Effective NC machining simulation with OptiX ray tracing engine

Marc Jachym, Sylvain Lavernhe, Charly Euzenat, Christophe Tournier

Effective NC machining simulation with OptiX ray tracing engine

Marc Jachym · Sylvain Lavernhe · Charly Euzenat · Christophe Tournier

Abstract The manufacturing of high added-value products in multi-axis machining requires advanced simulation in order to validate the process. Whereas CAM software editors provide simulation software that allows the detection of global interferences or local gouging, research works have shown that it is possible to consider multi-scale simulations of the surface, with a realistic description of both the tools and the machining path. However, computing capacity remains a problem for interactive and realistic simulations in 5-axis continuous machining. In this context, using general-purpose computing on graphics processing units as well as Nvidia OptiX Ray Tracing Engine makes it possible to develop a robust simulation application. Thus, the aim of this paper is to evaluate the use of Nvidia OptiX Ray Tracing Engine compared to a fully integrated CUDA software, in terms of computing time and development effort. Experimental investigations are carried out on different hardware such as Xeon CPU, Quadro4000, Tesla K40 and Titan Z GPUs. Results show that the development of such an application with the OptiX development kit is very simple and that the performances in roughing simulations are very promising. Developed software as well as dataset can be downloaded from http://webserv.lurpa.ens-cachan.fr/simsurf.

Keywords Machining simulation · ray tracing · GPU computing · CUDA architecture · OptiX

Christophe Tournier
LURPA, ENS Paris-Saclay, Université Paris Sud, Université Paris-Saclay, 94230 Cachan, France
Tel.: +33 1 47 40 27 52
Fax: ++33 1 47 40 22 20
E-mail: christophe.tournier@ens-paris-saclay.fr

1 Introduction

In molds and dies industry, simulation of machining process is mandatory to validate the tool path generated with the CAM software before launching the production of parts with very high added value. Indeed, machining operations including roughing, reworks and finishing are particularly time-demanding, especially for large size parts as for example in the automotive industry. Thus, the occurrence of defects in the final stages of the process has a dramatic impact on manufacturing companies. CAM software editors therefore provide cutting simulation applications that allow to validate the paths from a macroscopic point of view, i.e. to test the presence of collisions. However, these simulations don't incorporate any features of the actual process likely to deteriorate the surface finish during machining operations. Finally, these simulations do not provide the accuracy required within a reasonable time or the possibility for the user to select an area in which he would have a greater precision. On the other hand, high-performance simulation software prototypes are developed in laboratories in order to offset the preceding shortcomings, but they require significant computer resources. Many methods have been published in the literature to perform machining simulations. Some of them are based on partitioning the space whether by lines [6], by voxels [5] or by planes [11], other are based on meshes [3]. Previous works have shown that it is possible to simulate the resulting geometry of the surface with Z- or N-buffer methods applied to a realistic description of both the tools and the machining path in a few minutes [7]. Simulation results are very close to experimental results but the simulated surfaces have an area of some few square millimeters with micrometer resolution. Therefore, to overcome the limits in

terms of computing capacity, some works deal with the use of GPGPU (general-purpose computing on graphics processing units) and especially Nvidia GPU (graphics processing units) and CUDA (Compute Unified Device Architecture) technology in the field of manufacturing simulation [4,9]. In this context we have developed a software called SIMSURF1 in order to simulate very quickly a selected machined area at different scales chosen by the user [1]. This tool, which is very fast, is based on the Z-buffer method and relies on GPU/Cuda technology or many CPU cores [10]. However, the development of such applications requires an extended and deep knowledge of these architectures. Low-level CUDA library has to be used and every aspect of the multi GPU-core architecture on which CUDA is based has to be managed, from the distribution of the parallel calculations on the cores to the memory exchanges between the CPU and the GPU.

This is why the use of an application framework such as NVIDIA OptiX Ray Tracing Engine would facilitate the development of high-performance ray tracing applications [2]. Thus, the proposed SIMSURF2 approach aims at taking advantage of the OptiX Ray Tracing Engine in order to facilitate the writing of a machining simulation software based on the GPU parallel calculation platform. For this purpose, OptiX provides integrated features such as the possibility of modifying the acceleration structure for each tool posture, i.e. position and orientation in the 3D space, without recalculating it completely. With OptiX and the specialized subset OptiX Prime, which is dedicated to the high-speed calculation of intersections between rays and triangles, gains are expected regarding the software development speed as well as regarding the optimization in the use of the CUDA architecture which, in turn, could accelerate the whole machining simulation process. We propose in this article to compare the performances of the OptiX Ray Tracing engine with the developments previously achieved in SIMSURF1 and updated here as an implementation for NVIDIA Tesla K40 and Titan Z GPU. The rest of the paper is organized as follows: the computation algorithm and the low-level approach used in SUMSURF1 are summarised in section 2, OptiX ray tracing engine and its associated features are described in section 3 and section 4 is dedicated to the experimental investigations and benchmarking of both approaches.

2 Computation algorithm and CUDA architecture

The computation algorithm relies on the Z-buffer method which consists in partitioning the space around the surface to be machined in a set of lines, which are equally distributed in the x-y plane and oriented along the z-axis. The machining simulation is carried out by computing the intersections between the lines and the tool along the tool path. The geometry of the tool is modeled by a triangular mesh including cutting edges, which allows to simulate the effect of the rotation of the tool on the surface topography. The tool path is either a 3-axis tool path with a fixed tool axis orientation or a 5-axis tool path with variable tool axis orientations. In order to simulate the material removal, all the intersections with a given line are compared and the lowest is registered. The complete simulation requires the computation of the intersections between the N lines ($\sim$1.e6) and the T triangles ($\sim$1.e4) of the tool mesh at every tool posture P ($\sim$1.e6) on the tool path. Thus simulations with 1.e16 potential intersections to compute are commonly encountered without taking into account the use of bounding boxes. For instance, in the case **blade roughing** described in Tab 1, the computation time is about 11 hours without any parallelization on the Xeon CPU described in section 4.

The developed algorithm can run on both CPU and GPU hardware. The implementation of the SIMSURF1 algorithm on CPU is based on the use of the OpenMP API and "for" loops as well as Streaming SIMD Extensions (SSE) instructions. The optimization of the code executed on GPUs is more difficult and it requires to divide the computation into threads and then blocks to take advantage of CUDA's massively parallel architecture. Indeed, the strength of the CUDA programming model lies in its capability to achieve high performance through its massively parallel architecture (Fig. 1). In order to achieve high throughput, the algorithm must be divided into a set of tasks with minimal dependencies. Tasks are mapped into lightweight threads, which are scheduled and executed concurrently on the GPU. The 32 threads within a same warp are always executed simultaneously; maximum performance is therefore achieved if all the 32 threads execute the same instruction at each cycle. Warps are themselves grouped into virtual entities called blocks; the set of all blocks forms the grid, representing the parallelization of the algorithm. Threads from the same block can be synchronized and are able to communicate efficiently using a fast on-chip memory, called shared memory, whereas threads from different blocks are executed independently and can only communicate through global (GDDR) memory of the GPU. The number of threads executed simultaneously can be two orders of magnitude larger than on a classical CPU architecture. As a consequence, task decomposition should be fine-grained opposed to the traditional coarse-grained approach for

CPU parallelization. The basic algorithm consists in determining whether there is an intersection between a line and a triangle associated to a tool posture. The intersection algorithm is based on triangle rasterization [12]. If this algorithm requires more operations and memory than the one developed in [8], this disadvantage is compensated by an extremely fast inclusion test of the intersection in each triangle. Given these three variables on which the algorithm iterates during the sequential computation, there are numerous possible combinations to affect threads and browse the set of lines, triangles and positions. Only one possibility is used hereafter which is the most appropriate for macro scale simulations [1]. Each thread is assigned to a position of the tool and applies the Z-buffer algorithm for every triangle of the tool mesh for this position. The pseudo code of both algorithms executed on the host (CPU) and on the device (GPU) is provided hereafter. The granularity of tasks is high: if the number of triangles to be processed is large, each thread will run for a long time. If the computation time between threads is heterogeneous, some threads of a warp may no longer be active, and therefore the parallelism is lost. A thread may affect the cutting height of several lines so a line can be updated by multiple threads and global memory access conflicts appear. Atomic operations proposed by CUDA are then used to allow concurrent update of the height of the lines.

Algorithm 1 Simsurf1 pseudo-code for the CPU host

```
1: Lines ← load Z-buffer description from file
2: path ← load tool path from file
3: toolMesh ← load tool description from mesh file

4: nbThreads ← query GPU configuration

5: allocate GPU memory for Z-buffer
6: allocate GPU memory for toolMesh
7: allocate GPU memory for toolpath

8: matrix transformation ← Compute matrix transformation from tool path file
9: move piece, toolMesh, matrix decriptions from CPU memory to GPU memory

10: while every block of tool positions not done do
11:     allocate nbThreads to GPU CUDA kernels for the current block
12:     launch the parallelized threads (GPU CUDA kernels)
13: end while

14: move Z-buffer results from GPU memory to CPU memory
15: create the STL file resulting from the intersections
```

Algorithm 2 Simsurf1 pseudo-code for the GPU parallelized CUDA kernels

```
1: for every triangle do
2:     apply transformation matrix to the triangle
3:     compute the 2D bounding box circumscribed to the triangle in the xy plane

4:     for each line in the bounding box do
5:         perform the actual intersection between lines and triangles

6:         atomicMin ← Z-buffer height updating by using atomic operation
7:     end for
8: end for
```

3 OptiX ray tracing engine

Nvidia OptiX is an engine for ray tracing 3D-rendering. It allows the developer to concentrate on the objects in a scene whose geometry is defined by the algorithms for the ray-object intersections and on the behavior of the light when it encounters some material. Those elements are the entry points to the ray-tracing parallel calculation engine that executes on the CUDA architecture. The OptiX engine is based on acceleration structures, which are hierarchies of bounding boxes, to determine which of the scene areas are empty and do not need any calculation. OptiX Prime is an OptiX's subset which is dedicated to the high-speed calculation of intersections between rays and triangle meshes. There is no notion of material properties in OptiX Prime and thus, it has nothing to do with optic rules and 3D object rendering. Rather, it provides a hopefully optimized way to use a hidden acceleration structure suited to triangle meshes and to perform a high-speed rays-triangles in-

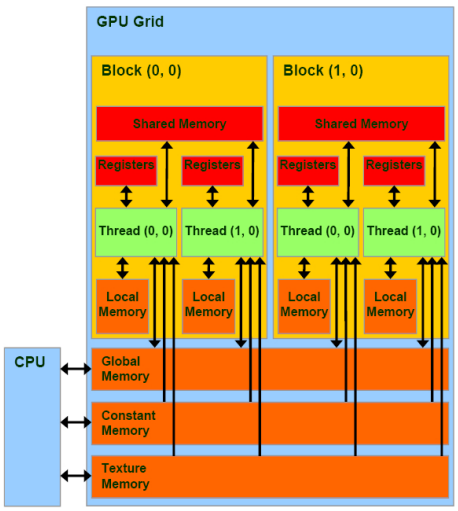

Fig. 1 Cuda architecture

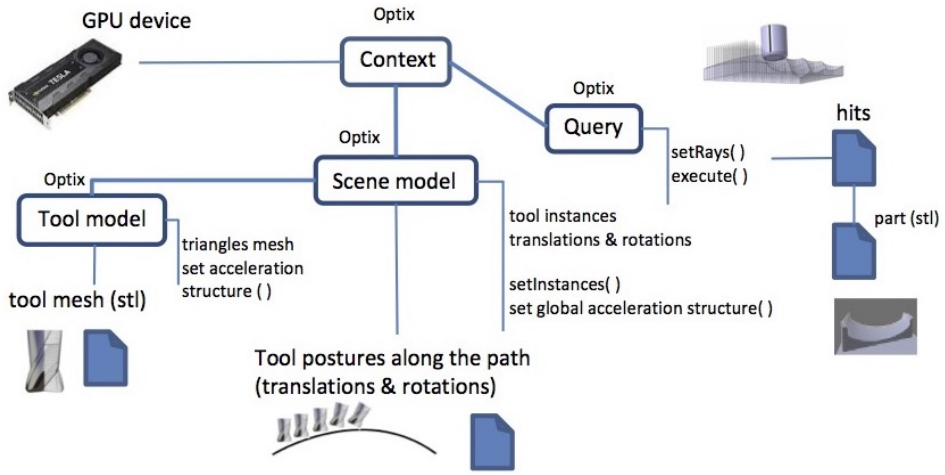

Fig. 2 OptiX engine process overview

tersection on the underlying CUDA architecture. By hidden, we mean hidden to the software developer who is freed from researching methods for reducing the number of possible intersections that the GPU will have to calculate.

Within SIMSURF1, the software programmer has to devise by himself clever methods to determine empty areas in the scene in order to avoid that the GPU would have to calculate every possible intersection between any ray and any triangle. Within SIMSURF2, the programmer has to choose between different possibilities regarding acceleration structures and traversal methods, whether he has to manage static vs dynamic scenes or whether his objects are defined with geometric formulas or meshes. OptiX Prime simplifies this greatly because the best possible choices, regarding Nvidia experience in acceleration structures and traversal algorithms, have been made for a static scene based on triangle meshes (Fig. 2). The calculation of the acceleration structures is the slowest stage of the process and, with previous OptiX Prime versions, an acceleration structure has to be built at every step of the loop even if the geometry of the tool is not changed but is simply moved along the planned path. This problem has been addressed with OptiX Prime 3.9 which offers a new possibility called instancing. From a model object; in the sense of Object Oriented Programming; which associates a triangle mesh and its dedicated acceleration structure, instancing composes complex scenes using existing triangle models. Then OptiX Prime is able to create a global acceleration structure for the whole scene without duplicating the elementary models' description. The programmer has to create a memory structure to associate each instance of a model object in the scene with a transformation descriptor, i.e. a translation, a rotation or/and a scaling matrix. The fact that the basic model description is not duplicated in memory allows to process much bigger path buffers.

The surface simulation of a 5-axis machining operation requires to move and rotate the tool. Regardless of the machine architecture, the Optix framework allows to define a transformation matrix for each time step. The initial tool axis orientation is defined by $\begin{bmatrix}0 & 0 & 1\end{bmatrix}^T$. Every line of the tool path file is made of three coordinates x, y, z for the tool's translation plus three coordinates i, j, k for the tool's rotation. All values are related to the global coordinate system. The rigid body transformation matrix is defined in order to move the initial tool mesh at the required location under a given orientation. For a given axis of rotation $\mathbf{u}$ and angle φ (Fig. 3), the rotation of a vector $\mathbf{x}$ is given by :

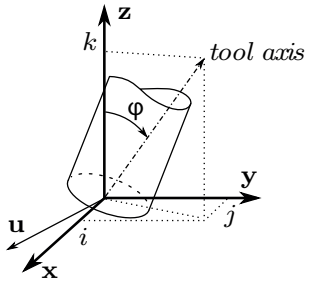

Fig. 3 Tool orientation parameters

$$\mathbf{v} = cos(\varphi)\mathbf{x} + (1 - cos(\varphi))(\mathbf{x}.\mathbf{u}).\mathbf{u} + sin(\varphi)(\mathbf{u} \times \mathbf{x}) \quad (1)$$

Applying this equation for $\mathbf{u} = \begin{bmatrix}j & -i & 0\end{bmatrix}^T$ and φ defined by $cos(\varphi) = k$ and $sin(\varphi) = -\sqrt{i^2 + j^2}$ lead to the following transformation matrix :

$$\begin{pmatrix} \frac{j^2+k(1-j^2)}{i^2+j^2} & \frac{-ij(1-k)}{i^2+j^2} & i & x \\ \frac{-ij(1-k)}{i^2+j^2} & \frac{-i^2+k(1+i^2)}{i^2+j^2} & j & y \\ -i & -j & k & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The algorithm sketching OptiX Prime usage is provided here after in pseudo-code format and describes the part-tool intersection main program. In order to manage large NC files, tool paths are split into blocks that fit in the GPU memory. It is important to note that the development and implementation of the SIMSURF2 algorithm in OptiX took about one month versus 6 months for the development of SIMSURF1.

Algorithm 3 Optix Prime pseudo-code

```
Create-OptiX-Context (GPU-context)
toolMesh ← Create tool mesh from stl file
toolModel ← Create OptiX Prime Tool Model
toolModel.Create_acceleration_structure()
path ← Create transformations buffer from tool path
       rotations file
boundingBox ← Compute the bounding box of the
              whole scene
raysBuffer ← Create vertical rays for the bounding
             box according to the chosen entensity
closestHitsBuffer ← Create the general hits buffer
for each block do
    block_number ← current-pos / NB_POS_PER_BLOCK
    toolInstances ← Create a container for the
                    models of all tool positions
    transformations ← Create tool position con-
                      tainer
    for every path position in current block do
        transformations[currentPos] ← transform matrix
        toolInstances[currentPosition] ← toolModel
    end for
    global_scene ← Create the model of the whole scene
                   with the association of toolInstances
                   & transformations
    global_scene.Create_global_acceleration_structure()
    hitBuffer ← Init buffer for current block
    Do perform the actual ray tracing on the global-
    scene from the raysBuffer
    closestHitsBuffer ← Update with block's hitBuffer
    Release the memory used by the global-scene
end for
Create the stl resulting file from the closestHitsBuffer
```

4 Experimental investigations

The objective is to compare the computation time obtained with SIMSURF1 and SIMSURF2 for different NC simulations with different hardware. Several test cases (table 1) have been investigated in 3 or 5-axis milling in roughing and finishing with variations in the number of tool postures on the tool path and triangles in the mesh. The Z-buffer is computed with a grid of 1024x1024 lines covering the X-Y trajectory range. Results are gathered in table 2 and in fig. 12.

- 3-axis roughing cases with filleted endmill and growing number of tool postures and air paths
  - Blade roughing (Fig. 4)
  - Mask roughing (Fig. 5)
  - Wave roughing (Fig. 6)
- 5-axis finishing cases with ball endmill and growing number of tool postures
  - Blade finishing (Fig. 7)
  - Wave finishing (Fig. 8)
- 3-axis finishing cases with ball endmill and growing number of tool postures
  - Mask finishing (Fig. 9)
  - Aero finishing (Fig. 10)

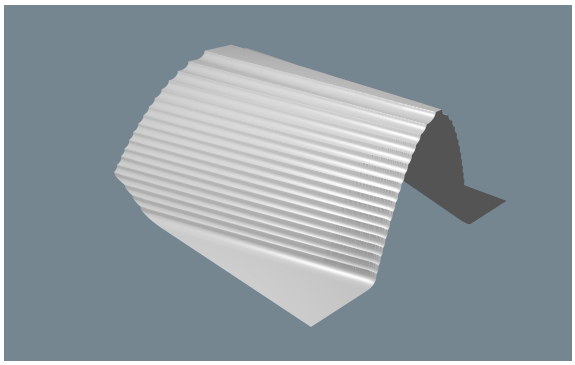

Fig. 4 Blade roughing simulation result

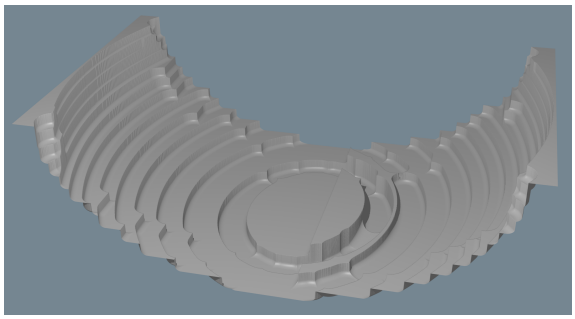

Fig. 5 Ski mask mold roughing simulation result

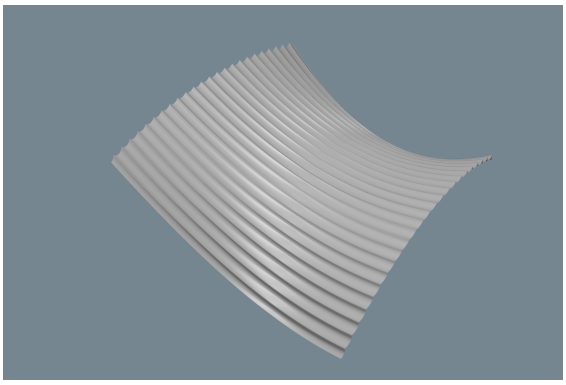

Fig. 6 Wave surface roughing simulation result

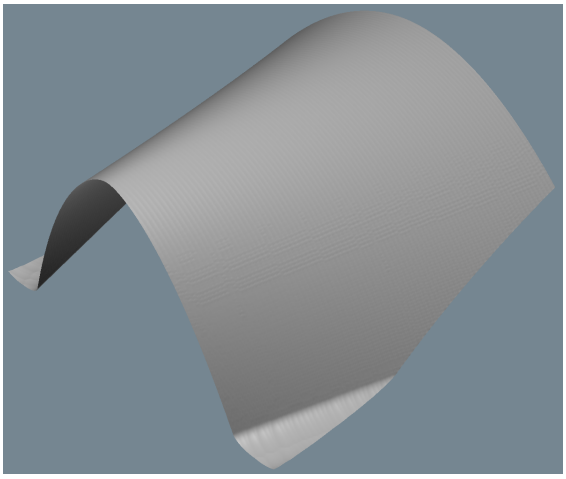

Fig. 7 Blade 5-axis finishing simulation result

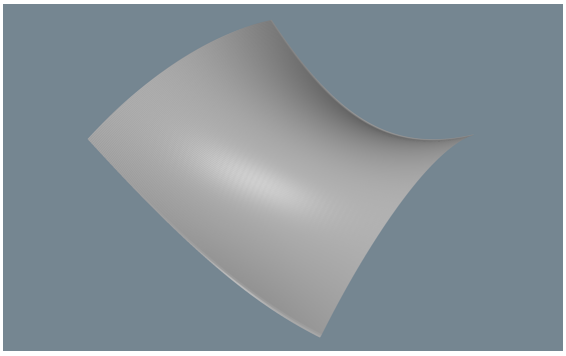

Fig. 8 Wave surface finishing simulation result

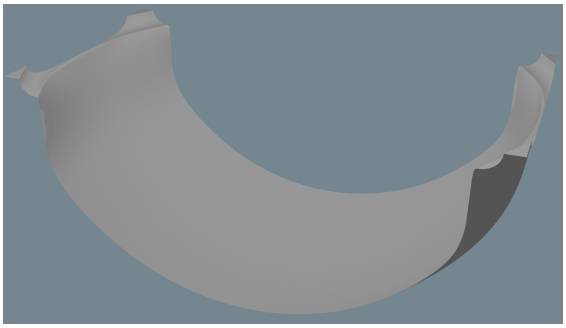

Fig. 9 Ski mask mold 3-axis finishing simulation result

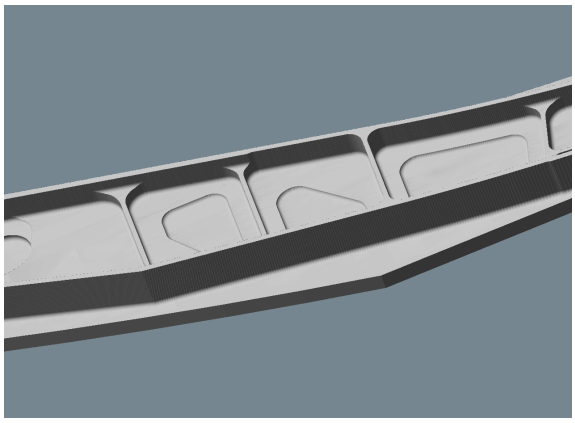

Fig. 10 Aeronautic part finishing simulation result

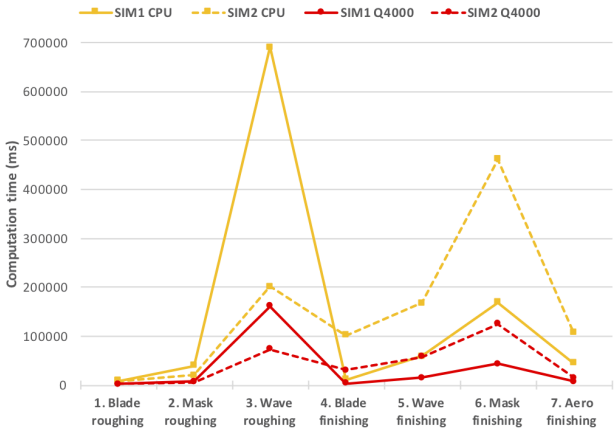

Fig. 11 Computation time on test cases for Xeon CPU and Quadro4000 GPU with a 1024x1024 Z-buffer

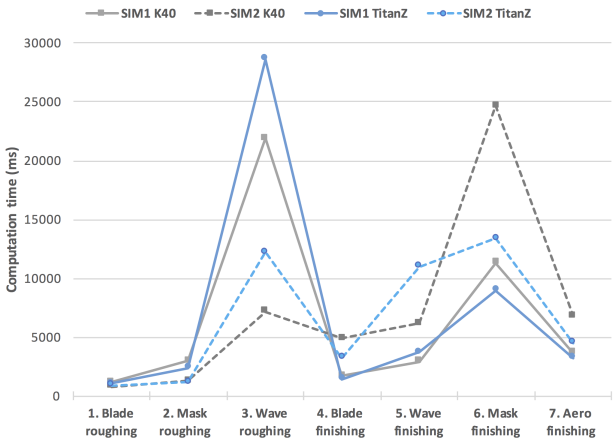

Fig. 12 Computation time on test cases for K40 GPU and TitanZ GPU with a 1024x1024 Z-buffer

Table 1 Test cases description

Case	Tool geom.	Triangles T	Postures P	CAM (s)
1. Blade rough.	Torus	25904	47837	25
2. Mask rough.	Torus	25904	345848	380
3. Wave rough.	Torus	25904	8.e6	3670
4. Blade finish.	Sphere	12482	53667	245
5. Wave finish.	Sphere	12482	1.e6	370
6. Mask finish.	Sphere	12482	3015072	450
7. Aero finish.	Sphere	12482	27425026	2520

NC simulations have been carried out with the following hardware configurations:

- XEON CPU : Intel Xeon Processor E5-1620V3, 3.5Ghz, 31 Gflops DP, 4 cores, 8 threads, 10Mo SmartCache
- Quadro 4000 GPU : 950 MHz, 486 SP, SP89.6 GB/sec Memory bandwidth, 2GB (GDDR5), 256 CUDA Cores
- Tesla K40 GPU (one GK110 GPU): 745 MHz, 4.29 Tflops SP, 288 GB/sec Memory bandwidth, 12GB (GDDR5), 2880 CUDA cores
- GeForce GTX Titan Z (two GK110 GPU) : 705 MHz, 4.06 Tflops SP, 288 GB/sec Memory bandwidth, 2x 6GB (GDDR5), 2x 2880 CUDA cores

One can notice that the implementation of SIMSURF1 does not take advantage of the two GPUs of the GeForce GTX Titan Z, only one GPU is used in this case. It explains the closeness of the following results with both GPU. The operating System is XUbuntu 14.04 64 bits which is based on the Linux kernel 3.5, and the programming language is C++ compiled with gcc (4.8.4). Regarding software configurations, SIMSURF1 relies on CUDA version 7.0 and SIMSURF2 on CUDA 7.5 and OptiX Prime 3.9.

The three roughing cases are those for which SIMSURF2 is the most efficient, regardless of the hardware used, which is a very satisfactory result. In addition, the higher number of tool positions, the greater the gains in computation time compared to SIMSURF1. The reason is that Optix uses an acceleration structure that minimizes the number of intersections to be calculated. Roughing paths contain a large number of tool positions that are not involved in the generation of the final shape. Thus, only a reduced number of positions in each Z-level of the path is evaluated in the intersection calculation. Since the construction of the acceleration structure is time-consuming, the more "air" tool positions in the roughing path, the greater the gains, as shown by the experimental results. The gain obtained between the worst roughing simulation with SIMSURF1 (Xeon CPU) and the best simulation with SIMSURF2 (K40 GPU) is around 100.

For 5-axis finishing simulation including translations and rotations of the tool, i.e. Blade finishing and Wave finishing, SIMSURF1 is faster than SIMSURF2 whatever the hardware. The performance difference is greater with the Xeon CPU and Q4000 GPU than with other hardware. It seems that the generation within OptiX of the scene including rotations of the instances of the tool takes a lot of computing resources. The increase in the number of positions to be processed leads to a proportional increase in computing time, except for the TitanZ GPU for which the SIMSURF2 method is more penalized.

Regarding 3-axis finishing simulations, i.e. Mask finishing and Aero finishing.1, the results between SIMSURF1 and SIMSURF2 are similar for all devices. In the case of Mask finishing, the speed-up, i.e. the ratio between SIMSURF2 and SIMSURF1 computation times is ranging from 2.89 (Q4000) to 1.49 (Titan Z). In this case, the ratio between the machined surface area and the tool dimension is low. This implies that a lot of intersections will be computed between triangles and lines. In the case of Aero finishing, SIMSURF1 is still faster but the speed-up is lower whatever the device. In this case, the number of intersections between lines and triangles per tool posture is low, around 7, and SIMSURF1 takes advantage of a simple bounding box for each tool posture, whereas OptiX generates an acceleration structure for the 27 million tool postures before launching the intersections computation. However, as mentioned above, the threads' computation times are heterogeneous and then the parallelisation is lost in SIMSURF1 [1], leading to comparable performances.

At last, for Aero finishing.2 the size of the Z-buffer is increased to 10000x10000 (table 3), leading to numerous intersections per triangle and a large acceleration structure for SIMSURF2, which again loses the advantage over SIMSURF1.

5 Conclusion

A comparison of two ray-tracing GPU and CPU implementations for NC simulations has been proposed in this paper. The first approach is based on the direct use of CUDA which requires rather steep learning curve and expertise to achieve high performances. The second one is based on the OptiX ray tracing engine which provides simpler application programming interfaces to compute the rendering of machining scenes. Experimental investigations have been conducted on 4 different hardware. They have shown that the approach based on OptiX

Table 2 32 bits computation times (ms) on test cases for a 1024x1024 Z-buffer

Case	Xeon CPU			Quadro 4000 GPU			Tesla K40 GPU			Titan Z GPU		
	Sim1	Sim2	SU	Sim1	Sim2	SU	Sim1	Sim2	SU	Sim1	Sim2	SU
1. Blade roughing	8353	8440	1.01	2606	2215	0.85	1166	812	0.69	1102	938	0.85
2. Mask roughing	39304	20538	0.52	7607	5564	0.73	2986	1272	0.42	2398	1198	0.5
3. Wave roughing	690700	201900	0.29	161392	72754	0.45	21880	7190	0.33	28651	12225	0.43
4. Blade finish.	10606	101311	9.55	3516	30204	8.59	1703	4925	2.89	1441	3233	2.24
5. Wave finish.	59523	167491	2.81	15037	57767	3.84	2960	6150	2.08	3727	10998	2.95
6. Mask finish.	168520	461582	2.74	43212	125002	2.89	11314	24671	2.18	8984	13374	1.49
7. Aero finishing	45022	107261	2.38	7847	15156	1.93	3698	6815	1.84	3273	4510	1.38

Table 3 32 bits computation times (ms) on test cases for a 10000x10000 Z-buffer

Case	Xeon CPU			Quadro 4000 GPU			Tesla K40 GPU			Titan Z GPU		
	Sim1	Sim2	SU	Sim1	Sim2	SU	Sim1	Sim2	SU	Sim1	Sim2	SU
8. Aero finishing	206284	3349908	16.3	57674	522275	9.06	30961	163565	5.3	28190	130312	4.6

is the most straightforward to implement and the most competitive in 3-axis roughing simulations for all hardware. However, 5-axis configurations remain a problem for OptiX due to the transformation matrix applied for every posture of the tool (position and rotation). In 3-axis cases, computation times between SIMSURF1 and SIMSUR2 are much closer, especially on GPU hardware, which can be considered as a positive outcome regarding the software development simplicity of SIMSURF2.

Acknowledgements We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research as well as the support of the Farman Institute (CNRS FR3311).

References

1. Abecassis, F.; Lavernhe, S.; Tournier, C.; Boucard, P-A.: Performance evaluation of CUDA programming for 5-axis machining multi-scale simulation. Computers in Industry 71, 1-9 (2015).
2. CUDA C Programming Guide, NVIDIA, 2012 http://developer.nvidia.com/cuda/
3. He, W.; Bin, H.: Simulation model for CNC machining of sculptured surface allowing different levels of detail. The International Journal of Advanced Manufacturing Technology 33(11-12), 1173-1179 (2007)
4. Inui, M.; Umezu, N.; Shinozuka, Y.: A comparison of two methods for geometric milling simulation accelerated by GPU. Transactions of the institute of systems, Control and Information Engineers 6(3), 95-102 (2013)
5. Jang, D.; Kim, K.; Jung, J.: Voxel-based virtual multi-axis machining. International Journal of Advanced Manufacturing Technology 16(10), 709-713 (2000)
6. Jerard, R.B., Hussaini, S.Z., Drysdale, R.L.: Approximate methods for simulation and verification of numerically controlled machining programs. The Visual Computer, 5(6), 329-348 (1989).
7. Lavernhe, S.; Quinsat, Y.; Lartigue, C.; Brown, C.: Realistic simulation of surface defects in 5-axis milling using the measured geometry of the tool. International Journal of Advanced Manufacturing Technology, 74(1-4), 393-401 (2014)
8. Moller, T.; Trumbore, B.: Fast, minimum storage ray-triangle intersection. J. Graph. Tools, 2(1), 2128 (1997)
9. Morell-Gimenez, V.; Jimeno-Morenilla, A.; Garcia-Rodrguez, J.: Efficient toolpath computation using multi-core GPUs. Computers in Industry 64(1), 50-56 (2013)
10. Parker, S.; Bigler, J.; Dietrich, A.; et al: OptiX: a general purpose ray tracing engine. ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH, 2010, 29(4), Article 66, 13 pages, (2010)
11. Quinsat, Y.; Sabourin, L.; Lartigue, C.: Surface topography in ball end milling process: description of a 3D surface roughness parameter. Journal of Materials Processing Technology, 195(1-3), 135-143 (2008)
12. Zhang, W.; Majdandzic, I.: Fast triangle rasterization using irregular Z-buffer on CUDA, (2010)