that NanoVDB rays require almost twice as much memory. Therefore a better data structure could be introduced to reduce the memory footprint. This new structure can also be used to make use of more efficient memory alignments.

Since 2018 NVIDIA is also selling GPUs with hardware accelerated raytracing capabilities (RTX 20X0 and 30X0 series). Due to its proprietary license the exact functionality of this technology is not part of the public domain but the technology seems to be a hardware implementation of NVIDIA's OptiX pipeline. A recent analysis [6] also shows that a performance increase of up to one order of magnitude is possible in certain raytracing scenarios. However additional research and testing is required to determine if NanoVDB is compatible with this technology. It should be noted that the GPU used for this benchmark also contains 40 RT cores. Therefore it should be possible to repeat this benchmark on the same platform.

5 Conclusions

The benchmark shows that NanoVDB runs approx. 1.5x as fast as OpenVDB on the same CPU in a hypothetical level-set raytracing scenario within the context of semiconductor process simulation. Launching the NanoVDB kernel on a comparable GPU achieves approx. 6.2x faster performance than OpenVDB. However in its current form NanoVDB is strictly limited to voxel-level accuracy and performs no additional steps to increase the accuracy of level set raytracing intersections.

Migrating code from OpenVDB to NanoVDB requires significant changes to the codebase. Therefore if GPUs are to be utilize if might be more efficient to choose NanoVDB and only convert to OpenVDB when needed.

A preliminary analysis of NanoVDBs code shows that improvements to performance are be possible. Furthermore recent benchmarks by independent researchers have shown that NVIDIAs new raytracing technology (RTX) may increase performance. However it is currently not known if NanoVDB is compatible with this technology.

[5] Measured using sizeof()

6 Appendix

Algorithm 1: NanoVDBs Coord class. Note that datatypes are hardcoded to integers

```
    // NanoVDB.h Line 859
    class Coord
    {
        int32_t mVec[3]; // private member data - three signed index coordinates
    public:
        using ValueType = int32_t;
        using IndexType = uint32_t;
        ...
```

Algorithm 2: NanoVDBs raytracing implementation.

```
template<typename RayT, typename AccT>
inline __hostdev__ bool ZeroCrossing(RayT& ray, AccT& acc, Coord& ijk, typename
    AccT::ValueType& v, float& t)
{
    if (!ray.clip(acc.root().bbox()) || ray.t1() > 1e20)
        return false; // clip ray to bbox
    static const float Delta = 1.0001f;
    ijk = RoundDown<Coord>(ray.start()); // first hit of bbox
    HDDA<RayT, Coord> hdda(ray, acc.getDim(ijk, ray));
    const auto        v0 = acc.getValue(ijk);
    while (hdda.step()) {
        ijk = RoundDown<Coord>(ray(hdda.time() + Delta));
        hdda.update(ray, acc.getDim(ijk, ray));
        if (hdda.dim() > 1 || !acc.isActive(ijk))
            continue; // either a tile value or an inactive voxel
        while (hdda.step() && acc.isActive(hdda.voxel())) { // in the narrow band
            v = acc.getValue(hdda.voxel());
            if (v * v0 < 0) { // zero crossing
                ijk = hdda.voxel();
                t = hdda.time();
                return true;
            }
        }
    }
    return false;
}
```

References

[1] Wil Braithwaite and Ken Museth. *Accelerating OpenVDB on GPUs with NanoVDB*. 2020. URL: https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/ (visited on 03/19/2022).

[2] Paul Ludwig Manstetten. "Efficient flux calculations for topography simulation". PhD thesis. Wien, 2018.

[3] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439.

[4] Ken Museth. "NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation". In: *ACM SIGGRAPH 2021 Talks*. SIGGRAPH '21. Virtual Event, USA: Association for Computing Machinery, 2021. ISBN: 9781450383738. DOI: 10.1145/3450623.3464653. URL: https://doi.org/10.1145/3450623.3464653.

[5] Ken Museth et al. "OpenVDB: An Open-Source Data Structure and Toolkit for High-Resolution Volumes". In: *ACM SIGGRAPH 2013 Courses*. SIGGRAPH '13. Anaheim, California: Association for Computing Machinery, 2013. ISBN: 9781450323390. DOI: 10.1145/2504435.2504454. URL: https://doi.org/10.1145/2504435.2504454.

[6] VV Sanzharov, Vladimir A Frolov, and Vladimir A Galaktionov. "Survey of nvidia rtx technology". In: *Programming and Computer Software* 46.4 (2020), pp. 297–304.