Array 29 (2026) 100701

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/locate/array

Efficient tool path computing for Industry 5.0: Application to turning lathe machining

Héctor Migallón [a,*], Antonio Jimeno-Morenilla [b], Eduard Duta-Costache [b], José-Luis Sánchez-Romero [b]

[a] Computer Engineering Department, Miguel Hernández University, Elche, Spain
[b] Systems Engineering and Automation Department, Miguel Hernández University, Elche, Spain

ARTICLE INFO

Keywords:
Industry 5.0
Customized manufacturing
Real time toolpath generation
Parallel computing
Multicore processing
Manycore processing

ABSTRACT

This paper presents an efficient approach to toolpath generation tailored to the needs of Industry 5.0, with a focus on turning lathe machining. The study addresses the challenge of rapidly and accurately generating helical toolpaths in personalized manufacturing, where traditional sequential methods often become computational bottlenecks. To overcome this limitation, we propose efficient parallel implementations of the Virtual Digitizing (VD) algorithm, specifically designed to accelerate the computation of machining trajectories on both multicore and manycore architectures. The multicore implementation achieves notable speedups, especially when execution is properly tuned. The manycore strategy explores both asynchronous (coarse-grained) and synchronous (fine-grained) execution models. In the asynchronous method, independent trajectory computations are assigned to separate CUDA threads, whereas the synchronous method further parallelizes the internal processing of each trajectory point, providing finer computational granularity. Experimental evaluations conducted on authentic industrial shoe last models reveal notable gains in computational efficiency. The manycore implementation achieves up to $70x$ acceleration on low-end GPUs, over $80x$ on high-range devices and over $270x$ on state-of-the-art GPU devices when compared to their respective CPU-based computations. Although the synchronous method introduces additional complexity, it delivers the best performance on powerful GPU platforms, whereas the asynchronous method is better suited for resource-constrained systems. Therefore, the study concludes that the optimal parallelization strategy depends on the available hardware.

1. Introduction

In the context of Industry 5.0, the manufacturing process has evolved towards a more agile and customized approach, tailored to meet the specific demands of the end customer [1]. This shift requires flexible and efficient systems that can be rapidly reconfigured to handle diverse production runs, achieving an unprecedented level of agility in the manufacturing sector. This new paradigm aims not only for efficiency and automation, but also for the integration of creativity and customization into production processes. It reinforces the need for companies to adopt emerging technologies to ensure that manufacturing practices are not only adaptable, but also efficient and responsive to changes in real time [2].

In this customized production approach, where design is clearly guided by the needs of the end user or customer, it becomes essential to implement simulation processes, not only to provide immediate graphical feedback, but also to ensure seamless integration with manufacturing processes. This involves the efficient generation of geometries and machining paths, particularly in sectors such as footwear design, orthopedic insole production, custom fashion accessories, and wearable technology, where rapid prototyping and adaptation to individual specifications are increasingly in demand [3]. Thus, machining simulation processes must not only operate quickly, but also be capable of real-time visualization as model design parameters are modified. As model design parameters change, machining simulation processes must be able to update and reflect these modifications in real time [4].

Furthermore, Industry 5.0 places growing emphasis on adaptability and efficiency across the entire product lifecycle, from design to manufacturing. Although sustainability is often discussed within the Industry 5.0 framework, the focus of this work is strictly on improving computational efficiency and responsiveness in toolpath generation. Optimizing these processes is essential to meet productivity and real-time requirements in customized manufacturing scenarios [5].

The generation of optimized toolpaths is essential to meet the requirements of Industry 5.0. By leveraging modern computational

* Corresponding author.
E-mail address: hmigallon@umh.es (H. Migallón).

https://doi.org/10.1016/j.array.2026.100701
Received 12 November 2025; Received in revised form 18 January 2026; Accepted 29 January 2026
Available online 31 January 2026

systems, which are inherently parallel due to multicore and manycore architectures, it is possible to accelerate these processes, ensuring that machining paths are both efficient and accurate. Such optimization not only enhances productivity, but also enables greater flexibility and responsiveness to the dynamic demands of the market [6].

Path generation methods for machine tools often involve high computational complexity, particularly when high levels of accuracy are required. In traditional mass production environments, characterized by long runs of identical models, this computational time was not a critical issue, as it could overlap with production processes: the same toolpath was reused for hundreds or thousands of parts or products. However, under the customized manufacturing paradigm, where product series are unique or extremely limited, path generation time becomes a critical factor. It directly impacts production time, thereby increasing overall costs, and may even lead to bottlenecks. In such scenarios, accelerating toolpath computation becomes essential.

In this work, we present a use case in which it was necessary to accelerate the execution of a toolpath generation system called Virtual Digitizing (VD), typically used for generating helical toolpaths for machining lathes. This acceleration is implemented on both standard desktop systems and high-performance computing platforms. The objective is to integrate real-time toolpath computation into the workflows of key functional areas across the product development and manufacturing chain, thereby ensuring that computational demands do not compromise operational agility or responsiveness.

This paper is structured as follows. Section 2 reviews recent research on the acceleration of trajectory computation. Section 3 describes the basic method for generating machining trajectories using the VD algorithm. Section 4 presents the developed parallel implementations for both multicore and manycore architectures. Section 5 details machining experiments performed using real industrial shoe lasts. Finally, Section 6 is devoted to the discussion of the results, offering a detailed interpretation and contextualization of the findings and also discusses the boundaries the proposed approach, and Section 7 presents the main conclusions and outlines future research directions.

This section also discusses the main applicability boundaries and limitations of the proposed approach

2. State of the art in accelerated toolpath computation

The increasing complexity of CNC machining processes has driven the research community to explore parallel computing as a means to enhance tool path computation efficiency. While many of these efforts focus on traditional CNC machining, our work is situated in a different context, customized manufacturing in particular for the footwear industry, where similar computational challenges arise. In this context, a variety of approaches have emerged, exploiting multicore CPUs, GPUs, and hybrid architectures to accelerate optimization, planning, and simulation tasks.

Early work by Medina-Rodríguez et al. [7] proposed a parallel implementation of the Ant Colony Optimization algorithm to minimize the travel time of the cutting tool by formulating the toolpath planning as a special case of the Traveling Salesman Problem. A similar objective was pursued in [8], where a Genetic Algorithm was employed. Both approaches achieved significant reductions in manufacturing time compared to commercial algorithms.

Vivanco et al. [9] laid the foundation for leveraging multicore processors in automatized machining by partitioning numerical control software into concurrently executable threads, enabling the dynamic distribution of tasks across multiple cores. Similarly, Kaiser et al. [10] demonstrated that the usage of multicore processors for tool path generation can significantly reduce execution times.

Szczepanski et al. [11] focused on accelerating feedrate optimization for NURBS toolpaths in CNC machining using parallel computation. They employed a Guaranteed Convergence Particle Swarm Optimization (GC-PSO) algorithm combined with the Augmented Lagrangian Method to handle constraints like velocity, acceleration, and jerk limits of the machine's axes. To improve the time-consuming optimization process, the authors implemented a parallel version of their algorithm using OpenMP. Experimental results demonstrate significant speed-up compared to a sequential implementation, with the best performance achieved using four threads on a dual-core processor with hyper-threading.

Yi et al. [12] optimized 5-axis flank milling of ruled surfaces by combining geometric decomposition with a parallel Multi-Population Harmony Search (MPHS) algorithm. The main innovation is the MPHS algorithm parallel implementation using MPI. Each population is processed by a separate MPI process, enabling simultaneous evaluation and dramatically reducing computation time. MPI communication facilitates information sharing and coordination between populations.

Manycore architectures, predominantly driven by GPUs, also offer compelling capabilities for toolpath computing. A GPU version of a turning lathe toolpath algorithm can be found in [13]. However, the technology used at the time still did not allow for sufficient efficiency while maintaining the accuracy of the process. Abecassis et al. [14] explored different parallelization strategies for parallelizing the Z-buffer algorithm for 5-axis machining simulations. Their CUDA-based implementation significantly reduced simulation time, achieving relevant performance gains compared to CPU-based methods. The performance gain allows for real-time adjustments and more precise simulations; however, the parallelization strategy needs to be adapted depending on the simulation granularity.

Konobrytskyi et al. [15] also highlighted the potential of GPU acceleration in tool path planning for 5-axis CNC milling. They introduced a parallel discrete volumetric geometry representation that simplifies the parallelization process by moving complexity from algorithm design to data structure design. Their approach involves a 2-level hybrid geometry representation optimized for GPU computing. This representation is then used to solve multiple geometric problems by reformulating the operations to work with volumes instead of surfaces.

Similarly, Balabokhin and Tarbutton [16] presented an algorithm for generating tool paths to cover a given milling zone while avoiding gouging. They implemented various sub-algorithms using GPUs to accelerate the most time-consuming parts.

Wang et al. [17] proposed a method that leverages the parallel processing capabilities of GPUs to accelerate the computation of the objective function used for tool posture optimization. The results show a reduction of up to two orders of magnitude of the computing time.

Casagrande-Faust et al. [18] developed a new parallel method for zigzag tool-path generation on additive manufacturing. The method is divided into six steps, one of which is carried out by the CPU, and the other five are executed on a GPU system. Experimentation demonstrates that the parallel method achieves significant computational gain, being the best case 9.8 times faster than the serial version.

Hsieh and Chu [19] applied GPU parallelism to solve the problem of machining error control in 5-axis flank milling of complex patterns. A particle swarm optimization-based method is developed to generate a set of cutter locations which produce a minimal error on the machined surface. The error quantity provoked by each cutter location is computed in parallel by the GPU platform. Experimentation shows that the proposal outperforms previous optimization methods not only in solution quality but also in computation efficiency.

In [20] a GPU-implemented tool parameter optimization methodology is presented, aimed at minimizing tool orientation discontinuities. The methodology optimizes tool diameter and length, constrained by tool interference and tool axis continuity. Depending on the different parameters considered for optimization, the speedup obtained when comparing CPU and GPU computation rises to $68x$.

In [21], the problem of model orientation is first formulated as a multi-objective optimization problem, whose objective is minimizing building time, surface quality, and support area. The multi-objective

problem is transformed into a single-objective, linearly weighted optimization problem. Then, a Genetic Algorithm is applied to give a solution to the optimization problem, being the processing of the algorithm parallelized and implemented on GPU. Experimentation demonstrates that, when coping with complex models in additive manufacturing, the GPU implementation speeds up the process up to 50 times compared with CPU performance.

In [22], the research focuses on utilizing voxel-based techniques, which are commonly used in computer graphics, to represent and manipulate the workpiece during the simulation. By harnessing the parallel processing power of GPUs, the researchers achieve significant speed improvements in the simulation process.

In [23], researchers conclude that using manycore architectures can significantly improve the performance of metaheuristics for optimizing tool path computation in CNC machining. Concretely, the Discrete TLBO algorithm is utilized to generate optimal tool paths. Experimental results demonstrate that the parallel implementation on GPUs can achieve speedups of up to an order of magnitude compared to sequential CPU implementations.

A unique approach was presented by Yu et al. [24] for NURBS surface tool path planning that leverages CPU–GPU heterogeneous parallel computing using OpenCL. The approach decomposes the NURBS surface calculation, specifically for the isoparametric line method, into matrix multiplications. The computationally intensive parts of this matrix multiplication are offloaded to the GPU, utilizing its data parallel processing capabilities, while the CPU handles logic control and less computationally demanding tasks. The authors report speedup ratios ranging from 1.5 to 15.9 compared to a traditional serial implementation, demonstrating the performance benefits of their CPU–GPU parallel approach, especially as the density of points on the toolpath increases.

While these contributions have significantly improved the computational performance of machining processes, generation of trajectories, and simulations, a perspective aligned with the principles of Industry 5.0 remains largely absent. Further research is required to develop user-centric, adaptive, and efficient solutions that fully exploit the computational resources already available in industrial environments, which typically include multicore CPUs, many of which also integrate GPUs suitable for general-purpose computing.

3. Case study: Virtual digitizing for turning lathe machining

As a case study, we present the efficient implementation of a specific tool path algorithm known as Virtual Digitizing (VD). By definition, the VD algorithm inherently avoids tool–surface collisions [25]. Since all machining processes are simulated, it imposes no constraints on tool or machine specifications. Consequently, the VD algorithm is applicable to any type of machinery for which a machining strategy is defined. These strategies may vary significantly and are often closely tied to the specific characteristics of the target machine.

In the VD method, surfaces are modeled as point clouds rather than as explicit geometric entities with topological relationships. This modeling choice simplifies tool–surface distance computations, at the expense of an approximation error that depends on point density. Increasing the sampling resolution mitigates this effect and allows the desired accuracy to be achieved.

The core operation of this method consists in computing the distance between the tool and the object to be manufactured at each step of the path generation computing. This distance, measured from a point on the object to the tool, is determined by projecting the grid point onto the tool along the tool's direction of approach. The distance between the original point and its projection is then used to compute the center position of the tool for that specific machining location. As shown in Fig. 1 (a simplified 2D example showing how a single path point is obtained for a circular tool), the minimum distance corresponds to the required offset from the tool center to reach the grid point without colliding with the surface geometry.

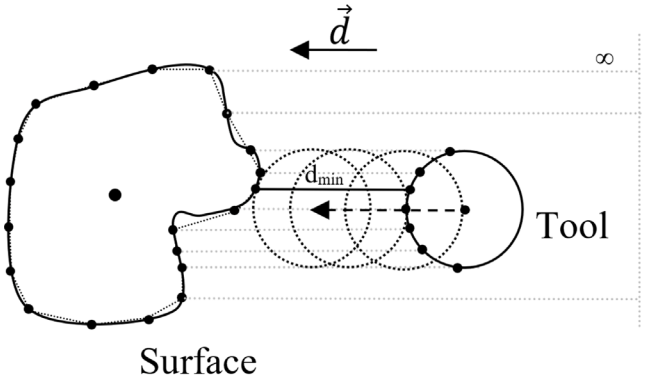


Fig. 1. Example of Virtual Digitizing (VD) process for 2D shapes.

The general VD pseudocode is presented in Algorithm 1. From a computational perspective, the object to be manufactured consists of a set of i surfaces (see line 4), each containing a set of points identified by the tuple (j, k). Therefore, for each point traversed by the tool the minimum distance is computed, as previously discussed.

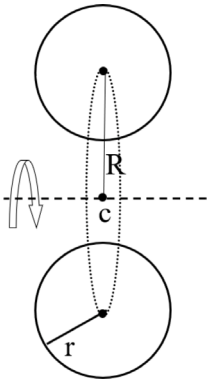
Algorithm 1 Basic Virtual Digitizing (VD) algorithm

```
1:  Trajectory ← ∅
2:  for all samplePoint ∈ samplingPath do
3:      MinDistance ← ∞
4:      for all surface_i ∈ Object do
5:          for all point_{j,k} ∈ surface_i do
6:              p'_{j,k} ← point_{j,k} · T R_{4×4}
7:              d ← d_v(p'_{j,k}, Tool)
8:              if d < MinDistance then
9:                  MinDistance ← d
10:             end if
11:         end for
12:     end for
13:     EECentre ← ComputeCentre(MinDistance, T R_{4×4})
14:     Trajectory ← Trajectory ∪ {EECentre}
15: end for
```

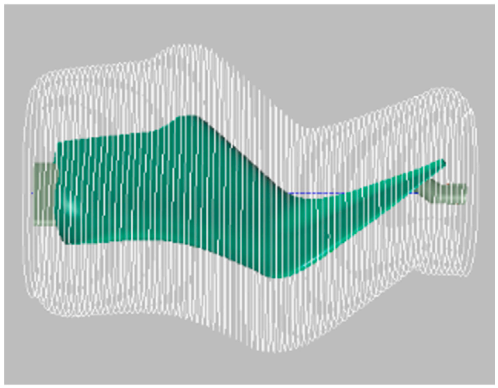
Fig. 2 illustrates the fundamental components of a machining lathe. The choice of cutting tool depends on the geometry and material of the object to be machined; in practice, the toroidal cutter, shown in Fig. 2(a), is most commonly used in this context, so named because of the torus-like shape described by its blades during rotation. It is also worth noting that the tool can be changed during the machining process to accommodate different geometrical features or material requirements. Machining lathes typically operate with three distinct axes, as shown in Fig. 2(c): a translational axis (X), a rotational axis (Z), and the tool approach axis (Y). The combined motion of these axes produces a helical toolpath around the workpiece, as illustrated in Fig. 2(b).

Algorithm 2 formalizes the VD strategy in the context of custom shoe last machining, extending the generic formulation from Algorithm 1 to meet the specific geometric and kinematic requirements of lathe-based manufacturing setups. The algorithm comprises four nested loops: the two outermost simulate the spatial motion of the virtual tool around the last, accounting for translational and rotational steps along a helical path. The inner loops traverse the discretized mesh of the last's surface to identify the point with the minimum distance to the tool geometry.

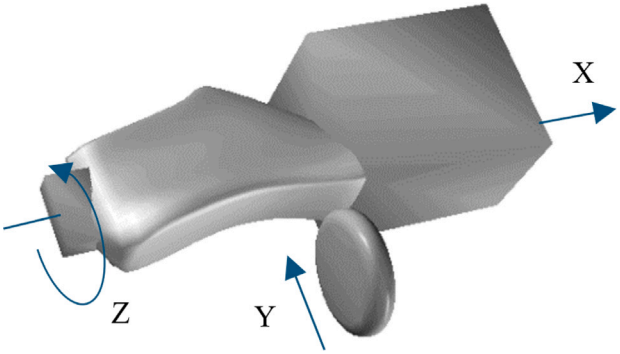
At each configuration, a homogeneous transformation matrix is applied, whose rotational component corresponds to a fundamental 3D rotation about the X-axis, representing the lathe's spindle motion. The distance metric used in the innermost evaluation step computes the



(a) Geometry of the toroidal tool



(b) Path calculated for the machining of a shoe last



(c) Simulation of the machining process

Fig. 2. Key elements involved in lathe toolpath generation.

shortest distance from each surface point to a virtual toroidal cutter, projected along the tool's approach direction, which is aligned with the Y-axis. This metric, analytically defined in Eq. (1), ensures that each tool pose along the trajectory remains optimally aligned with the object's geometry, while simultaneously avoiding potential tool–surface collisions.

Algorithm 2 Refined VD algorithm for simulating turning lathe dynamics

```
Trajectory ← ∅
for all x ∈ longitudinal_steps do
    for all θ ∈ angular_steps do
        MinDistance ← ∞
        TR_4×4 ← RotationMatrixX(x, θ)
        for all surface_i ∈ Object do
            for all point_j,k ∈ surface_i do
                p'_j,k ← point_j,k · TR_4×4
                d ← d_v(p'_j,k, Tool)
                if d < MinDistance then
                    MinDistance ← d
                end if
            end for
        end for
        EECentre ← ComputeCentre(MinDistance, TR_4×4)
        Trajectory ← Trajectory ∪ {EECentre}
    end for
end for
```

$$D(x,y,z) = T_y - y - \sqrt{\left(R + \sqrt{r^2 - (x - T_x)^2}\right)^2 - z^2} \tag{1}$$

Where :

T_x, T_y are the x, y coordinates of the torus center.

x, y, z are the 3D point coordinates.

R, r are the major and minor torus radii.

Depending on the size and resolution of the object to be manufactured, as well as the required machining precision and the density of the generated trajectory, the VD algorithm may become computationally intensive. Therefore, efficient computational strategies are required to reduce processing time. In the following section, we evaluate different strategies to accelerate the process without compromising accuracy.

4. Developed methods for accelerating the virtual digitizing

This section presents the developed methods aimed at accelerating the execution of the VD algorithm. These methods explore different levels of parallelism and hardware-aware optimizations, targeting both multicore and manycore architectures.

4.1. Multicore-based parallel algorithm for VD

On general-purpose shared-memory platforms, commonly referred to as multicore systems, near-ideal efficiencies can be achieved when two key conditions are met: first, the parallel algorithms must be designed with a coarse-grained structure; and second, the sequential portions of the workload must account for only a small fraction of the overall computational cost. However, excessively coarse-grained designs may suffer from limited parallel scalability due to inadequate workload distribution across the available computing resources.

Consequently, our parallel implementation adopts a coarse-grained strategy, with an explicit focus on certifying its parallel scalability. Additionally, careful consideration is given to the system's internal architecture, with particular attention to aspects such as low-level memory hierarchy behavior, potential contention, and required synchronization mechanisms.

As shown in Algorithm 3, the coarsest-grain parallelization strategy should distribute the workload across iterations of the outermost loop (see line 2 of Algorithm 2), which controls the linear displacement of the virtual tool along the object's main axis, thereby enabling full workspace coverage through successive translational positions. The number of iterations in the outermost loop depends on the total length of the object along the scanning axis and the chosen translational step size.

Algorithm 3 Baseline OpenMP-based parallel version of Algorithm 2

```
1: Trajectory ← ∅
2: parallel for x ∈ longitudinal_steps
3:     for all θ ∈ angular_steps do
4:         MinDistance ← ∞
5:         TR_4×4 ← RotationMatrixX(x, θ)
6:         for all surface_i ∈ Object do
7:             for all point_j,k ∈ surface_i do
8:                 p'_j,k ← point_j,k · TR_4×4
9:                 d ← d_v(p'_j,k, Tool)
10:                if d < MinDistance then
11:                    MinDistance ← d
12:                end if
13:            end for
14:        end for
15:        EECentre ← ComputeCentre(MinDistance, TR_4×4)
16:        Trajectory ← Trajectory ∪ {EECentre}
17:    end for
18: end parallel for
```

Another important aspect in the design of parallel algorithms is the definition of variables within a parallel region, considering that we work with a fork-join model provided by OpenMP [26]. To preserve the goal of getting as close as possible to ideal parallel efficiencies, all scalar variables will be defined as private variables, while dynamic data spaces will be defined as global variables. Additionally, all precomputed configurations and auxiliary computations that do not change during execution are stored in shared memory. Since these structures are read-only throughout the loop, they can be safely accessed by all threads in the parallel region, enabling efficient reuse without incurring data races or synchronization costs. As can be seen in Algorithm 4, the only shared memory structure that is written during execution is the one used to store the resulting trajectory. However, since each thread writes to a distinct, non-overlapping portion of this memory, no contention control is required.

On the other hand, the configuration of the parallel loop scheduling policy is determined empirically. Although certain strategies may introduce higher overheads due to thread management and synchronization, this can be offset by improved load balancing depending on the characteristics of the workload. In our case, the specific scheduling policy and its implications are evaluated in detail in Section 5, together with an analysis of the algorithm's scalability and its effective utilization of the parallel execution environment.

4.2. Manycore-based algorithms for VD

This section explores algorithmic optimization through fine-grained and hierarchical parallelism, with a particular focus on architectures capable of executing massive numbers of threads concurrently. In this

Algorithm 4 Shared-memory parallel version of the VD algorithm

```
1: nTh ← number of threads
2: nLonStep ← number of longitudinal_steps
3: nAngStep ← number of angular_steps
4: helicalPath_size = nLonStep × nAngStep
5: Trajectory ← allocate(helicalPath_size)
6: ParFor m ∈ {1, ..., nLonStep}                    ▷ (nTh threads)
7:     for n ∈ {1, ..., nAngStep} do
8:         priv_T ← Tool(m)
9:         priv_x ← x(m)
10:        priv_θ ← degree(n)
11:        MinDistance ← ∞
12:        TR_4×4 ← RotationMatrixX(priv_x, priv_θ)
13:        for all surface_i ∈ Object do
14:            for all point_j,k ∈ surface_i do
15:                p'_j,k ← point_j,k · TR_4×4
16:                d ← d_v(p'_j,k, priv_T)
17:                if d < MinDistance then
18:                    MinDistance ← d
19:                end if
20:            end for
21:        end for
22:        EECentre ← ComputeCentre(MinDistance, TR_4×4)
23:        Trajectory[m × nAngStep + n] ← EECentre
24:    end for
25: EndParFor
```

context, we exploit GPU architectures for general-purpose computing (GPGPU), using the CUDA programming model to harness the large number of processing cores typically available in modern graphics processors.

The motivation for this approach stems from the computational nature of the VD algorithm, which presents multiple independent operations over spatially distributed data. These properties suggest that the algorithm may benefit from a data-parallel execution model, where thousands of lightweight threads can process distinct subsets of the input in parallel, thereby reducing the average machining response time.

The degree of parallelization granularity significantly influences both the implementation strategy and the computational challenges. In our algorithm, when a finer granularity is used, more threads can be launched in parallel, but this increases the need for synchronization and careful coordination, which can introduce overhead. Conversely, with a coarser granularity, synchronization is reduced, but the degree of parallel occupancy is also more limited. Both alternatives have been implemented and will be analyzed in detail to evaluate their respective trade-offs on GPU architectures.

4.2.1. Coarse-grained asynchronous algorithm

As shown in Algorithm 5, each trajectory point in the VD algorithm can be computed independently, provided that each point is entirely processed by a single thread. This allows for parallel execution without inter-core synchronization, as no point requires cooperation between multiple processing threads. We exploit this by assigning one CUDA thread to each trajectory point computation. We call this algorithm asynchronous because no inter-thread synchronization is needed during its execution. Therefore, the execution of all threads is independent, which helps prevent a major source of inefficiency in massively parallel systems: the overhead introduced by thread synchronization. However, a potential drawback of this strategy is the relatively high computational load assigned to each core. On GPU architectures, where each processing element is lightweight a coarse-grained workload may lead to suboptimal resource utilization, particularly if the available parallelism is insufficient to fully occupy the hardware.

As previously mentioned, the coarse-grained algorithm presented in Algorithm 5 avoids the need for CUDA thread synchronization. Therefore, a grid is launched with at least as many threads as the number of points to be computed along the helical path (`helicalPath_size` in Algorithm 5). These points correspond to the two outermost loops in Algorithm 2 (see lines 2 and 3). In Algorithm 5, the CUDA-provided variables `blockIdx.x`, `blockDim.x` and `threadIdx.x` are used to compute the global thread identifier (`threadId`) associated with each point of the trajectory solution. The kernel is launched using a one-dimensional configuration for both the grid and the blocks, meaning that both the number of blocks and the number of threads per block are specified along a single dimension.

Algorithm 5 Manycore-based coarse-grained asynchronous VD algorithm

1: `dTrajectory` ← `cudaMalloc(helicalPath_size` × `sizeof(double))`
2: Initialize GPU with input values
3: **In parallel:** A CUDA thread for each `helixPoint` ∈ `helicalPath`
4: `threadId = blockIdx.x × blockDim.x + threadIdx.x;`
5: **if** `threadId < helicalPath_size` **then**
6: $x \leftarrow \lfloor$`threadId/nAngStep`$\rfloor$
7: $\theta \leftarrow$ `blockIdx.x` mod `nAngStep`
8: MinDistance ← ∞
9: $TR_{4\times4} \leftarrow$ Rotation_MatrixX(x, θ)
10: **for** each surface $i \in$ Object **do**
11: **for** each point $j, k \in$ surface$_i$ **do**
12: $p'_{j,k} \leftarrow$ point$_{j,k} \cdot TR_{4\times4}$
13: $d \leftarrow d_v(p'_{j,k}$, Tool$)$
14: **if** $d <$ MinDistance **then**
15: MinDistance ← d
16: **end if**
17: **end for**
18: **end for**
19: EECentre ← ComputeCentre(MinDistance, $TR_{4\times4}$)
20: Trajectory[`threadId`] ← EECentre
21: **end if**
22: **End parallel**
23: `hTrajectory` ← `dTrajectory`

4.2.2. Fine-grained synchronous algorithm

Beyond the parallelism inherent in the independent evaluation of each point, as discussed in the previous proposal, an even finer-grained level of parallelism can be exploited: the computation of the distance between the tool and the surface at a given trajectory point, as shown in Algorithm 6. The computational workload to be distributed is defined between lines 6 and 14 of Algorithm 2, where two inner nested loops are used to determine the minimum distance between the surface and the cutting blade. Specifically, the distance from each surface point to the blade is computed independently, and the minimum among these values is subsequently selected. Since the distance computation for each surface point is entirely independent, this step can be parallelized efficiently; however, the reduction phase required to determine the global minimum introduces a synchronization requirement.

Algorithm 6 presents the designed fine-grained synchronous algorithm. As previously noted, this version entirely avoids the use of iterative constructs, enabling full parallelization of the solution computation. The kernel is launched using a standard CUDA grid configuration consisting of multiple blocks, each containing a set of threads. Each block corresponds to a point along the helicoidal trajectory and is identified by the internal variable *blockIdx.x* (see lines 6 and 7), while the threads within each block are assigned to evaluate this point. For simplicity, this implementation considers only a single surface. Since VD operates directly on point clouds without relying on surface topology, multiple surfaces can be treated as a single effective surface by merging their corresponding point sets without altering the correctness of the distance computation. Specifically, each surface point (j, k) is identified by the thread indices `threadIdx.y` and `threadIdx.z`, respectively.

Algorithm 6 Manycore-based fine-grained synchronous VD algorithm

1: `dTrajectory` ← `cudaMalloc(helicalPath_size` × `sizeof(double))`
2: Initialize GPU with input values
3: *dim3* *blockDim*(1, `surface_size_y, surface_size_z)`
4: *dim3* *gridDim*(`helicalPath_size`)
5: **In parallel:** CUDA kernel size: <<< *gridDim*, *blockDim* >>>
6: $x \leftarrow$ `blockIdx.x` × `nAngStep`
7: $\theta \leftarrow$ `blockIdx.x` mod `nAngStep`
8: $i \leftarrow 0$ ▷ surface index (single-surface simplification)
9: $j \leftarrow$ `threadIdx.y`, $k \leftarrow$ `threadIdx.z`
10: $TR_{4\times4} \leftarrow$ Rotation_MatrixX(x, θ)
11: $p'_{i,j,k} \leftarrow$ point$_{j,k} \cdot TR_{4\times4}$
12: $d \leftarrow d_v(p'_{i,j,k}$, Tool$)$
13: `SynchronizeThreads()`
14: MinDistance ← ComputeReduction(d)
15: EECentre ← ComputeCentre(MinDistance, $TR_{4\times4}$)
16: Trajectory[`blockIdx.x`] ← EECentre
17: **End parallel**
18: `hTrajectory` ← `dTrajectory`

This processing approach exhibits a higher degree of parallelism than the coarse-grained alternative. Therefore, the amount of code executed by each CUDA thread is smaller than in the coarse-grained version, which can positively impact performance on massively parallel architectures. However, the need to synchronize the threads within the same block (line 13) to compute the minimum distance, as well as the reduction operation itself, introduces an overhead that may degrade the overall performance of the algorithm.

To compute the minimum distance within each CUDA block (see line 14), the classical reduction scheme proposed by Mark Harris in his NVIDIA technical presentation on parallel reduction [27] is used. Following this scheme, each thread computes the distance between a transformed surface point and the tool, and stores the result in a shared memory array. A parallel in-block reduction is then performed by iteratively comparing and updating minimum values between pairs of threads, halving the number of active participants at each step. Thread synchronization ensures correct access to shared memory during the process. It is important to note that this method requires the number of threads per block to be a power of two; having more threads than surface points does not negatively affect the efficiency of the algorithm. On the other hand, we consider surfaces of no more than $1,024$ points, which is sufficient since it represents a slice of the treated shoe last and, in practice, does not represent a limiting factor.

To compute the minimum distance within each CUDA block (see line 14), the classical reduction scheme proposed by Mark Harris in his NVIDIA technical presentation on parallel reduction [27] is used. Following this scheme, each thread computes the distance between a transformed surface point and the tool, and stores the result in a shared memory array. A parallel in-block reduction is then performed by iteratively comparing and updating minimum values between pairs of threads, halving the number of active participants at each step. Thread synchronization ensures correct access to shared memory during the process. It is important to note that this method requires the number of threads per block to be a power of two; having more threads than points in the processed slice does not negatively affect the efficiency of the algorithm. In this work, each helicoidal step operates on a local surface slice composed of no more than $1,024$ points, which is sufficient

for the treated shoe lasts and, in practice, does not constitute a limiting factor. Consequently, this fine-grained strategy assumes that, at each helicoidal step, the relevant local surface slice can be represented by a bounded number of points. For objects exhibiting extremely high local geometric complexity, this assumption may no longer hold, as the number of points required to accurately represent a slice could exceed the maximum number of threads per CUDA block.

5. Numerical experiments

This section presents an analysis of the computational performance of the developed algorithms. It begins by describing the experimental conditions under which the evaluations were carried out. Subsequently, the performance of the parallel algorithms designed for shared memory architectures is examined, followed by an assessment of those optimized for manycore platforms. The section concludes with a comparative analysis aimed at identifying the relative advantages of each approach.

5.1. Experimental setup

Several approaches have been proposed to accelerate the VD algorithm used in shoe last machining. These algorithms are evaluated using real 3D models from the footwear sector, which are employed to compute machining trajectories for shoe lasts on lathe-based manufacturing systems. The 3D models correspond to real digitized lasts provided by the Spanish Technological Institute for Footwear Research (INESCOP[1]).

The machining of shoe lasts was traditionally performed using copying lathes. In these machines, a physical master model was used, along which a toroidal feeler, identical in shape and dimensions to the cutting tool, was guided. The feeler followed a helical path over the surface of the original model, and its motion was mechanically transmitted to the cutting axes, which reproduced the shape by removing material from a plastic block. In modern machining systems, these mechanical feelers are no longer used. Instead, the toolpaths are generated directly by CAM software, which provides the precise positions of the three machining axes to shape the last. Fig. 3 illustrates the shoe last machining process, showing the initial and final shapes of the model.

The accuracy requirements in the manufacture of shoe lasts are stringent, with a maximum allowable error of ± 0.1 mm. This level of precision necessitates the use of high-definition 3D models, featuring surfaces represented by dense point meshes. Additionally, toolpaths must be computed with high accuracy, describing helicoidal trajectories with great reliability. To ensure rigor, the study includes real cases involving both last geometries and trajectory definitions at varying levels of resolution.

Table 1 presents the shoe lasts used in the experiments, along with their key dimensions, length and ball perimeter, and their corresponding mesh resolution, expressed as the total number of surface points. The table includes two categories of women's lasts: on the one hand, models digitized from original physical prototypes, shown with a wood texture (test, 0511pos and blu22b); and on the other, models obtained by digitizing previously machined lasts, displayed in blue tones (P39f, abba and Papera). The latter group exhibits the highest resolution, with some models reaching nearly 300,000 vertices, or grid points from a computational perspective.

In practice, a typical shoe last surface can be adequately described with approximately 0.17 points/mm^2. In the experimental dataset, surface densities range up to 0.72 points/mm^2, which can be considered as representing the highest level of geometric fidelity.

With regard to the definition of the toolpath for the lathe, Table 2 summarizes the characteristics of the shoe lasts used in the experiments. The toric tool follows standard specifications for shoe last manufacturing, with a minor radius of 8 mm and a major radius of 27 mm.

As for the helicoidal path, its two main parameters are specified: the helicoid pitch, the axial distance between two turns, and the rotation pitch, the angular increment (in degrees) between consecutive rotations around the axis. The total number of trajectory points to be generated depends on the model's length, which determines the number of turns and, consequently, the rotation pitch, as expressed in Eq. (2).

$$T_p = \frac{L}{H_s} \cdot \frac{360}{R_s} \tag{2}$$

where:

T_p is the number of points of the trajectory

L is the length of the model (in mm)

H_s is the pitch of the helicoid (in mm)

R_s is the rotation pitch (in sexagesimal degrees)

In terms of trajectory resolution, satisfactory results can be achieved starting from 0.2 points/mm. In the experiments, this density is increased up to 1.1 points/mm to encompass a broad range of accuracy levels.

The three helical configurations presented in Table 2 correspond to different levels of trajectory resolution, each aligned with typical industrial applications. The High Resolution (HR) configuration, characterized by small helical and angular steps, is representative of production environments, where precise toolpaths are critical to ensure dimensional accuracy and surface quality in the final product. The Medium Resolution (MR) setting reflects the needs of the design and development phase, where sufficient geometric fidelity is required to validate and refine last prototypes, while optimizing computational resources. Lastly, the Low Resolution (LR) configuration is suitable for commercial or visualization purposes, where fast processing and simplified geometry suffice for cataloging, virtual fitting, or client communication, without requiring high geometric accuracy. These levels of resolution thus reflect a practical balance between computational cost and the fidelity required for each stage of the product lifecycle.

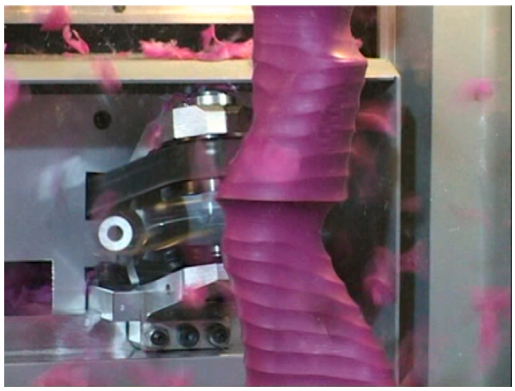
Two computational platforms of differing performance levels were used in this study. The first platform is a high-performance compute node intended exclusively for computing tasks. It is equipped with two Intel Xeon Gold 6230 CPUs (20 cores each at 2.10 GHz), 256 GB of RAM, and an NVIDIA TITAN RTX GPU, running CentOS Stream 8. The Intel Xeon Gold 6230 supports Hyper-Threading, Intel's implementation of Simultaneous Multithreading (SMT), allowing its 20 physical cores to handle 40 concurrent threads, therefore, with two processors, the system can run up to 80 threads simultaneously for enhanced parallel performance. Compilation was performed using GCC version 8.5.0 with OpenMP 4.5 enabled and CUDA 12.6.

The second platform corresponds to a standard user-level workstation, typically used for everyday tasks. It is equipped with an Intel Core i7-4790 processor, 8 GB of RAM, and an NVIDIA GeForce GTX 970 GPU, running the Windows 11 operating system, the CUDA version was 12.6.

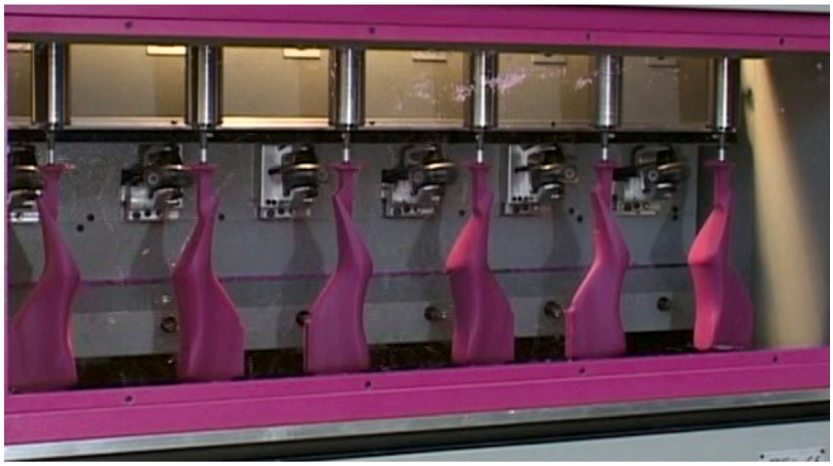
In order to cover a representative spectrum of computational capabilities, ranging from hardware commonly available in small and medium-sized enterprises in industrial sectors such as footwear manufacturing to high-performance and state-of-the-art GPU architectures, a third GPU, the NVIDIA RTX 4090, was incorporated into the experimental evaluation. This choice supports the validity of the performance analysis across both industrial and advanced computing scenarios.

[1] https://inescop.es/en/

(a) Injected block of rough machining in a turning lathe machine



(b) First step of rough machining in a turning lathe machine



(c) Finished shoe lasts

Fig. 3. Stages of the shoe last machining process.

5.2. Multicore parallel algorithm performance

We will computationally analyze the parallelization without using any additional performance-enhancing technique on a high-perf ormance shared-memory platform.

Table 3 presents the results for all the shoe lasts described in Table 1, and for the three configurations detailed in Table 2. Additionally, results are shown using up to 80 threads, i.e., when SMT is enabled on the two Intel Xeon Gold 6230 processors, allowing each of their 20 physical cores per processor to run two threads concurrently. In this table, we can clearly observe that increasing the number of grid points defining the shoe last, which corresponds to a higher number of vertices, as well as increasing the resolution of the trajectory, results in higher computational (sequential) times. Similarly, the parallel developed algorithm consistently reduces computational time cost as the number of threads increases, except in a few low-cost experiments, where using the maximum number of threads (twice the number of physical cores available) does not lead to further improvement.

Moreover, Table 4 presents the parallel efficiencies corresponding to the results shown in Table 3. It can be observed that the efficiencies are relatively good in experiments with the highest computational cost, but they are not satisfactory across all cases. In particular, when using as many threads as there are physical cores (i.e., 40), the average efficiency is only 40%. This suggests that the first aspect to analyze is the potential load imbalance, which could be mitigated by using the *dynamic* scheduling policy in the for loop construct.

A degradation in multicore efficiency is observed when the number of software threads exceeds the number of available physical cores, i.e., 40. This behavior is primarily associated with the use of Simultaneous Multithreading (SMT), where multiple logical threads share the same physical core execution resources. While SMT can improve performance by hiding memory and pipeline latencies, it often leads to resource contention in compute-intensive and memory-bound workloads, such as the one evaluated in this study.

On the other hand, synchronization overhead and operating system scheduling costs always contribute to this effect, particularly when logical cores are used. This impact becomes more pronounced given the relatively short execution times reported in Table 3, where fixed overheads can represent a larger fraction of the total runtime, even under static scheduling, which introduces the lowest runtime overhead.

Although the system cannot be considered scalable in the strict sense once logical cores are used, increasing the level of parallelism still leads to a consistent reduction in execution time. This behavior enables improved user-perceived responsiveness and compliance with strict timing requirements in latency-sensitive applications.

Therefore, Table 5 presents the results obtained using the *dynamic* scheduling policy with a chunk size of 1. Although this configuration entails the highest computational overhead, it is the most effective at mitigating potential load imbalances. The table shows a clear improvement in computational efficiency, increasing from 40% in the previous experiment to 65% in the current one when using as many threads as there are physical cores (i.e., 40). Efficiency exceeds 80% for the most

Table 1
Shoe lasts used for the experiments.

Image	Name	Length (mm)	Ball (mm)	Grid points
	test	245	202	3,720
	0511pos	278	213	24,930
	blu22b	283	222	55,080
	P39f	286	209	77,580
	abba	244	215	151,200
	Papera	258	210	283,680

computationally demanding experiments, in which it is necessary to use all available resources.

Table 6 reports the execution times corresponding to the efficiencies shown in Table 5. When dynamic scheduling is employed, absolute execution times are consistently lower than those obtained with static scheduling (Table 3), despite exhibiting poorer scalability in terms of execution time reduction. This behavior indicates that the dynamic scheduling strategy effectively mitigates minor load imbalances present in the workload. However, when Simultaneous Multithreading (SMT) is enabled, execution times no longer decrease and may even slightly degrade. In this regime, the additional overhead introduced by dynamic scheduling, together with increased contention and synchronization costs, outweighs any potential benefit from using logical cores. Consequently, the use of SMT does not provide practical advantages under dynamic scheduling and is therefore not justified for this configuration. Accordingly, SMT is disabled in the remaining experiments.

Finally, the optimal chunk size is analyzed in Table 7. This table presents data for the blu22b shoe last, since the conclusions can be

Table 2
Different helical configurations for the experiments.

Name	Helical step (mm)	Rotation step (sexagesimal degrees)
Low Resolution (LR)	3.0	6.0
Medium Resolution (MR)	1.2	2.4
High Resolution (HR)	0.6	1.7

extrapolated to the rest of the shoe lasts. It shows that the best efficiencies are obtained when the chunk size is set to 1. Although, as previously discussed, this configuration introduces the highest computational overhead, the experimental results demonstrate that it is the most effective in mitigating load imbalance.

5.3. Manycore parallel algorithm performance

In the case of the algorithms developed for manycore architectures, they are evaluated on two different platforms, as described in Section 5.1. These platforms differ, among other aspects, in the GPU devices employed, namely the NVIDIA GTX 970 on the workstation platform and the NVIDIA TITAN RTX and NVIDIA RTX 4090 on the high-performance compute node, which results in markedly different computational capabilities.

Compared to the NVIDIA GTX 970, both the TITAN RTX and the RTX 4090 provide substantially more powerful architectures for parallel computing. While the GTX 970 includes 13 Streaming Multiprocessors (SMs) with a total of 1664 CUDA cores, the TITAN RTX features 72 SMs and 4608 CUDA cores, and the RTX 4090 further extends this parallelism with 128 SMs and 16,384 CUDA cores, reflecting its more recent Ada Lovelace architecture. In addition, architectural improvements over Maxwell enable higher floating-point throughput per core on both high-end GPUs. Regarding memory resources, the GTX 970 provides 48 KB of shared memory per SM and 4 GB of global memory, whereas both the TITAN RTX and the RTX4090 offer 64 KB of shared memory per SM and 24 GB of global memory. In all cases, these memory capacities comfortably satisfy the requirements of the proposed algorithms.

The two algorithms described in Section 4.2 will be analyzed on both architectures, using the sequential computational times on their respective CPUs as reference times.

5.3.1. Coarse-grained asynchronous version performance

Table 8 shows the computational time of the coarse-grained algorithm for all shoe lasts and for the three trajectory resolutions analyzed, both for sequential execution on the CPU and execution on the GPU, which in this case is the GTX 970. It can be observed that the proposed algorithm is able to reduce execution times in all cases. Even for the shoe lasts with the lowest number of vertices and using the lowest trajectory resolution, a reduction greater than 50% is achieved.

As with the parallel algorithm for multicores, an increasing performance is observed as a function of the complexity of the trajectory calculation; that is, the higher the number of vertices on the surface, the greater the acceleration, and the greater the number of trajectory points to be obtained. This trend, shown in Fig. 4, is systematically observed in all scenarios evaluated.

This figure shows maximum speed-ups of up to $70x$, which would be unattainable on a CPU such as the i7-4790, which features 4 physical cores and 8 logical threads; therefore, in the ideal case, its maximum achievable speed-up would be only $8x$.

The second architecture, which is oriented towards high-performance processing, includes both more powerful processors and a more advanced GPU, the TITAN RTX. Table 9 shows the computational times for both the sequential (CPU) version and the coarse-grained GPU version, as well as the speed-up achieved by the latter on this platform. It can be observed that certain experiments exhibit a speed-up below 1, meaning that GPU execution is not advisable in those cases. However, this only occurs when the computational times are extremely short.

Furthermore, although the speed-up is lower than that shown in Fig. 4, this does not imply worse performance. On the contrary, when comparing execution times, both the CPU and the GPU show improvements, as expected, compared to the results on the user-level platform. In fact, for the most computationally demanding experiment, the CPU time is reduced to one third, and the GPU time to less than half.

Table 10 reports the Nsight Compute analysis of the coarse-grained kernel on the TITAN RTX GPU. Across all models and resolutions, the kernel consistently achieves very high occupancy levels (around 97%), indicating that the GPU resources are effectively populated and that thread-level parallelism is not a limiting factor. Compute throughput remains high for low and medium resolutions, typically exceeding 70%, and gradually decreases for higher resolutions as the computational workload per thread increases. In contrast, memory throughput remains low (below 5%) for all configurations.

Table 11 reports the Nsight Compute analysis of the coarse-grained kernel on the RTX 4090. Across all models and resolutions, the kernel shows a stable and high level of hardware utilization, with Achieved Occupancy values consistently around 60%. Compute throughput remains high for all configurations, typically exceeding 70% and reaching values close to 80% for low and medium resolutions, with a moderate decrease observed at higher resolutions as the computational workload per thread increases. In contrast, memory throughput remains very low for all models and resolutions.

5.3.2. Fine-grained synchronous version performance

In this section, we analyze the behavior of the proposed fine grained algorithm. As explained in Section 4.2.2, the set of cores within a block cooperate to compute the minimum distance by performing a reduction process. This process involves synchronizations between threads and also implies that not all threads are active at all times, as described in [27].

Fig. 5 shows the speed-ups of the fine-grained algorithm on the GTX 970 GPU. It can be observed that the speed-up remains stable for all the lasts with a lower number of vertices and different toolpath resolutions, while it increases for those with a higher number of vertices. Additionally, for this shoe-last, the speed-up varies depending on the toolpath resolution.

Table 12 presents the computational cost in seconds and the corresponding speed-up values obtained on the high-performance platform. The results demonstrate excellent performance across all configurations. Notably, in none of the experiments does the execution time on both the TITAN RTX and the RTX 4090 GPUs exceed one second. In particular, the TITAN RTX reaches speed-up values of up to 80×, whereas the RTX 4090 significantly outperforms it, exceeding 270× in the most demanding high-resolution configurations).

Table 13 presents the Nsight Compute analysis of the manycore fine-grained kernel on the TITAN RTX GPU. The results show consistently high compute throughput across most models and resolutions, typically around 80% for small and medium workloads and slightly lower values for the largest models. Achieved occupancy increases with problem resolution for most cases, reaching values above 90% for medium and high resolutions, although more complex geometries exhibit lower occupancy levels. In contrast, memory throughput remains relatively high and stable across all configurations, close to 46%.

Table 14 presents the Nsight Compute analysis of the fine-grained kernel on the RTX GPU. The results show consistently high compute throughput across all models and resolutions, typically between 80% and 88%. Achieved occupancy generally increases with problem resolution for most cases, reaching values above 90% for medium and high resolutions in several models, while more complex geometries exhibit lower occupancy levels due to increased resource usage. Memory throughput remains high and stable across all configurations, with values ranging between approximately 46% and 72%.

Table 3
Multicore parallel algorithm: execution times (s) for each shoe last and number of threads. Schedule *static*.

N. threads	test			0511pos			blu22b		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
1	0.112	0.692	1.942	0.204	1.150	3.210	0.605	3.798	10.192
2	0.061	0.370	1.025	0.105	0.625	1.741	0.320	1.958	5.391
5	0.027	0.157	0.436	0.048	0.271	0.745	0.139	0.824	2.283
10	0.015	0.081	0.225	0.037	0.165	0.396	0.076	0.473	1.265
15	0.017	0.075	0.206	0.028	0.124	0.339	0.074	0.282	1.009
20	0.010	0.045	0.124	0.023	0.126	0.309	0.047	0.366	0.738
30	0.009	0.055	0.145	0.014	0.091	0.242	0.042	0.251	0.705
40	0.009	0.044	0.124	0.020	0.095	0.271	0.041	0.250	0.616
50	0.009	0.040	0.104	0.013	0.079	0.223	0.039	0.164	0.573
60	0.009	0.044	0.109	0.013	0.067	0.191	0.028	0.181	0.515
70	0.010	0.037	0.096	0.014	0.067	0.186	0.027	0.154	0.448
80	0.018	0.037	0.082	0.015	0.061	0.158	0.030	0.149	0.416

N. threads	P39f			abba			Papera		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
1	1.134	7.161	19.729	1.968	12.351	36.462	3.421	21.705	63.097
2	0.583	3.641	10.386	1.069	6.285	18.563	1.692	10.706	29.639
5	0.281	1.646	4.470	0.481	2.860	7.770	0.823	4.603	13.045
10	0.158	0.971	2.351	0.391	1.525	4.506	0.507	2.738	6.612
15	0.144	0.587	1.659	0.259	1.107	2.989	0.370	2.354	4.893
20	0.098	0.723	1.610	0.193	0.859	2.385	0.326	2.120	4.530
30	0.078	0.506	0.926	0.147	0.688	1.833	0.234	1.459	4.211
40	0.055	0.425	1.197	0.172	0.602	1.693	0.228	1.098	3.305
50	0.069	0.406	1.062	0.098	0.637	1.881	0.166	1.110	3.207
60	0.049	0.340	0.943	0.099	0.586	1.490	0.147	0.942	2.729
70	0.049	0.271	0.828	0.080	0.530	1.438	0.168	0.989	2.654
80	0.044	0.274	0.800	0.084	0.515	1.327	0.139	0.726	2.313

Table 4
Multicore parallel algorithm: parallel efficiencies for each shoe last and number of threads. Schedule *static*.

N. threads	test			0511pos			blu22b		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
2	92%	94%	95%	97%	92%	92%	95%	97%	95%
5	83%	88%	89%	85%	85%	86%	87%	92%	89%
10	75%	86%	86%	55%	70%	81%	79%	80%	81%
15	44%	62%	63%	49%	62%	63%	54%	90%	67%
20	55%	77%	78%	45%	46%	52%	65%	52%	69%
30	40%	42%	45%	47%	42%	44%	48%	50%	48%
40	31%	39%	39%	25%	30%	30%	37%	38%	41%
50	25%	35%	37%	31%	29%	29%	31%	46%	36%
60	20%	26%	30%	26%	29%	28%	36%	35%	33%
70	17%	26%	29%	21%	25%	25%	32%	35%	33%
80	8%	24%	30%	17%	23%	25%	25%	32%	31%

N. threads	P39f			abba			Papera		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
2	97%	98%	95%	92%	98%	98%	100%	100%	100%
5	81%	87%	88%	82%	86%	94%	83%	94%	97%
10	72%	74%	84%	50%	81%	81%	68%	79%	95%
15	53%	81%	79%	51%	74%	81%	62%	61%	86%
20	58%	50%	61%	51%	72%	76%	53%	51%	70%
30	48%	47%	71%	45%	60%	66%	49%	50%	50%
40	52%	42%	41%	29%	51%	54%	38%	49%	48%
50	33%	35%	37%	40%	39%	39%	41%	39%	39%
60	38%	35%	35%	33%	35%	41%	39%	38%	39%
70	33%	38%	34%	35%	33%	36%	29%	31%	34%
80	32%	33%	31%	29%	30%	34%	31%	37%	34%

6. Discussion

The Industry 5.0 paradigm encompasses multiple dimensions, with agile interaction in product design and manufacturing processes being one of the most prominent. In the footwear sector, this approach aims to enable the production of highly customized shoes, tailored to both foot geometry and individual aesthetic preferences. While toolpath computation is not typically an issue in traditional manufacturing environments, it becomes a bottleneck in this new scenario, as the generation time cannot overlap with production time due to the absence of large batch sizes. By implementing and optimizing parallel algorithms on both multicore and manycore platforms, we achieve execution times compatible with near real-time constraints, even for high-resolution 3D models.

In this research area, parallel algorithms for both CPUs and GPUs have been developed and analyzed. The CPU algorithm has been

Table 5
Multicore parallel algorithm: parallel efficiencies (%) for each shoe last and number of threads. Schedule *dynamic* and chunk = 1.

N. threads	test			0511pos			blu22b		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
2	95%	95%	99%	94%	95%	100%	100%	100%	99%
5	92%	95%	90%	89%	93%	100%	99%	100%	98%
10	81%	86%	86%	85%	90%	93%	93%	96%	96%
15	75%	83%	84%	79%	83%	94%	93%	95%	93%
20	64%	75%	78%	71%	76%	91%	80%	91%	89%
30	53%	65%	66%	61%	66%	78%	69%	79%	81%
40	32%	56%	61%	36%	62%	68%	52%	68%	74%
50	28%	48%	52%	28%	50%	55%	43%	55%	60%
60	23%	39%	41%	24%	42%	48%	36%	47%	51%
70	17%	33%	37%	21%	37%	41%	31%	40%	44%
80	8%	29%	33%	10%	29%	36%	24%	35%	39%

N. threads	P39f			abba			Papera		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
2	96%	96%	100%	100%	94%	100%	100%	100%	100%
5	93%	96%	100%	100%	94%	100%	100%	100%	100%
10	90%	92%	100%	97%	92%	99%	100%	100%	100%
15	89%	91%	98%	89%	91%	97%	96%	100%	100%
20	79%	88%	95%	90%	86%	95%	96%	100%	100%
30	70%	79%	87%	74%	78%	87%	77%	90%	89%
40	54%	73%	81%	68%	71%	80%	70%	85%	82%
50	43%	54%	64%	44%	53%	61%	50%	65%	63%
60	36%	45%	54%	38%	45%	52%	47%	55%	54%
70	30%	39%	47%	32%	39%	45%	32%	48%	47%
80	23%	35%	41%	28%	37%	40%	27%	42%	40%

Table 6
Multicore parallel algorithm: execution times (s) for each shoe last and number of threads. Schedule *dynamic* and chunk = 1.

N. threads	test			0511pos			blu22b		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
1	0.112	0.692	1.917	0.186	1.151	3.446	0.617	3.798	10.419
2	0.059	0.364	0.967	0.098	0.608	1.690	0.295	1.837	5.273
5	0.024	0.145	0.424	0.042	0.247	0.682	0.125	0.756	2.134
10	0.014	0.080	0.224	0.022	0.128	0.371	0.066	0.396	1.087
15	0.010	0.056	0.152	0.016	0.092	0.244	0.044	0.268	0.744
20	0.009	0.046	0.122	0.013	0.076	0.189	0.039	0.209	0.584
30	0.007	0.036	0.097	0.010	0.058	0.146	0.030	0.160	0.429
40	0.009	0.031	0.078	0.013	0.046	0.127	0.030	0.139	0.350
50	0.008	0.029	0.074	0.013	0.046	0.124	0.029	0.139	0.346
60	0.008	0.030	0.077	0.013	0.046	0.119	0.028	0.136	0.340
70	0.009	0.030	0.074	0.013	0.044	0.120	0.028	0.135	0.335
80	0.017	0.030	0.072	0.023	0.049	0.119	0.032	0.135	0.338

N. threads	P39f			abba			Papera		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
1	1.072	6.633	19.960	2.118	12.291	36.840	3.627	22.656	63.385
2	0.561	3.442	9.606	1.028	6.534	17.912	1.625	10.314	28.323
5	0.231	1.381	3.936	0.424	2.607	7.223	0.676	4.143	11.557
10	0.120	0.724	1.994	0.218	1.334	3.710	0.354	2.133	5.929
15	0.080	0.488	1.359	0.159	0.898	2.529	0.251	1.464	4.056
20	0.068	0.377	1.054	0.117	0.711	1.946	0.189	1.124	3.165
30	0.051	0.279	0.764	0.095	0.525	1.416	0.157	0.837	2.363
40	0.050	0.227	0.618	0.078	0.431	1.146	0.130	0.665	1.944
50	0.050	0.245	0.628	0.096	0.466	1.198	0.146	0.700	1.998
60	0.050	0.244	0.616	0.094	0.455	1.175	0.129	0.690	1.940
70	0.050	0.242	0.608	0.094	0.453	1.170	0.160	0.681	1.916
80	0.058	0.240	0.605	0.094	0.416	1.157	0.166	0.679	1.964

designed for execution on high-performance computing (HPC) platforms, particularly targeting multicore systems. The obtained speed-ups demonstrate high throughput and efficient utilization of resources, particularly when leveraging the physical cores of the computing platform. Experimental results also reveal excellent scalability, particularly under the *dynamic* load balancing policy. This feature enables the proposed solution to be applicable not only to HPC computational systems, but also to user level computing platforms, such as the platform used in this study, where the availability of idle cores varies dynamically and cannot be predetermined. This challenge is addressed by the effectiveness of the *dynamic* policy, which adjusts the computational workload by assigning tasks to threads mapped to idle or underutilized cores. Additionally, this workload assignment adapts to the evolving load of the cores, even though the load distribution across cores is

Table 7
Multicore parallel algorithm: parallel efficiencies (%) using the *dynamic* schedule with different chunk sizes for blu22b shoe last.

N. threads	chunk = 1			chunk = 2			chunk = 3		
	LR	MR	HR	LR	MR	HR	LR	MR	HR
2	100%	100%	99%	97%	100%	100%	99%	96%	100%
5	99%	100%	98%	92%	98%	98%	92%	94%	98%
10	93%	96%	96%	85%	94%	95%	82%	88%	96%
15	93%	95%	93%	80%	90%	94%	71%	88%	92%
20	80%	91%	89%	76%	85%	90%	75%	78%	88%
30	69%	79%	81%	67%	73%	79%	49%	75%	80%
40	52%	68%	74%	51%	66%	73%	35%	49%	70%

Table 8
Manycore coarse-grained algorithm: execution times (s) on the user-level platform.

Model	CPU: i7-4790			GPU: GTX 970		
	LR	MR	HR	LR	MR	HR
test	0.302	1.688	4.851	0.146	0.176	0.262
0511pos	0.460	2.884	7.926	0.229	0.242	0.352
blu22b	1.111	6.963	18.959	0.258	0.357	0.700
P39f	1.592	9.801	27.360	0.335	0.456	0.948
abba	3.124	19.940	52.162	0.546	0.770	1.629
Papera	15.636	72.130	181.810	0.948	1.342	2.628

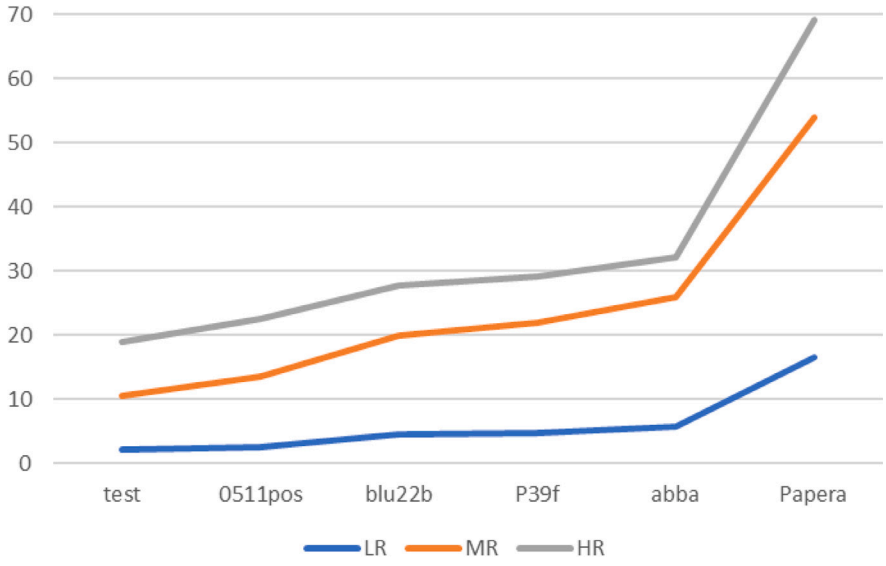


Fig. 4. Speed-up of the coarse-grained manycore algorithm on the GTX 970.

unpredictable. This explains why using a chunk size equal to 1 is the most reasonable choice for achieving optimal performance.

In scenarios where systems lack a CUDA-compatible GPU, or where such a GPU presents limitations that hinder its effective use for general-purpose computation, an alternative has been proposed in the form of a parallel algorithm optimized for multicore systems. This approach increases the portability of the solution and significantly enhances the user experience by enabling reduced response times, local execution of processes, and decreased dependence on specialized hardware.

For scenarios where CUDA-capable GPUs are available and can be effectively exploited, parallel algorithms with both fine-grained and coarse-grained designs have also been developed using CUDA and evaluated on GPUs with varying architectures and computational capabilities. This analysis takes into account the hardware heterogeneity present in both HPC and end-user environments. The latter often possess GPUs compatible with GPGPU, but typically from mid-range or entry-level categories, whose performance falls significantly short of that provided by high-performance computing systems. Nevertheless, the results obtained from GPU experiments have demonstrated highly efficient computational performance, achieving execution times compatible with the requirements of real-time applications, or at least sufficiently low to approach such temporal thresholds.

The analysis of the computational results obtained from the CUDA-based algorithms reveals that both the coarse-grained and fine-grained approaches provide high performance; however, the relative efficiency of each one depends on the computational characteristics of the GPU device. The experimental results indicate that, for high-end GPUs such as the NVIDIA TITAN RTX and the RTX 4090, the fine-grained algorithm consistently delivers superior results, exhibiting excellent scalability. In particular, as the computational demands of the experiments increase, due to both higher resolution of the tool path and higher geometrical complexity, the speed-up achieved by the fine-grained algorithm continues to grow, demonstrating its ability to effectively leverage the available parallelism.

In contrast, on the mid-range GPU GTX 970, the performance behavior differs between the two algorithms. In such scenarios, the coarse-grained algorithm proves to be more suitable for experiments with high computational loads, such as those involving the highest toolpath resolution and shoe lasts with a large number of vertices. For less demanding cases, however, the fine-grained algorithm yields better

Table 9
Manycore coarse-grained algorithm: execution times (s) and speed-up on high-end GPU platforms.

Model	Resolution	CPU: Intel Xeon Gold 6230 time (s)	GPU: TITAN RTX time (s)	Speed-up	GPU: RTX 4090 time (s)	Speed-up
test	LR	0.11	0.15	0.74	0.21	0.52
	MR	0.69	0.16	4.35	0.24	2.86
	HR	1.94	0.21	9.47	0.24	7.98
0511pos	LR	0.20	0.22	0.95	0.25	0.82
	MR	1.15	0.18	6.25	0.26	4.46
	HR	3.21	0.26	12.20	0.25	12.60
blu22b	LR	0.60	0.26	2.33	0.32	1.90
	MR	3.80	0.33	11.48	0.32	11.83
	HR	10.19	0.40	25.80	0.31	32.56
P39f	LR	1.13	0.29	3.95	0.37	3.06
	MR	7.16	0.42	17.01	0.38	18.88
	HR	19.73	0.50	39.38	0.35	56.53
abba	LR	1.97	0.60	3.27	0.54	3.67
	MR	12.35	0.63	19.70	0.54	22.93
	HR	36.46	0.77	47.11	0.50	72.65
Papera	LR	3.42	0.93	3.70	0.81	4.21
	MR	21.70	0.99	21.99	0.82	26.39
	HR	63.10	1.28	49.37	0.76	82.91

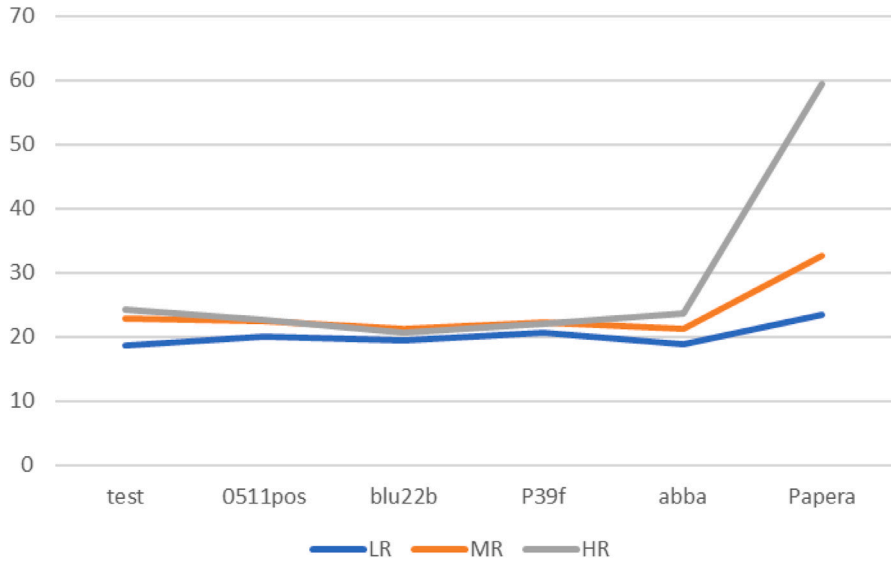


Fig. 5. Speed up of fine-grained manycore synchronous algorithm on GTX970.

performance, likely due to its more efficient use of the limited parallel resources available in such devices.

A comparison of the coarse-grained kernel across the TITAN RTX and RTX 4090 GPUs reveals distinct utilization patterns despite similar memory behavior. On the TITAN RTX, the kernel consistently achieves very high occupancy across all models and resolutions, indicating that the execution configuration effectively populates the available SM resources, while compute throughput decreases as problem resolution increases. In contrast, the RTX 4090 exhibits lower achieved occupancy across all configurations, while maintaining similar scaling trends and consistently higher compute throughput. Memory throughput remains low on both architectures, confirming that the coarse-grained kernel is compute-bound in both cases.

For the fine-grained kernel, both the TITAN RTX and RTX 4090 GPUs exhibit high compute utilization, albeit with distinct execution characteristics. On the TITAN RTX, compute throughput remains consistently high across most models and resolutions, while achieved occupancy increases with problem size, reaching high levels. On the RTX 4090, compute throughput is generally higher, whereas achieved occupancy is lower and more strongly influenced by model complexity. Unlike the coarse-grained case, memory throughput is substantially higher for the fine-grained kernel on both architectures, indicating a stronger interaction with the memory hierarchy.

These differences are consistent with architectural scaling between both GPUs. With the same launch configuration, the larger SM count of the RTX 4090 increases sensitivity to grid granularity and workload imbalance, which can reduce achieved occupancy even when overall compute-throughput trends are preserved.

Finally, it is important to highlight the applicability boundaries of the proposed approach. The VD algorithm is based on point cloud representations, which simplify geometric processing while introducing an approximation error that depends on the sampling density; this effect can be minimized by increasing the resolution. Furthermore, the fine-grained GPU implementation assumes that, at each helicoidal step, the relevant local surface section can be represented by a limited number of points, constrained by the architectural limit on the number of CUDA threads per block. While these assumptions are fully satisfied for the industrial shoe-last models and trajectory resolutions considered

Table 10
Nsight Compute analysis of the manycore coarse-grained kernel on TITAN RTX.

Model	Resolution	Compute Throughput (%)	Achieved Occupancy (%)	Memory Throughput (%)
test	LR	75.5	97.1	4.0
	MR	65.5	97.2	3.2
	HR	50.3	97.2	2.0
0511pos	LR	75.5	97.1	4.0
	MR	65.5	97.2	3.2
	HR	50.3	97.2	2.0
blu22b	LR	74.4	96.9	3.9
	MR	60.6	97.3	2.9
	HR	48.6	97.4	2.0
P39f	LR	51.4	97.5	2.1
	MR	51.4	97.5	2.1
	HR	48.6	97.4	2.0
abba	LR	72.7	97.0	3.8
	MR	60.2	97.2	2.9
	HR	48.6	97.4	2.0
Papera	LR	74.4	96.9	3.9
	MR	60.6	97.3	2.9
	HR	47.8	97.4	1.9

Table 11
Nsight Compute metrics for the coarse-grained kernel on the RTX 4090.

Model	Resolution	Compute Throughput (%)	Achieved Occupancy (%)	Memory Throughput (%)
test	LR	79.8	62.4	4.3
	MR	72.4	61.2	3.8
	HR	71.0	56.8	3.4
0511pos	LR	80.0	62.5	4.2
	MR	72.8	61.3	3.7
	HR	71.5	56.9	3.3
blu22b	LR	78.6	62.1	4.1
	MR	71.9	61.0	3.6
	HR	71.5	56.7	3.3
P39f	LR	79.7	62.3	4.2
	MR	72.1	61.0	3.7
	HR	76.7	56.6	3.5
abba	LR	77.4	62.4	4.1
	MR	72.6	61.1	3.7
	HR	71.2	56.7	3.3
Papera	LR	77.9	62.4	4.1
	MR	72.2	61.3	3.7
	HR	70.6	57.2	3.3

in this study, they may become restrictive for objects with extremely high local geometric complexity or substantially larger scales. In such scenarios, the proposed coarse-grained algorithm provides a suitable alternative. Overall, these considerations indicate that the optimal implementation strategy depends not only on the target hardware, but also on the geometric characteristics and resolution requirements of the application domain.

7. Conclusions

This work has presented and validated an efficient strategy for toolpath generation based on the VD algorithm, specifically adapted to the needs of customized manufacturing within the Industry 5.0 paradigm. By leveraging both multicore and manycore parallel computing architectures, we demonstrated that it is possible to meet real-time or near-real-time constraints even when handling high-fidelity 3D models, as required in shoe last manufacturing.

From the multicore perspective, the computation was accelerated by up to 32× when using 40 threads on 40 physical cores. On average, using 40 threads, corresponding to the number of physical cores, yielded a parallel efficiency of 94% under *dynamic* scheduling, proving the suitability of this approach for HPC environments.

The manycore algorithms designed showed even more remarkable results. The coarse-grained asynchronous GPU version achieved speedups of up to $70x$ on mid-range hardware (NVIDIA GTX 970), while the fine-grained synchronous implementation achieved speedups of up to $87x$ on the NVIDIA TITAN RTX and up to $276x$ on the NVIDIA RTX 4090, particularly in the most computationally demanding scenarios such as the Papera model at high resolution. Notably, execution times in all cases on both the TITAN RTX and the RTX 4090 remain well below one second; in particular, the RTX 4090 achieves execution times in the order of a few milliseconds, clearly satisfying real-time processing constraints. These results demonstrate that the proposed fine-grained strategy is especially well suited for modern state-of-the-art GPU architectures.

Future work will investigate hybrid CPU–GPU orchestration strategies to dynamically balance computational workloads and maximize hardware utilization. Additionally, OpenCL support will be incorporated to ensure compatibility with devices lacking native CUDA support, including embedded GPUs, thereby extending the applicability of the proposed methods to a broader range of computing platforms.

CRediT authorship contribution statement

Héctor Migallón: Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Funding acquisition. **Antonio Jimeno-Morenilla:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Funding acquisition, Conceptualization. **Eduard Duta-Costache:** Writing – original draft, Software, Investigation. **José-Luis Sánchez-Romero:** Writing – review & editing, Writing – original draft, Investigation, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Hector Migallon reports article publishing charges was provided by Spanish MCIN and AIE. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research was partially supported by grant PID2021-123627OB-C55, PID2024-158682OB-C33 and by grant PID2023-152804OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by “ERDF A way of making Europe”. We also want to thank the Spanish Technological Institute for Footwear Research (INESCOP) for providing the shoe last of footwear used for the experiments.

Data availability

The data used in this study consist of real industrial 3D shoe-last models provided by an industrial partner. Due to confidentiality agreements and intellectual property restrictions, these datasets are not publicly available. Access to the data may be granted upon reasonable request and subject to permission from the data owner.

Table 12
Fine-grained algorithm: execution times (s) and speed-up on high-end GPU platforms.

Model	Resolution	CPU: Intel Xeon Gold 6230	GPU: TITAN RTX	Speed-up	GPU: RTX 4090	Speed-up
test	LR	0.11	0.007	15.94	0.001	111.56
	MR	0.69	0.024	28.82	0.006	115.27
	HR	1.94	0.060	32.36	0.016	121.34
0511pos	LR	0.20	0.009	22.70	0.002	102.14
	MR	1.15	0.037	31.07	0.011	104.52
	HR	3.21	0.097	33.09	0.029	110.67
blu22b	LR	0.60	0.015	40.32	0.004	137.44
	MR	3.80	0.086	44.17	0.020	189.91
	HR	10.19	0.166	61.40	0.073	139.62
P39f	LR	1.13	0.019	59.69	0.005	226.83
	MR	7.16	0.092	77.84	0.028	255.76
	HR	19.73	0.228	86.53	0.075	263.05
abba	LR	1.97	0.033	59.64	0.009	218.68
	MR	12.35	0.156	79.18	0.049	252.07
	HR	36.46	0.415	87.86	0.132	276.22
Papera	LR	3.42	0.063	54.30	0.017	201.23
	MR	21.70	0.291	74.59	0.087	249.48
	HR	63.10	0.776	81.31	0.237	266.23

Table 13
Nsight Compute analysis of the fine-grained kernel on TITAN RTX.

Model	Resolution	Compute Throughput (%)	Achieved Occupancy (%)	Memory Throughput (%)
test	LR	79.8	86.9	45.9
	MR	80.6	92.1	46.5
	HR	81.0	93.0	46.8
0511pos	LR	80.0	87.4	46.2
	MR	81.0	92.7	46.8
	HR	81.2	93.3	46.8
blu22b	LR	78.9	88.1	45.8
	MR	80.7	92.4	46.6
	HR	81.3	93.5	46.9
P39f	LR	79.6	88.7	45.9
	MR	80.9	92.8	46.7
	HR	81.4	93.6	47.0
abba	LR	72.1	71.9	45.6
	MR	76.0	74.9	46.7
	HR	76.2	74.9	46.8
Papera	LR	72.1	71.9	45.6
	MR	71.8	73.1	45.4
	HR	72.8	71.9	46.0

Table 14
Nsight Compute metrics for the fine-grained kernel on the RTX 4090.

Model	Resolution	Compute Throughput (%)	Achieved Occupancy (%)	Memory Throughput (%)
test	LR	79.8	86.9	45.9
	MR	80.6	92.1	46.5
	HR	81.0	93.0	46.8
0511pos	LR	80.0	87.4	46.2
	MR	81.0	92.7	46.8
	HR	81.2	93.3	46.8
blu22b	LR	78.9	88.1	45.8
	MR	80.7	92.4	46.6
	HR	81.3	93.5	46.9
P39f	LR	76.4	75.0	71.0
	MR	87.6	74.4	66.4
	HR	87.8	74.8	65.9
abba	LR	86.1	72.9	71.0
	MR	87.6	74.7	72.3
	HR	87.8	74.9	69.6
Papera	LR	85.9	47.9	71.1
	MR	87.2	47.9	72.2
	HR	87.4	47.9	72.3

References

[1] Banáš D, Chovanová HH. Agile manufacturing vs. Lean manufacturing. Research Papers Faculty of Materials Science and Technology Slovak University of Technology, 2023;31(52):58–67, https://reference-global.com/download/article/10.2478/rput-2023-0007.pdf. (Accessed 07 January 2026).

[2] Pant R, Singh R, Gehlot A, Thakur AK. A systematic review of emerging industry 5.0 technologies: Enhancing agile manufacturing for sustainability. In: Operations research forum, vol. 6, (1):Springer; 2025, p. 29. http://dx.doi.org/10.1007/s43069-025-00427-y.

[3] Chen S-C, Chen H-M, Chen H-K, Li C-L. Multi-objective optimization in industry 5.0: Human-centric AI integration for sustainable and intelligent manufacturing. Processes 2024;12(12):2723. http://dx.doi.org/10.3390/pr12122723.

[4] Davia M, Jimeno-Morenilla A, Salas F. Footwear bio-modelling: An industrial approach. Computer-Aided Des 2013;45(12):1575–90. http://dx.doi.org/10.1016/j.cad.2013.08.006.

[5] Zhang C, Wang Z, Zhou G, Chang F, Ma D, Jing Y, Cheng W, Ding K, Zhao D. Towards new-generation human-centric smart manufacturing in industry 5.0: A systematic review. Adv Eng Informatics 2023;57:102121. http://dx.doi.org/10.1016/j.aei.2023.102121.

[6] Tallat R, Hawbani A, Wang X, Al-Dubai A, Zhao L, Liu Z, Min G, Zomaya AY, Alsamhi SH. Navigating industry 5.0: A survey of key enabling technologies, trends, challenges, and opportunities. IEEE Commun Surv Tutor. 2023;26(2):1080–126. http://dx.doi.org/10.1109/COMST.2023.3329472.

[7] Medina-Rodríguez N, Montiel-Ross O, Sepúlveda R, Castillo O. Tool path optimization for computer numerical control machines based on parallel ACO.. Eng Lett 2012;20(1). https://www.engineeringletters.com/issues_v20/issue_1/EL_20_1_13.pdf.

[8] Qudeiri JEA, Raid A-M, Jamali MA, Yamamoto H. Optimization hole-cutting operations sequence in CNC machine tools using GA. In: 2006 international conference on service systems and service management, vol. 1, IEEE; 2006, p. 501–6. http://dx.doi.org/10.1109/ICSSSM.2006.320513.

[9] Vivanco JM, Keinert M, Lechler A, Verl A. Analysis and design of computerized numerical controls for execution on multi-core systems. Procedia CIRP 2016;41:864–9. http://dx.doi.org/10.1016/j.procir.2015.12.021.

[10] Kaiser B, Keinert M, Lechler A, Verl A. CNC tool path generation on multi-core processors. Appl Mech Mater 2015;794:339–46. http://dx.doi.org/10.4028/www.scientific.net/AMM.794.339.

[11] Szczepanski R, Erwinski K, Paprocki M. Accelerating PSO based feedrate optimization for NURBS toolpaths using parallel computation with OpenMP. In: 2017 22nd international conference on methods and models in automation and robotics. MMAR, IEEE; 2017, p. 431–6. http://dx.doi.org/10.1109/MMAR.2017.8046866.

[12] Yi J, Chu C-H, Kuo C-L, Li X, Gao L. Optimized tool path planning for five-axis flank milling of ruled surfaces using geometric decomposition strategy and multi-population harmony search algorithm. Appl Soft Comput 2018;73:547–61. http://dx.doi.org/10.1016/j.asoc.2018.08.041.

[13] Morell-Giménez V, Jimeno-Morenilla A, García-Rodríguez J. Efficient tool path computation using multi-core GPUs. Comput Ind 2013;64(1):50–6. http://dx.doi.org/10.1016/j.compind.2012.09.009.

[14] Abecassis F, Lavernhe S, Tournier C, Boucard P-A. Performance evaluation of CUDA programming for 5-axis machining multi-scale simulation. Comput Ind 2015;71:1–9. http://dx.doi.org/10.1016/j.compind.2015.02.007.

[15] Konobrytskyi D, Hossain MM, Tucker TM, Tarbutton JA, Kurfess TR. 5-axis tool path planning based on highly parallel discrete volumetric geometry representation: Part I contact point generation. Computer-Aided Des Appl 2018;15(1):76–89. http://dx.doi.org/10.1080/16864360.2017.1353730.

[16] Balabokhin A, Tarbutton J. Iso-scallop tool path building algorithm "based on tool performance metric" for generalized cutter and arbitrary milling zones in 3-axis CNC milling of free-form triangular meshed surfaces. J Manuf Process 2017;28:565–72. http://dx.doi.org/10.1016/j.jmapro.2017.04.025.

[17] Wang J, Luo M, Zhang D. A GPU-accelerated approach for collision detection and tool posture modification in multi-axis machining. IEEE Access 2018;6:35132–42. http://dx.doi.org/10.1109/ACCESS.2018.2848938.

[18] Faust RC, Minetto R, Volpato N. Parallel tool-path generation for additive manufacturing: a GPU-based zigzag filling. Adv Ind Manuf Eng 2023;6:100107. http://dx.doi.org/10.1016/j.aime.2022.100107.

[19] Hsieh H-T, Chu C-H. Particle swarm optimisation (PSO)-based tool path planning for 5-axis flank milling accelerated by graphics processing unit (GPU). Int J Comput Integr Manuf 2011;24(7):676–87. http://dx.doi.org/10.1080/0951192X.2011.570792.

[20] Wang J, Zhang D, Luo M, Zhang Y. A GPU-based tool parameters optimization and tool orientation control method for four-axis milling with ball-end cutter. Int J Adv Manuf Technol 2019;102:1107–25. http://dx.doi.org/10.1007/s00170-018-2954-1.

[21] Li Z, Xiong G, Zhang X, Shen Z, Luo C, Shang X, Dong X, Bian G-B, Wang X, Wang F-Y. A GPU based parallel genetic algorithm for the orientation optimization problem in 3D printing. In: 2019 international conference on robotics and automation. ICRA, IEEE; 2019, p. 2786–92. http://dx.doi.org/10.1109/ICRA.2019.8793989.

[22] Schnös F, Hartmann D, Obst B, Glashagen G. GPU accelerated voxel-based machining simulation. Int J Adv Manuf Technol 2021;115(1):275–89. http://dx.doi.org/10.1007/s00170-021-07001-w.

[23] Rico-Garcia H, Sanchez-Romero J-L, Gomis HM, Rao RV. Parallel implementation of metaheuristics for optimizing tool path computation on CNC machining. Comput Ind 2020;123:103322. http://dx.doi.org/10.1016/j.compind.2020.103322.

[24] Yu W, Bi Y, Li Z. Research on tool path planning method of NURBS surface based on CPU-GPU parallel computing. In: 2017 international conference on computer network, electronic and automation. ICCNEA, IEEE; 2017, p. 85–8. http://dx.doi.org/10.1109/ICCNEA.2017.52.

[25] Jimeno A, Chamizo JMG, Salas F. Shoe last machining using virtual digitising. Int J Adv Manuf Technol 2001;17:744–50. http://dx.doi.org/10.1007/s001700170120.

[26] OpenMPArchitecture Review Board. Openmp application programming interface version 4.5. 2015, https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (Accessed 01 April 2025).

[27] Harris M. Optimizing parallel reduction in CUDA. 2007, NVIDIA Developer Technology Presentation. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. (Accessed 01 April 2025).