# Octree-based, GPU Implementation of a Continuous Cellular Automaton for the Simulation of Complex, Evolving Surfaces

N. Ferrando[*,a], M. A. Gosálvez[b,c], J. Cerdá[a], R. Gadea[a], K. Sato[b]

[a] *Instituto de Instrumentación de Imagen Molecular, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain*
[b] *Dept. of Micro-Nanosystems Engineering, Nagoya University, 464-8603 Aichi, Japan*
[c] *Dept. of Materials Physics, University of the Basque Country (UPV-EHU), Donostia International Physics Center (DIPC), and Spanish National Research Council (CSIC), 20018 Donostia - San Sebastian, Spain*

**Abstract**

Presently, dynamic surface-based models are required to contain increasingly larger numbers of points and to propagate them over longer time periods. For large numbers of surface points, the octree data structure can be used as a balance between low memory occupation and relatively rapid access to the stored data. For evolution rules that depend on neighborhood states, extended simulation periods can be obtained by using simplified atomistic propagation models, such as the Cellular Automata (CA). This method, however, has an intrinsic parallel updating nature and the corresponding simulations are highly inefficient when performed on classical Central Processing Units (CPUs), which are designed for the sequential execution of tasks. In this paper, a series of guidelines is presented for the efficient adaptation of octree-based, CA simulations of complex, evolving surfaces into massively parallel computing hardware. A Graphics Processing Unit (GPU) is used as a cost-efficient example of the parallel architectures. For the actual simulations, we consider the surface propagation during anisotropic wet chemical etching of silicon as a computationally challenging process with a wide-spread use in microengineering applications. A continuous CA model that is intrinsically parallel in nature is used for the time evolution. Our study strongly indicates that parallel computations of dynamically evolving surfaces simulated using CA methods are significantly benefited by the incorporation of octrees as support data structures, substantially decreasing the overall computational time and memory usage.

*Key words:* parallel computing, dynamic surface, octree, graphics processing unit (GPU), Many-core processors, cellular automata (CA), anisotropic wet etching
*PACS:* 89.20.Ff, 85.85.+j, 81.65.Cf

## 1. Introduction

Dynamic surfaces appear often in a wide variety of physical systems. These surfaces vary their overall shape, size and morphology as the underlying physics and/or chemistry change during the evolution. Several examples of this behavior include the propagation of a wave front [1, 2], the conformal growth of thin films on three-dimensional landscapes during atomic layer deposition [3, 4] or the complex propagation of the etch front in chemical etching for microengineering applications [5, 6]. When the desired system resolution requires a large number of surface points and the underlying propagation model requires processing a large fraction of those points in each time step, the use of classical sequential processors can lead to highly inefficient simulations. This is the case of typical Cellular Automata (CA) models, where groups of points (or cells) with identical states need to be processed simultaneously. On traditional computers with a single Central Processing Unit (CPU), one is forced to use a computational loop structure, which will be sequentially executed by the CPU. As a result, the simulations are characterized by a linear computing complexity, where the amount of calculations depends on the size of the modeled surface. This has been usually reported as a limiting factor whose only solution might be the use of simpler but less accurate physical or chemical models [7].

In recent years, however, a huge increase in the development and commercialization of parallel computing architectures has occurred. Multicore processors have reached the mainstream market and Many Core Architectures (MCA) such as Graphics Processing Units (GPUs) have become an affordable alternative for the computation of massively parallel algorithms [8, 9]. Traditionally developed for improved and faster graphics, this type of environment has recently become popular also for scientific applications due to the high performance-to-price ratio of most GPU cards. Although these novel architectures provide new possibilities and promise significant speedups, they also impose new restrictions to traditional data structures, which need to be re-evaluated and adapted for the new environments.

One such example is the sparse, spatially-organized data structure of an octree [10]. This structure is very beneficial in

[*]Corresponding author.
Address: Instituto de Instrumentación de Imagen Molecular, Universidad Politcnica de Valencia, Camino de Vera S/N CP 46022 Valencia, Spain.
E-mail: nesferjo@upvnet.upv.es
Phone: +34 626401251
Fax: +34 963879609

a single-processor, single-memory environment as it provides a good balance between low memory and reasonably fast access to its elements, as well as acceptable searching properties [11, 12, 13]. In this way, the octree offers wide flexibility in order to represent an evolving surface that is reshaped continuously. As an example, octrees have been successfully used to describe anisotropically wet etched silicon surfaces, giving rise to complex, often multivalued, three-dimensional structures of ample use in microengineering applications in the area of Micro Electro Mechanical Systems (MEMS) [11]. The wide use of wet etching is due to its many advantages, including its low cost, the simplicity of the experimental setup, the ability to perform batch fabrication, an unmatched capability to release mechanical structures and the excellent uniformity and surface quality. Incidentally, it has been recently shown that the propagation of the evolving front during wet etching can be accurately modeled using an intrinsically parallel method known as the Continuous Cellular Automaton (CCA) [11, 14, 15, 16]. The major drawback, however, is the relatively low efficiency of the simulations when performed on a traditional CPU environment. On the contrary, affordable GPUs have demonstrated remarkable performance for the simulation of CA-based models [17].

In this paper, the use of the octree structure for the management of dynamically evolving, multivalued, complex surfaces is revised in order to meet the particular needs of the GPU architecture, serving as an example for other many-core architectures. By using the CCA method as an example of a strongly parallel evolution model and anisotropic etching of silicon as a relevant application that leads to the formation of complex, dynamically evolving surfaces, we present a series of guidelines in order to optimize this type of simulation on massively parallel computing environments. Not only the results show that cellular automata simulations of dynamically evolving surfaces can be successfully adapted to a parallel computing GPU environment, thus circumventing the slowness of a CPU-only implementation, but more importantly, the results also show that the incorporation of the octree data structure is an effective procedure to minimize the amount of GPU memory use and computational effort, thus resulting in faster overall simulation times and enabling the exploration of larger-size and/or larger-resolution systems.

## 2. Describing surfaces using octrees: adaptation to the GPU architecture

### *2.1. Using octrees to describe complex, evolving surfaces*

The dynamic surfaces considered in this study are composed of points on which the model equations are applied. The points may correspond to lattice nodes located at the interface between two different physical regions (or materials), or the atoms at the surface of a growing/etched crystal. A way to avoid storing the full lattice or crystal in the computer memory is to regard the surface points as the leaves of a tree [10, 11, 14]. The idea is to represent the full computational cubic region as the root node of the tree. This region is then subdivided into eight subregions, represented by eight children nodes in the tree, directly connected to the root node. The non-empty subregions (containing surface points) are further subdivided –and corresponding nodes are inserted in the tree– until partitioning cannot be performed any further due to the finite size of the original set of surface points (Figure 1(a)). The cubic region defined by the whole tree can grow by just embedding the root node as a child of a new root node that encloses a bigger domain. This type of data structure, known as an octree, has been used by the computer graphics community in many different contexts (*e.g.* space traversal acceleration in Ray Tracing, efficient collision detection) or for scientific simulation purposes (*e.g.* adaptive simulations, parallel dynamic decompositions, etc...).

The main two characteristics of octrees are (i) the ability to provide data storage using reduced memory, since the space portions without relevant information are kept undivided, and (ii) a moderate cost $O(\lceil log_8(N) \rceil)$ for accessing the leaf nodes at the finest level, where $N$ is the number of nodes. In other words, an octree provides a good balance between memory use and access time. As shown in Figure 1(b), these features are specially useful in order to store dynamic surfaces in the computer memory since only the space surrounding the surface needs to the partitioned in detail [10]. In particular, octrees have been successfully used for the simulation of dynamic surface-based models [14].

In our case, the minimum spatial region associated to the leaf nodes of the octree is an *orthorhombic unit cell* containing a small number of silicon atoms, as shown in Figure 2(a). This is based on the fact that any crystalline material, such as silicon, can be constructed by repeating the unit cell along three independent directions. An orthorhombic unit cell is a stretched/squeezed cubic cell with all angles equal to 90 deg and has the advantage that the three directions are orthogonal. The size, aspect ratio and atom contents of this unit cell depend on the crystallographic orientation of the simulated surface, *e.g.* (100), (110), (221), etc..., where (hkl) are the Miller indices. More generally, the minimum spatial region will contain points from a discretization of the space where the physical model is solved.

To improve the parallel computing efficiency, as considered in Section 3.6, the concept of *supercell* is used in this study. The name "supercell", which refers to a collection of several (or many) adjoint unit cells, as described in Figure 2(b), is borrowed from the well established computational field of Density Functional Theory [18]. The term is used in this study to refer indistinctively to both the crystallographic unit cell and any multiple of it. In this manner, a supercell is the smallest space partition associated to the leaf nodes of the octree, as shown on the left-hand-side of Figure 2. Although generally each atom is regarded as *a cell* in cellular automata applications, in this work such identification is avoided in order to minimize confusion with the unit cell and supercell concepts. For the cellular automaton used in this work, the supercell is understood simply as a collection of atoms, as described in Figure 2(c). The number of atoms in the supercell, regarded as $M$, is different for different surface orientations.

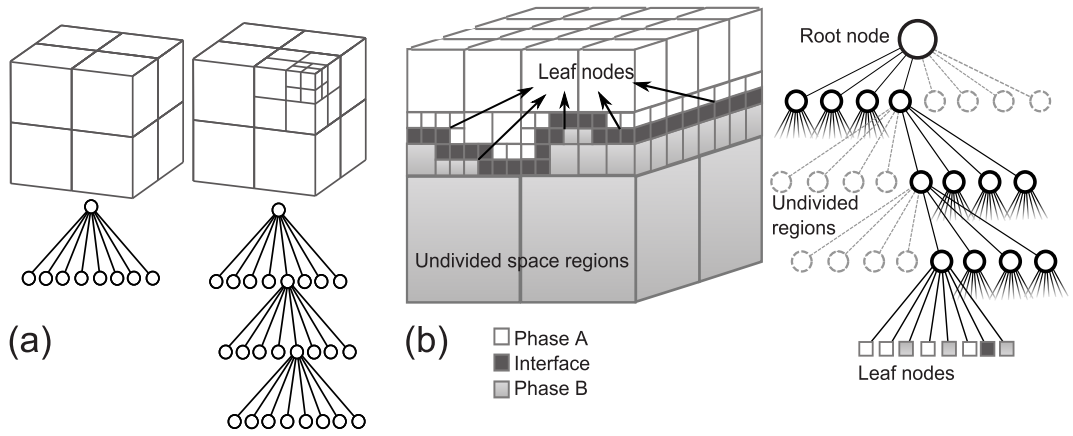


Figure 1: (a) Space discretization and octree depth. (b) 3D surface as an interface between two phases and corresponding octree structure.

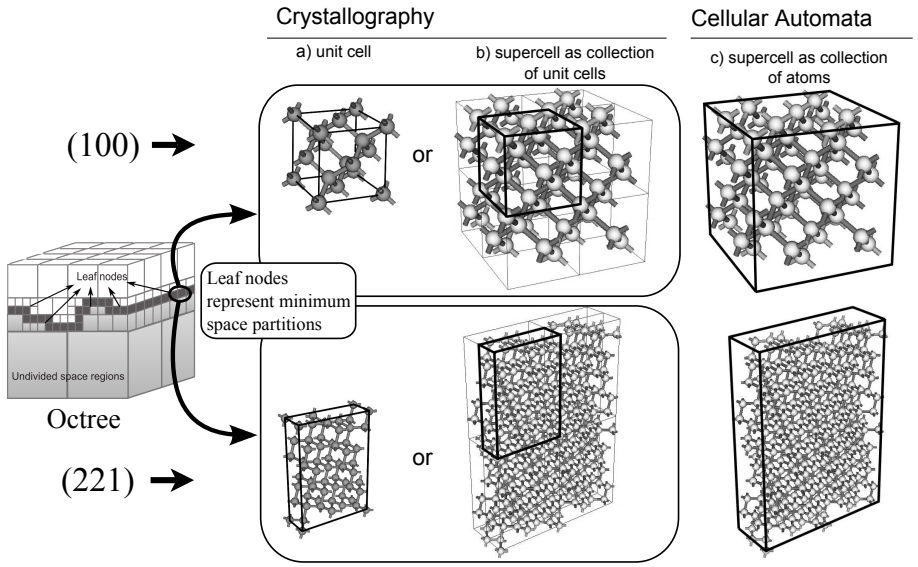


Figure 2: Leaf nodes of the octree represent the minimum spatial region. For a crystalline structure, this minimum space region can be defined as: (a) an orthorhombic unit cell or (b) as a group of unit cells, namely a supercell. (c) In a Cellular Automaton the supercell is simply a stretched-box-like collection of $M$ atoms.

### *2.2. Using octrees in the GPU environment*

In the last years, GPUs have demonstrated an affordable, yet powerful architecture for the computation of massively parallel algorithms. We choose the CUDA architecture as a general purpose parallel computing platform, available for the latest generations of GPUs developed by Nvidia [23]. From the perspective of CUDA, a GPU is abstracted as a device that contains an array of independent cores, an interconnection network and a device memory (DRAM). The actual structure of a GPU is far more complex, with Scalar Processors (SPs) grouped into Streaming Multiprocessors (SMs) grouped into Thread Processing Clusters (TPCs), such that SMs and TPCs contain arithmetic units and other resources (such as shared memory, caches, texture fetch units, etc...) that not only vary with the GPU series model but are also mainly hidden from the programmer. A "core" can typically be pictured as one SM. Each core is able to execute a large number of threads simultaneously. A typical CUDA function, referred to as a *kernel*, is performed in parallel by a large number of threads, all of which execute the same algorithm. The threads, organized in blocks, are sent to the cores for execution. Inside each core, on-chip *shared memory* allows fast communication between the threads of a block. In addition, the GPU has a *global memory* that enables persistent data storage along application lifetime and a (slower) inter-thread communication between different thread blocks. Each core has fast read access to both a *texture memory cache*, optimized for 2D spatial locality, and a *constant memory cache*. The programmer can use reserved keywords, such as *__constant__*, *__shared__*, etc..., in order to control the storage of the variables in the different memories. Finally, the CUDA environment provides atomic read/write operations (on the shared and global memories), which guarantee that only one thread can access a certain memory address until the operation is complete.

To maximize the performance in this environment, the program code should exercise moderate access to the GPU global memory and, when doing so, the access pattern should be optimized in order to achieve *memory coalescing* (see Section 3.3). In addition, the code should try to maximize the number of ac-

tive threads in each core as well as minimize the use of flow control and atomic instructions, as these can easily disrupt the parallel execution of the thousands of threads that should be simultaneously handled. In general terms, as indicated below, we use (i) the constant cache to store data that do not change during the simulation, such as the silicon structure and the etch rates for the chosen etchant; (ii) the global memory to store most model data, using optimized access patterns; (iii) the texture cache for some model data, such as the neighbors of the currently processed atom, in order to improve non-optimal read access that has spatial locality; (iv) shared memory in some particular cases.

The computing strength of most parallel architectures is based on massively concurrent execution of threads. Sequentiation of any procedure results in slow and inefficient algorithms. In principle, adaptation of the dynamic surfaces described in Section 2.1 into many core architectures, such as a GPU, can be done simply by grouping the surface points into tasks in order to compute the tasks concurrently using multiple threads. However, in a GPU the management of a surface using an octree suffers from one crucial restriction, namely, the allocation and deallocation of space portions of the octree become *critical sections* in the program code. This means that uncontrolled simultaneous execution by two or more threads can cause data corruption. An example of this problem occurs when two concurrent threads proceed to allocate a new octree node. Without any synchronization, a typical *race condition* may occur, where both threads attempt to use the same memory region for storing model data belonging to different leaf nodes. Similarly, the decision about the future state of any space portion can also be considered as a *critical section*. As an example, if two threads try to modify the state of the same space portion at the same time, they can allocate or deallocate it twice, resulting in unknown behavior. As a result, thread exclusion for octree management becomes mandatory. Although it is theoretically possible to implement intensive global thread exclusion through atomic read/write operations, the read/write access to the stored octree will become random, thus resulting in an inefficient access pattern. Although both operations –atomic instructions and random access– are possible on GPUs, they are far from the ideal CUDA computing paradigm and their use will typically lead to very inefficient implementations.

In order to efficiently implement concurrent thread execution, we propose to decouple the dynamic surface management and the model calculations: the dynamic surface is completely stored in the GPU global memory so that all the tasks related to the model evolution can be quickly performed by the GPU, while the underlying tree structure is stored in a classical CPU so that the decisions related to the allocation and removal of octree regions can be properly carried out by the CPU. In order to implement this approach, it is convenient that the data storage is performed according to the following three guidelines:

- The GPU global memory is regarded as an array of Memory Clusters (MCs). Each MC is used to store model data for one supercell (Figure 3(a)). Thus, the number of assigned MCs is equal to the number of currently used supercells.
- The CPU is provided with a repository of pointers to all the currently available, non-used MCs (Figure 3(b)).
- Although the octree management is performed by the CPU, the octree leaves that form the dynamic surface are actually pointers to the assigned MCs (Figure 3(b)).

This means that each octree leaf stored in the CPU points to a supercell, which contains $M$ atoms and is stored as a MC in the GPU. With this approach the surface is completely stored in the GPU memory and the CPU has the necessary information about the used and free MCs in the GPU, making possible to adequately decide whether allocation and/or removal of spatial regions is needed in the GPU.

The proposed decoupling requires continuous communication between both platforms. While processing the dynamic surface, the algorithms executed in the GPU may decide that a particular region needs to be updated (*e.g.* a supercell is not needed anymore and can be eliminated). For this purpose, each MC has a region for indicating, as a flag, if it needs any type of update. The execution threads in the GPU will activate the MC flags when needed, indicating what type of action the CPU should perform on the corresponding MC. The CPU will receive and analyze the resulting *flag array* and, if any update is detected, it will determine which supercell is affected from a lookup table of assigned MCs. By knowing the supercell and update type, the CPU is able to access the repository of used / non-used pointers in order to *add/remove* an octree leaf node. Finally, the CPU must inform back to the GPU about the newly assigned and/or cleared MCs. Figure 3(c) shows a diagram of this procedure. Using this method, all the calculations related to the octree management are made in the CPU while the parallel calculations related to the model equations are performed by the GPU.

Due to this decoupling, the positions of the supercells in 3D space do not have any correlation to the locations of the corresponding MCs in the GPU global memory. Since many physical/chemical models require inspecting the neighborhood of each surface atom in order to decide the next series of events, enabling fast access to the neighbor supercells becomes necessary, especially for the atoms located close to the supercell boundaries. Furthermore, keeping the number of memory reads independent of the model complexity becomes a relevant feature when finding the neighbors. A data structure that can achieve these two functionalities is a simple look-up table that stores pointers to the neighbor supercells for each MC (see Section 3.2 ). This table should be stored in the GPU global memory. In order to keep the information updated, changes in the surface morphology during the simulation should be immediately reflected in this table. As the CPU keeps the information about the spatial organization of the GPU clusters, the CPU is responsible for sending the updates to the GPU.

Finally, regarding the repository of used / non-used MCs held in the CPU, a stack data structure works well enough. Stack access times are constant ($O(1)$) and, due to its LIFO (last in first out) behavior, a recently released memory segment will be the next one to be used. This feature prevents excessive dispersion of the supercells along the GPU global memory. Keeping the data essentially gathered together makes the parallel algorithms

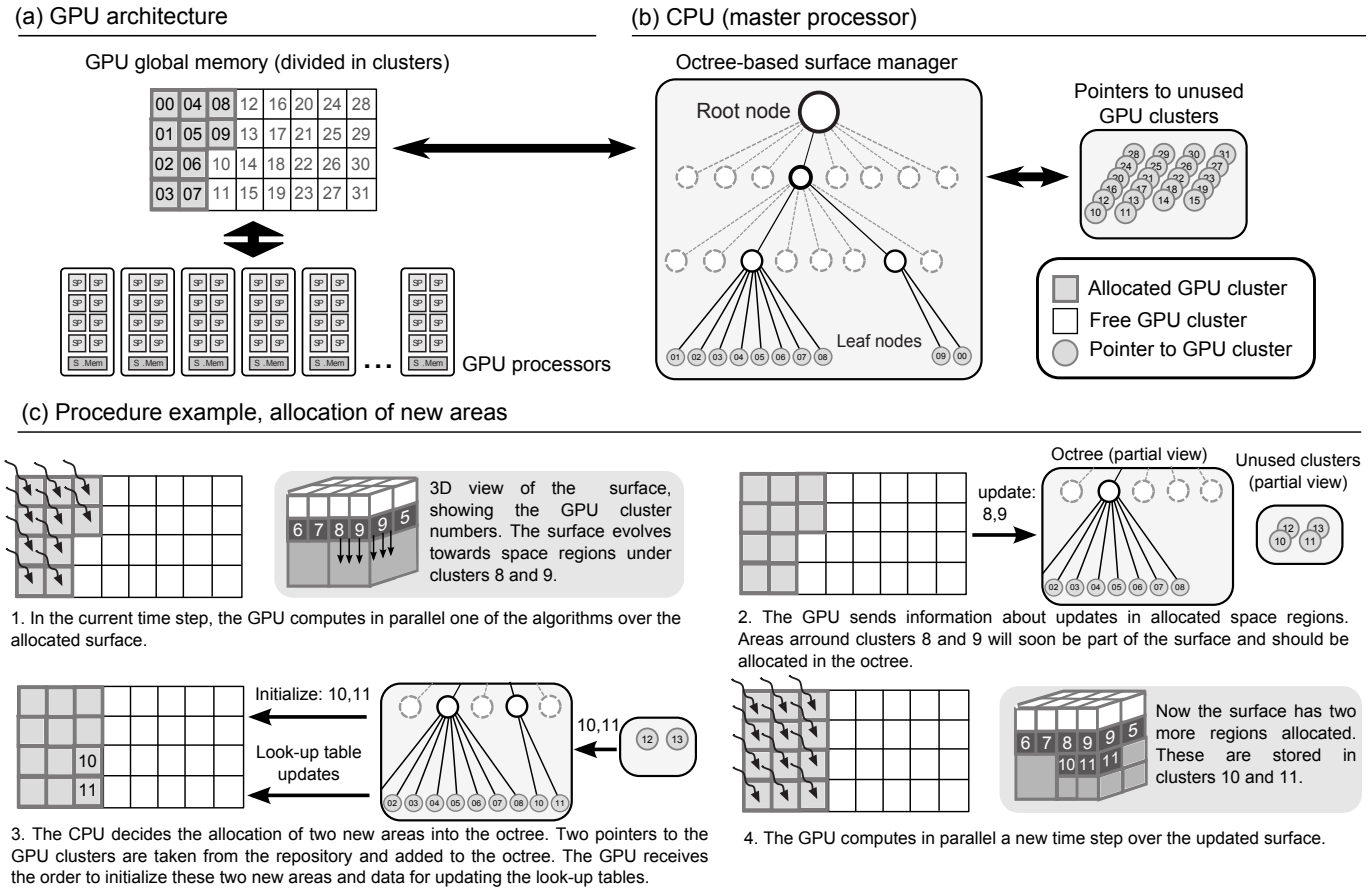


Figure 3: Proposed implementation for the efficient simulation of dynamic surfaces described using octrees on a GPU environment. (a) GPU memory divided into clusters. (b) Data structures stored in the master processor memory (CPU). (c) Procedure example for the addition of new subregions into a simulated surface.

more efficient since full memory occupancy until a certain location can be assumed and, thus, data processing needs to be performed up to that point only.

## 3. Example application

The methodology presented in Section 2 to incorporate the octree data structure in a GPU environment is applied to the simulation of the evolving front during anisotropic wet chemical etching of silicon using a Continuous Cellular Automaton (CCA) for the time evolution.

### *3.1. Physical model for wet etching*

Wet etching consists on the removal of material from a solid sample immersed in an alkaline or acidic solution [19]. It is the result of chemical reactions between the surface atoms and the etchant phase molecules. Wet etching is a popular process for the creation of microstructures in a wide range of applications, such as pressure sensors [20], gas-flow sensors [21] or micro-switches [22]. In this process, the advancing front propagates with a different rate depending on the crystallographic direction. Certain crystallographic orientations, such as {111}, are etched at a lower rate than others. In simple terms, such planes are etched slowly due to stronger atom bonds and/or larger atom coordination. Combined with the use of mask patterns, the etch-rate anisotropy becomes a valuable property as it provides a low-cost, precise means for the production of three-dimensional shapes delimited by smooth, shiny facets. Slow etching planes are developed at concave features of the masking patterns while fast etching planes compete to reshape the convex features. Due to dependencies on the composition of the etchant, such as its concentration and temperature, the three-dimensional etched shapes are difficult to grasp intuitively. Furthermore, their correct description by means of simulations has traditionally remained a challenge.

Recently, however, anisotropic etching has been successfully simulated using a CCA model [15]. Etching is considered as a *step-flow* process where rows of silicon atoms located along morphological steps are removed in an ordered manner. Under closer inspection, the underlying theory makes a classification of the silicon surface sites depending on the number of surface nearest neighbors ($n^{1s}$), interior nearest neighbors ($n^{1b}$, where "b" stands for "bulk", *i.e.* interior), surface next-nearest neighbors ($n^{2s}$) and interior next-nearest neighbors ($n^{2b}$). Taking into account this information, it is possible to exactly classify the different atom configurations on the surface of a silicon substrate. These configurations are used to determine the reaction rates of the surface atoms. In particular, the removal rates for all atom types can be obtained mathematically using the exper-

imental etch rates for several surface orientations [14, 15, 16].

Inside the model, the etching process is understood as a continuous decrease of a scalar field, referred to as the *occupation (*$\Pi$*)*, which is defined to take values at the locations of the surface atoms (*i.e.* the surface *sites*). When an interior surface atom becomes a surface atom, $\Pi$ is initialized with value '1'. During the simulation, $\Pi$ is gradually reduced until it reaches value '0' (completely removed). At each time step ($k$), the reduction in the occupation of the $i$-th surface site is equal to the current value of the removal rate of the atom that populates the site ($r_i$), multiplied by the time step ($\Delta t$):

$$\begin{aligned} &\Pi_i(k+1) = \Pi_i(k) - \Delta t \cdot r_i(k), \\ &\qquad \text{where} \quad r_i(k) = R(n_i^{1s}(k), n_i^{1b}(k), n_i^{2s}(k), n_i^{2b}(k)). \end{aligned} \tag{1}$$

The atom removal rates $r_i$ depend on the number of neighbors, as indicated in the second line of Equation 1. Here, $R$ is a function that depends on the simulated etchant type (such as KOH, TMAH, etc...), including its concentration and temperature. Since $n^{1s}$ and $n^{1b}$ can take values from 0 to 3, and $n^{2s}$ and $n^{2b}$ from 0 to 11 for surface atoms, $R$ can be thought as a table. Most of the table entries are meaningless and only a subset of 33 removal rates is typically needed [15, 16]. Until a surface atom is completely removed, its removal rate keeps changing as the numbers of surface and interior nearest and next-nearest neighbors are modified due to the removals of nearby atoms. For this reason, $r_i$ and $n_i^{1s}$, $n_i^{1b}$,... are written as functions of $k$ in Equation 1.

Although the CCA method seems to be reliable and accurate, the present academic and commercial simulators that use this model have limited functionality due to the relatively low efficiency of the calculations when performed on a traditional CPU environment.

### *3.2. GPU implementation: Main variables*

We consider the previous CCA model for wet etching as a particular example of an evolving dynamic surface. By definition, CA methods are memory-access intensive. For every atom, the state of all the neighbor atoms has to be read and the new state has to be determined and stored in each time step. It has been already reported that, for mathematical models with similar memory access patterns, such as Finite-Difference Time-Domain, the performance bottleneck for their implementation in GPU architectures is the GPU global memory bandwidth [24]. Thus, we focus on algorithm optimizations that can reduce the global memory bandwidth usage.

Our proposed implementation is able to simulate a wide range of different supercells that can vary in size and inner structure. We label all the atoms inside the supercell from 1 to $M = size(supercell)$ or, equivalently, from 0 to $M-1$, and refer to any atom in the silicon crystal by using four coordinates ($n_1$, $n_2$, $n_3$, $m$), where the trio ($n_1$, $n_2$, $n_3$) designates the supercell location and the "$m$-coordinate" ($m \in [0, M-1]$) designates the atom number inside the supercell. As an example, (2,4,-5,7) designates the 8-th atom in the supercell that is located 2 supercells away to the right of–, 4 supercells away behind– and 5 supercells away below the center supercell. Since each supercell is stored as an MC, the array position $p$ of atom $m$ in the $K$-th memory cluster is: $p = m + M * K$.

Table 1 shows the main variables used in our GPU implementation. All the model data variables are stored as large continuous arrays, allocated prior to the simulation. *num_Atoms* and *num_MC* are, respectively, the maximum numbers of atoms and MCs that can be used during the simulation. The variables *Occ* and *Erate* denote the occupation $\Pi_i$ and the current value of the removal rate $r_i$ for every surface atom $i$ on the surface, as described in Equation 1. Similarly, *TRate* denotes the removal rate table $R$ and *N1sb* stores the two variables $n^{1s}$ and $n^{1b}$ together in the same byte. In addition, the variables *CellInfo*, *UCInfo* and *TArea* refer to the look-up tables that enable the threads (being executed for each surface atom) to access the neighbor atoms, as described below. A set of additional variables, including *Syb*, *MCstate* and *TBuf* is also used.

The look-up table *CellInfo* can be pictured as a 2D matrix with 4 columns and (at most) 4096 rows, where column $j$ of row $i$ stores the $m$-coordinate of the $j$-th neighbor of [the $i$-th atom in the (0,0,0) supercell]. Correspondingly, *UCInfo* is also a 2D matrix with 4 columns and (at most) 4096 rows, where column $j$ of row $i$ stores the trio ($n_1$, $n_2$, $n_3$) of the $j$-th neighbor of [the $i$-th atom in the (0,0,0) supercell], compacting the three $n_k$ values in the same byte by using one bit per axis per direction (totaling 6 bits < one byte). The look-up tables *CellInfo* and *UCInfo* are complementary in the sense that both are required in order to know the location of a neighbor of a surface atom. The two arrays are stored in the GPU constant memory. For comparison, the look-up table *TArea* has a different structure, which can be pictured as a 2D matrix with 6 columns and *num_MC* rows, storing memory pointers to the front, back, east, west, north and south neighbor supercells, using one pointer per axis and direction. This second lookup table, already mentioned in Section 2.2, is stored in the GPU global memory and is used in order to recover the 3D location of any supercell, thus compensating the fact that neighboring supercells in 3D space are typically stored at distant MCs in the GPU memory.

It is useful to picture the silicon etching process as occasional atom removals from the surface. Only a small amount of atoms is removed in each time step and, as a result, the neighborhoods and removal rates of most atoms will remain invariant in time and do not need to be recalculated. As an example, the ratio of the number of updated atoms to the number of visited atoms is typically about 1/50 for constant time stepping simulations and decreases down to $O(1/1000)$ for variable time stepping simulations, considering relatively large removal rates and small unit cells in both cases. Thus, by keeping the neighborhood data $n^{1s}$ and $n^{1b}$ for each surface atom in the GPU global memory (see array *N1sb* in Table 1), the neighbors have to be updated only when an atom is etched away and, thus, the number of global memory accesses is reduced.

Furthermore, reading a single entry of table $R(n^{1s}, n^{1b}, n^{2s}, n^{2b})$ in Equation 1 requires in principle as many as 16 neighbor atom reads for every atom (4 nearest and 12 next-nearest neighbors). A way to reduce the number of reads to 8 is to keep $n^{1s}$ and $n^{1b}$ stored for every surface atom,

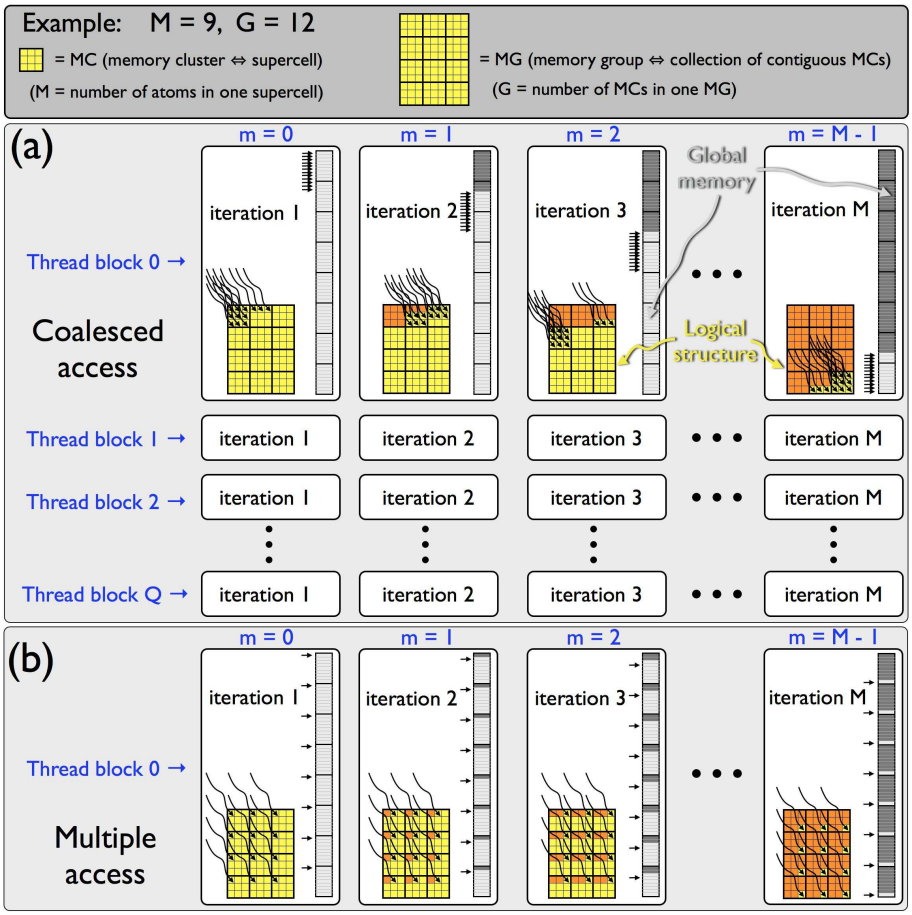


Figure 4: Graphical representation of a Memory Group (MG): (a) implementation scheme to obtain coalesced access to the GPU global memory, (b) an example of uncoalesced access.

and to compute $n^{2s}$ and $n^{2b}$ from the values of $n^{1s}$ and $n^{1b}$ of the first neighbors:

$$n_i^{2s} = \left( \sum_{j=0}^{3} n_j^{1s} \right) - 4; \qquad n_i^{2b} = \sum_{j=0}^{3} n_j^{1b}. \tag{2}$$

Although $n^{1s}$ and $n^{1b}$ are stored in the global memory (array *N1sb* in Table 1), $R$ remains invariant along all the simulation and, thus, it can be take advantage of GPU constant caches in order to reduce global memory accesses.

### 3.3. GPU implementation: Coalesced memory access

As described in Section 2.1, a Memory Cluster (MC) stores model data for one supercell (M atoms). In this study, a collection of G contiguously stored MCs in the GPU memory is referred to as a Memory Group (MG). For simplicity, we chose $G$ to be equal to the number of threads per thread block. A conceptual example of an MG containing $G = 12$ MCs is given at the top of Figure 4, where each MC stores data for $M = 9$ atoms. In practice, grouping the MCs into MGs is as easy as choosing the global memory region to be processed: *Block 0*, process from $MC_0$ to $MC_{G-1}$; *Block 1*, process from $MC_G$ to $MC_{2G-1}$; etc...

In our implementation, all threads execute a loop consisting of $M$ iterations: $m = 0, 1, ..., M - 1$. For each iteration, each thread processes one atom and each thread block processes $G$ atoms stored contiguously in the same MG, as described in Figure 4(a). In order to process all the active MCs, the adequate amount of thread blocks should be launched per kernel execution, this value is decided by the CPU, based on the total number of MCs currently active in the GPU. Effectively, each thread processes an amount of atoms equivalent to the size of the supercell ($M$ atoms), and each thread block processes a contiguous collection of atoms in the GPU memory space. Thus, this procedure provides a coalesced access pattern to the GPU global memory within each thread block, reducing to a minimum the read/write access latencies. For comparison, Figure 4(b) describes an example of non-coalesced access within each thread block, which would lead to dramatically reduced computational efficiency.

### 3.4. GPU implementation: Simulation pipeline

Figure 5 offers an abstracted view of the main simulation pipeline. Based on the total amount of MCs currently in use for storing the surface in the GPU memory, the CPU determines the amount of required GPU threads before initiating the next time step and then launches a thread grid composed of the adequate

Table 1: Main variables defined in the GPU memory space. $^{(*)}$These variables are read using the texture fetch unit in those algorithms where the underlying procedure is known to make frequent accesses to neighbor atoms/supercells.

| Name | Size(type) | Memory | Use of the variable |
|---|---|---|---|
| Occ | *num_Atoms*($float$) | global | Store current site occupation $\Pi$. |
| Erate | *num_Atoms*($float$) | global | Store current atom removal rate $r$. |
| Syb | *num_Atoms*($byte$) | global$^{(*)}$ | store atom type (bulk, surface, masked...). One bit used as flag for neighborhood updates. |
| N1sb | *num_Atoms*($byte$) | global$^{(*)}$ | Store $n^{1s}$ and $n^{1b}$ atom info (compacted as one byte). |
| MCstate | *num_MC*($short$) | global | Store number of remaining unetched atoms inside each MC. |
| TArea | $6 \cdot$ *num_MC*($int$) | global$^{(*)}$ | $2^{nd}$ level look-up table: pointers to front, back, east, west, north and south neighbor supercells. |
| TBuf | $K \cdot$*num_MC*($uint$) | global | Buffer for data transfer between CPU and GPU. |
| TRate | 4096($float$) | constant | Precalculated etch rate table $R$. |
| CellInfo | $\leq 4 \cdot 4096$($short$) | constant | $1^{st}$ level look-up table: $m$-coordinates of the four neighbors. |
| UCInfo | $\leq 4 \cdot 4096$($byte$) | constant | $1^{st}$ level look-up table: one-byte-compacted ($n_1$, $n_2$, $n_3$) for the four neighbors. |

number of blocks. In both parts a) and b), each thread block operates over an MG.

Part a) of Figure 5 has the purpose of updating the removal rates of the surface atoms. This is performed by running two similar kernels sequentially so that tasks (1)-(4) are carried out by the second kernel only after they have been completed by the first kernel. First of all, either kernel checks if updates are needed by reading the array *Syb*, which has a one-bit flag to indicate if changes have occurred in the neighborhood. If the evaluated atoms have the 'updated' flag activated, then: (i) the first kernel reads the state of the neighbor atoms, calculates $n^{1s}$ and $n^{1b}$, and stores them into *N1sb*; and (ii) the second kernel reads the *N1sb* variable for the current atom and the *N1sb* variables of the neighbors, calculates $n^{2s}$ and $n^{2b}$ using Equation 2, reads the new removal rate $R(n^{1s}, n^{1b}, n^{2s}, n^{2b})$ from the corresponding element of *TRate* and stores it in *Erate*. The two kernels are used to reduce the number of memory accesses, as described in the context of Equation 2. The first kernel updates the neighborhood and the second the removal rate.

Since each surface atom is processed by one thread at a time, the global memory access can be easily tuned to obtain coalesced read/writes, *e.g.* by enforcing contiguous threads to process contiguous atoms. However, the data reads for the neighbor atoms are harder to optimize. This is mainly due to two factors: (i) the set of atoms contained in the supercells (stored as MCs) varies depending on the simulated crystallographic orientation (Section 2.1), and (ii) the locations of the MCs where the neighbors are stored are not known *a priori* but, rather, are decided during execution, as the MCs are constantly freed and assigned in the GPU, depending on the actual shape and evolution of the surface itself. For atoms located at the supercell boundaries, it is rather likely that their nearest neighbors (in 3D space) will be stored at distant MCs (in the GPU). However, for the atoms located in the supercell interior the neighbors are located in the same MC. In this context, we choose to perform texture fetches of the *Syb*, *N1sb* and *TArea* variables in order to improve the overall read performance over that of raw uncoalesced access to the global memory.

Part b) of Figure 5 has the purpose of updating the occupation of the surface sites and monitoring which atoms reach zero occupation. The removal of these atoms provides the atomistic origin of the macroscopic propagation of the surface. In addition, the MCs whose atom occupations have been depleted to zero are flagged for later re-use and/or deallocation from the GPU memory. Steps (1)-(3) follow a computing procedure similar to that of part a). For this case, each thread checks if the processed atom is a surface atom by reading the *Syb* value. If so, Equation 1 is applied to reduce the occupation value (stored in *Occ*) using the value of the etch rate (stored in *Erate*). If the occupation is totally depleted, then: (i) the 'modified' flag (stored in *Syb*) is activated for the direct and indirect nearest and next-nearest neighbors of the current atom, (ii) the state of the direct neighbors is modified from 'bulk' to 'surface', and (iii) the *MC-state* counter is decreased by one unit. This counter monitors the amount of remaining unetched atoms. When $MC\,state = 0$, the whole MC has been depleted from its atom contents and can thus be deallocated or re-used to store another supercell.

At first glance, write access to the previous flags will lead to an expensive uncoalesced memory access pattern. However, the action of setting the flags is performed only for a small fraction of the large amount of processed occupation reductions, thus resulting in low intensity memory access. On the other hand, some of the neighbor atoms are also removed in the same time step, thus requiring repeated flag writes for those neighbors shared by nearby surface atoms. For this reason, we use the shared memory as a buffer to pre-store the 'modified' flags, thus preventing multiple global activation of the same atom flag and reducing global memory accesses.

Finally, part c) of Figure 5 performs the required computations related to the surface management. Since the neighborhood state is used by the surface atoms in order to determine their removal rates, the atoms located at the supercell boundaries must be able to access their neighbors. Thus, the program must ensure that a supercell containing at least one surface atom can easily access the front, back, east, west, north and south neighbor supercells. Since each supercell is stored in a MC, access to the neighbor supercells is provided by always ensuring that the corresponding MCs have been allocated. For this purpose, the following three different states are defined for the MCs:

- There is no need to perform any action (00).
- For the first time a bulk atom has become a surface atom in this supercell. Etching has started in the current supercell and the neighbor supercells must be allocated (01).
- All the atoms in the current supercell have been completely etched away and the neighbor supercells do not have any surface atoms. The current supercell can be removed (10).

When a change is detected (cases 01 and 10), the corresponding thread activates the MC flag, which is stored in *TBuf*. The CPU will analyze *TBuf,* perform the corresponding operations

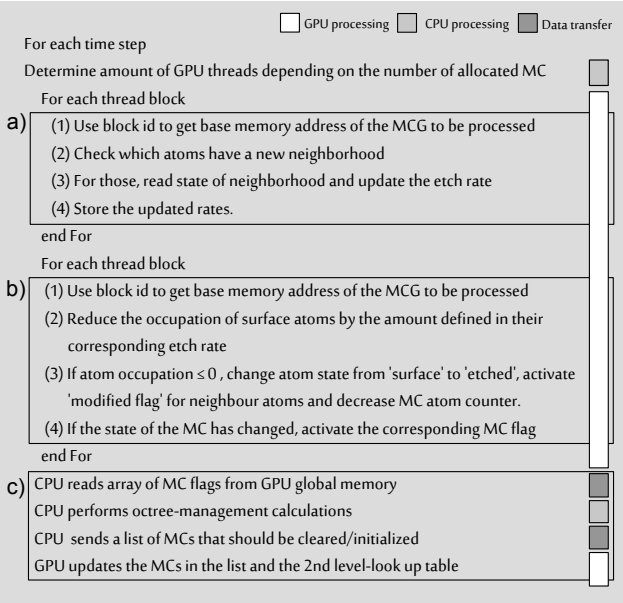


Figure 5: Pseudocode for the parallel implementation (wet etching). (a) Neighbor and removal rate updates. (b) Occupation reduction and atom removal. (c) Structure management.

for the octree management and inform the GPU about (i) the required initialization of certain MCs, and (ii) updates to the look-up tables. This process requires that a total information of two bits per MC must be sent from the GPU to the CPU in each time step. Although this transfer rate may seem small, the total amount of transferred data depends directly on the size of the MC (= supercell). Thus, the MC size should be a trade-off between (i) reducing the number of unnecessary processed atoms, which is obtained by using smaller supercells, and (ii) preventing excessive GPU-CPU data transfers and CPU octree-management calculations, which is obtained by using larger supercells. The effect of the supercell size on the efficiency of the overall implementation is analyzed in detail in Section 3.6.

The ultimate purpose of part c) of Figure 5 is to update the $2^{nd}$ level look-up table *TArea* stored in the GPU and to initialize recently assigned MCs. The GPU kernel that performs this function receives through *TBuf* (i) a list of removed and allocated MCs and (ii) the new set of neighbor pointers for each updated entry . Threads executing this kernel read data from this list and update the corresponding locations in the look-up table, one entry per thread. In addition, if the MC related to the look-up table entry is meant to be initialized, the thread itself will also perform this action, setting the type of all the atoms in the supercell to 'bulk' and their occupation to '1'. The memory access pattern for updating the look-up table and initializing the MCs is not specially optimized since the number of updated MCs per time step is always small and, thus, mostly irrelevant.

### *3.5. Description of the computational tests*

he main objectives of the tests performed in this work are: (i) to study the impact of the supercell size (= MC size) on the overall simulation time and GPU memory occupation, (ii) to describe the scalability of the proposed implementation as a function of the size of the simulated systems, and (iii) to present typical execution times for a wide range of surface dimensions. For comparison of the computational results to the experiments, 2000 time steps of chemical etching are simulated on a <100>-oriented silicon wafer masked using a square pattern that partially covers 45% of the surface in an aqueous solution of Potassium Hydroxide (KOH) as the etchant at a concentration of 40 wt% and a temperature of 70 C. The atomistic etch rates used by the CCA method are available in Ref. [16]. In this experiment, the silicon regions under the square convex corners of the masking pattern are etched away –as shown in Figure 6– due to the development of fast etching planes. When observed from the top, the shape of the etch front differs from that of the original mask. This phenomenon is known as *convex corner undercutting*, also referred to as *underetching*.

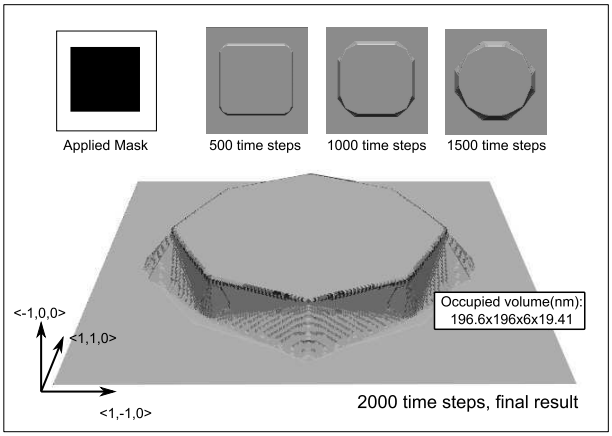


Figure 6: Simulated structure for the study of the efficiency dependence on the supercell size. 2000 time steps applied. Model calibrated for simulating wet etching in 40wt% KOH at 70C.

In order to study the impact of the supercell size on the computational time and memory (objective (i)), we use the fact that the smallest space partition at the leaf level of the octree corresponds to an orthorhombic unit cell of the silicon crystal. By choosing multiples of this minimum unit cell we can generate increasingly larger supercells and thus analyze the impact that the allocated MC size has on the overall GPU memory occupation and computational speed. Relevant parameters, such as the execution time, GPU main memory use, total data transfer between CPU and GPU, and the amount of intructions performed by the GPU are analyzed.

In order to determine the scalability of the proposed implementation (objective (ii)), surface sizes in the range of $2^{16} \approx 6.6 \times 10^4$ up to $2^{24} \approx 1.7 \times 10^7$ silicon atoms are simulated, doubling the size on every new simulation. In order to keep the aspect ratio of the resulting three-dimensional shape, the etched depth needs to be increased by a factor of $\sqrt{2}$ per surface doubling. This results in a total increase of the occupied volume of the system by a factor of $\sqrt{2} \times \sqrt{2} \times \sqrt{2} = 2\sqrt{2} = 2.83$ per surface doubling. As shown in Section 3.6, this volume is directly related to the number of computations required to reach the same final state for the larger of every two consecutive systems.

In addition, we also present typical execution times for the proposed parallel implementation of the CCA model applied to a total of six different systems (objective (iii)), and compare the results to an already existing, fully sequential implementation of the same model, known as VisualTAPAS [25]. The latter is a well-known etching simulator using the same theoretical model but implemented over a CPU-only environment without any kind of parallelization. The six different systems, etched in 30 wt% KOH at 80 C, are described and simulated with VisualTAPAS in Ref. [16]. Each system contains a different mask pattern whose features have different aspect ratios and are orientated along different crystallographic directions. This serves to check the correctness of our GPU-based algorithm and test whether substantial performance improvements can be obtained independently of the resulting shapes of the surfaces.

The hardware used for the simulations consists on a Nvidia 9800GT graphics card with 512MB of memory as GPU and an Intel Core i7 at 2.66GHz with 3 GB of PC1333 DDR3 SDRAM as CPU.

### *3.6. Results*

The data in Table 2 shows the improvement in several aspects of the GPU performance when the supercell size is reduced. This characterizes the effect of splitting the GPU memory into gradually smaller MCs. Reducing the size from 4096 atoms (last row) leads to a strong reduction in the amount of occupied GPU memory (column 7). As the supercell size is reduced the active space region can be defined more accurately by the octree, thus avoiding the storage of many bulk, useless atoms far from the interface. This effect is also present in the number of performed instructions by the GPU (column 5). On the other hand, the supercell size does not affect significantly the performance of the GPU algorithm, keeping a constant instruction throughput, namely, around 0.5 Instructions Per Cycle (IPC, column 6). Reducing the MC size (column 2) while maintaining the algorithm efficiency (column 6) leads to a reduction in the overall simulation time (column 3). According to the timing results, the optimal supercell size for the implemented model contains 64 atoms. Reducing the supercell size further makes the simulations slower (column 3). This is due to three effects: (1) the reduction in the amount of processed bulk and surface atoms with decreasing supercell size becomes less significant; (2) the octree complexity and the amount of octree modifications become large enough to make the amount of CPU processing relevant, as shown in Figure 7 (a); and, (3) very small supercells require increased access to the look-up tables stored in the GPU main memory. For supercell sizes of 4 and 8 atoms, the third effect is clearly observed in Table 2 as an increase in the number of instructions performed by the GPU(column 4) and a reduction of the instruction throughput (column 5).

The scalability of the presented implementation is presented in Figure 7(b). As explained in Section 3.5, doubling the size of each simulated surface represents a theoretical instruction increase of $2\sqrt{2}$. The increase factor in computational time per surface doubling ($T_j/T_{j-1}$) is shown using diamond markers. These values are obtained by dividing the computational time obtained for one size (*e.g.* $j = 7$ for $4096 \times 1024$ atoms) by the computational time for the previous size (*e.g.* $j = 6$ for $2048 \times 1024$ atoms). These values should approach the theorical value ($2\sqrt{2}$). Lower values mean that the algorithm performs better when applied to the bigger surface. The reason for this is that a high amount of parallelism should be achieved in order to keep all the GPU cores working and thus obtain the optimal processing power. In our implementation, full use of the GPU capabilities is achieved for initial surfaces with $2048 \times 1024$ atoms or larger, where 131072 threads or more are launched simultaneously. The GPU is underused for smaller surfaces. Such a large amount of threads is required since only a small fraction of the visited atoms by all the threads will be removed in each time step. Faster removal rates for the atoms and/or bigger surfaces lead to larger amounts of atoms etched away per time step and, thus, a higher amount of threads with substantial computational work. Once this point is reached (e.g. a surface bigger than $2048 \times 1024$ atoms), the increase in the processing time agrees well with the theorical increase in the number of calculations, this way demonstrating a good scalability of the proposed implementation.

Finally, Figure 8 demonstrates that, for a wide range of different mask designs, the etched structures obtained with the proposed GPU-based implementation of the CCA model are essentially identical to the ones obtained with a fully sequential implementation of the model, known as VisualTAPAS [16]. Furthermore, row 5 of the figure demonstrates that the GPU-based algorithm is at least two orders of magnitude faster than the purely sequential code for a wide range of test cases. The differences in the parallel-to-sequential speedup factors for the different mask patterns are due to the differences in the average number of atom removals per time step. Small masked areas lead to more atom removals, resulting in a more parallel behavior and a better adaptation to the GPU (as described in the previous paragraph). As a matter of fact, a comparison of rows 5 and 6 of Figure 8 shows that the speedup factor is correlated to the percentage of initial white pixels in each mask pattern. The relative amount of white area directly determines the percentage of the initial area attacked by the etchant. Based on this correlation, the speedup factor for column 3 of figure 8 seems significantly larger than expected. This is explained by the fact that this particular system presents a large amount of underetching (*i.e.* the removal of material below the original masking features) thus leading to a larger percentage of attacked area than the initial white-pixel area estimate. Correspondingly, this leads to an improved performance by the parallel algorithm on this system.

### *3.7. Practical application*

Finally, as a demonstration of the utility of the new implementation in practical simulations, a complex etching simulation for the manufacture of a triple-axis micro-accelerometer is shown in Figure 9. This accelerometer consists of three seismic masses suspended by high aspect ratio beams acting as springs, thus enabling the sensing of accelerations in the three perpendicular directions [27]. The process uses two steps of anisotropic etching in 40 wt% KOH at 70 C, each performed

Table 2: Performance of proposed GPU-based implementation *vs* supercell size. Each simulation consisted on 2000 time steps performed on a Nvidia 9800GT graphics card with 512 MB of memory.

| Supercell size (Å Å Å) | Si atoms in supercell | Execution time (sec) | GPU-CPU total transfered data (MB) | GPU performed instructions | GPU instruction throughput (IPC) | GPU global memory usage (%) |
|---|---|---|---|---|---|---|
| (3.84 3.84 5.43) | 4 | 30.27 | 572.76 | $2.370 \times 10^9$ | 0.36 | 18.51 |
| (7.68 3.84 5.43) | 8 | 22.60 | 296.67 | $2.159 \times 10^9$ | 0.36 | 15.49 |
| (7.68 7.68 5.43) | 16 | 15.82 | 153.13 | $1.938 \times 10^9$ | 0.43 | 14.18 |
| (7.68 7.68 10.86) | 32 | 15.19 | 121.96 | $2.634 \times 10^9$ | 0.55 | 22.85 |
| (15.36 7.68 10.86) | 64 | 14.71 | 63.65 | $2.549 \times 10^9$ | 0.52 | 23.54 |
| (15.36 15.36 10.86) | 128 | 14.82 | 33.09 | $2.564 \times 10^9$ | 0.54 | 24.64 |
| (15.36 15.36 21.72) | 256 | 18.34 | 27.84 | $3.867 \times 10^9$ | 0.61 | 42.25 |
| (15.36 30.72 21.72) | 512 | 21.48 | 14.75 | $3.888 \times 10^9$ | 0.52 | 45.27 |
| (30.72 30.72 21.72) | 1024 | 23.19 | 7.57 | $3.827 \times 10^9$ | 0.48 | 47.94 |
| (30.72 30.72 43.44) | 2048 | 30.87 | 6.59 | $6.196 \times 10^9$ | 0.57 | 84.25 |
| (30.72 61.44 43.44) | 4096 | 37.38 | 3.45 | $6.701 \times 10^9$ | 0.51 | 91.33 |

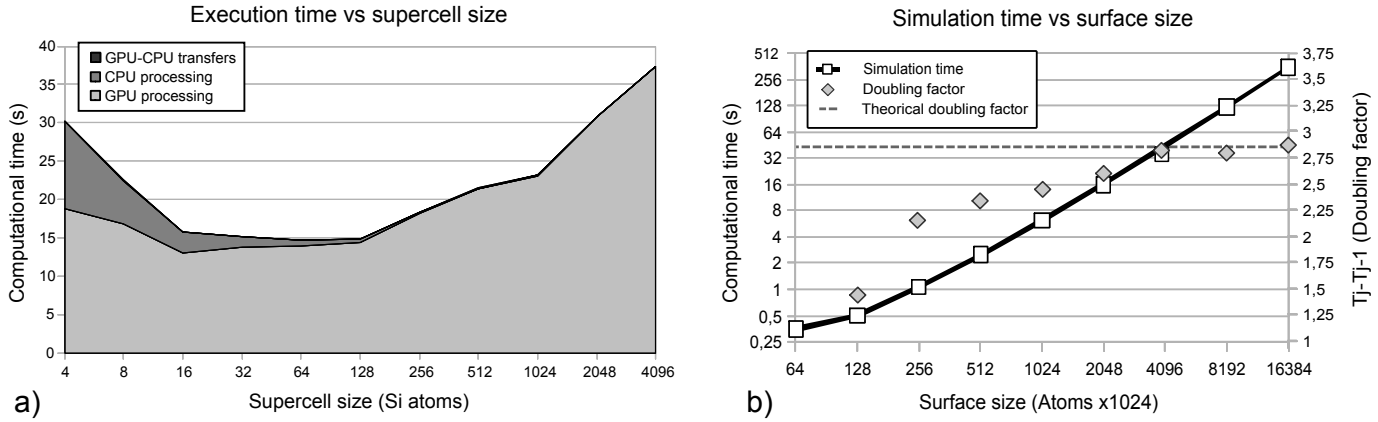


Figure 7: (a) Contributions to the overall computational time, including execution of various algorithms by the GPU (GPU processing), octree management by the CPU (CPU processing) and GPU-CPU data transfers, for a wide range of supercell sizes. (b) Computational time increase factor per surface doubling ($T_j/T_{j-1}$).

using a different mask pattern that is applied to both the top and bottom surfaces of the silicon wafer. The GPU implementation not only enables accurate simulations in faster times, but also makes possible the simulation of significantly larger systems (containing more surface atoms). As compared to the CPU platform, this can be used to significantly increase the resolution of a simulation while keeping the system size fixed, thus offering greater detail in the different features of the front, as shown in Figure 7(b) for the corner regions. It is important to note that these simulations are performed in the GPU environment in just a few seconds instead of minutes when using a traditional CPU. This is specially important since the number of engineering design iterations of a structure can be relatively large before a satisfactory answer is obtained.

## 4. Discussion

Specialized parallel hardware, such as that of a GPU, has been continuously introduced in recent years for the simulation of a wide range modeling methods. Relevant examples of this fact are parallel implementations of Level Set Method [28], Finite-Difference Time-Domain (FDTD) method and the Finite Element Method (FEM) [29, 30], Cellular Automata [17], Lagrangian Models [31], Monte Carlo simulations [32] and Mollecular Dynamics [33]. Although in these examples the speed of the simulations is improved by adapting the algorithms to the new processors, adapting also the supporting data structures, such as the octree, is an essential part of the optimization effort. Our results strongly indicate that the performance and efficiency obtained with parallel processors can be increased even further by performing this type of data structure modification.

Current level set models utilize the narrow band and sparse field methods to avoid unnecessary processing of non-useful space regions. The GPU implementations have mostly focused on the narrow band method, partly due to the fact that the sparse field approach relies heavily on the use of strongly-sequential, hard-to-paralellize linked-lists in order to keep in memory the active voxels during the calculation. As an example, a recent GPU implementation of the narrow band has been proposed, based on a list of active tiles, maintained in the GPU by extensively using atomic instructions [35]. Although this significantly improves the performance as compared to purely sequential CPU implementations, the use of the atomic operations limits unnecessarily the overall computational throughput. An approach closer to our work was presented by Lefhon *et al.* [34]. This approach gives a solution specifically oriented for level-set algorithms, dividing the GPU memory in two-dimensional pages and using the CPU to manage the allocation/removal of these pages in order to reduce the computational domain.

Roberts *et al.* have recently presented a fast, work-

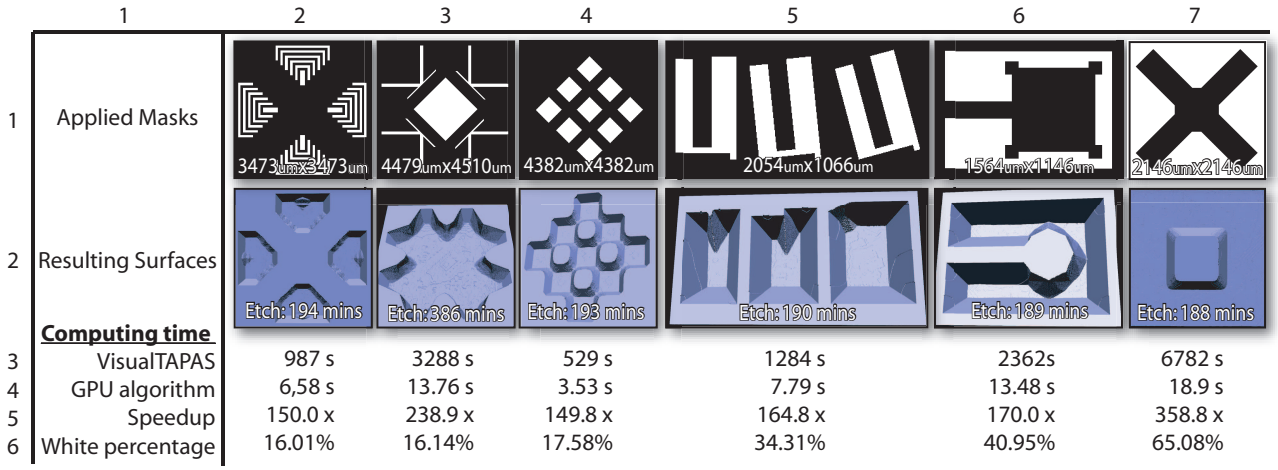


Figure 8: Performance comparison between VisualTAPAS and the proposed GPU-based algorithm for different surfaces. Mask designs after Ref. [26].

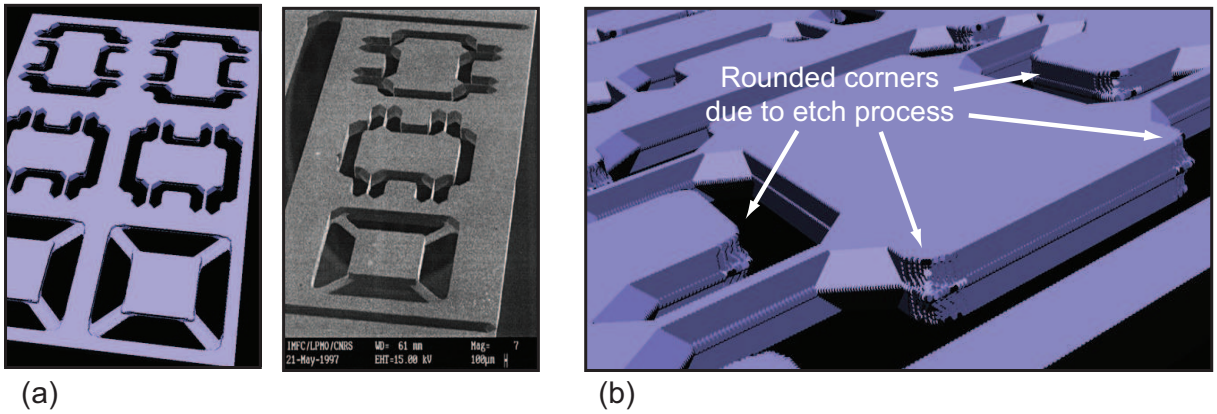


Figure 9: (a) Simulation (left) and experimental results of the fabrication of a 3-axis accelerometer. (b) Close view of the simulated surface. Experimental image from Ref. [27].

efficient, race-condition-free, GPU implementation of the level set method for volume segmentation applications, completely avoiding atomic operations and fully independent from any CPU processing [36]. This method makes two major contributions, namely, (i) reducing the active computational domain to a minimal set of changing voxels based on examining both the temporal and spatial derivatives of the level set function, and (ii) implementing a GPU-efficient procedure to update in parallel a large number of elements of a 1D dense array that stores the currently active voxels and serves as an alternative to the linked-list data structure typically used in the sparse field method. Although the improved overall efficiency justifies the large bookkeeping effort (to maintain the 1D dense array), the method makes excessive use of the GPU memory, with up to three versions of the level set function entirely kept in the GPU memory for the whole 3D computational grid and, seven versions of the 1D array (required to update in parallel the dense, active-voxel list). In this respect, Roberts *et al.*'s implementation could benefit from the use of an octree data structure, as presented in this study, in order to effectively reduce the large memory consumption associated to maintaining the whole computational grid. In fact, it has already been shown that the octree implementation for level sets is a viable alternative to reduce the memory requirements and speed up the calculations, comparable to the narrow band and sparse field methods in single CPU environments [12, 13]. The present study strongly indicates that not only the GPU memory use will be reduced but also the overall computational speed will probably be increased by optimizing the size of the supercells associated with the octree leaves (in this case containing grid points).

While the procedure of Ref. [36] to determine the minimal set of active elements is irrelevant for typical CCA simulations (as monitoring the active atoms is already a fundamental feature of the underlying method), the use of a similar 1D dense array to tightly store the active atoms is likely to improve further our results, as explained in this paragraph. According to our experience, the most efficient implementation of a dynamic surface that is modeled using a CA over a three-dimensional (3D) array in a CPU-only environment is obtained by using a complementary linked list of surface cells. Without the list, typical searches for new events –or for updating the states of neighboring cells– are made inside a volume (3D array) while the same searches are effectively performed inside a 1D object with far less elements when the linked list is used. Although

the use of an octree still allows the use of such a list, accessing the leaves of the octree has a larger computational cost than the direct access of an equivalent 3D array. As a result, the combination of the octree and the linked list results in lower performance than the combination of a 3D array and the linked list. Typically, however, the reduction in the memory requirements in our applications justifies the use of the octree. If the linked list is not used, the overall performance becomes significantly better for the octree-based system, since the number of searches at the leaf level of the octree remains much smaller than inside the 3D array. In a parallel computing environment where the computation freedom is reduced, such as for a GPU, the use of a surface linked list is not traditionally suited to the parallel computing methodology: the list management, like the octree, can suffer race conditions (*e.g.* several threads adding a new atom at the end of the list) and thread exclusion is neccesary. As a result, in our GPU implementation without linked lists the octree not only uses less memory than a 3D array but also increases the computational speed. The method presented by Roberts *et al.* to track active points in level-set simulations using the GPU environment [36] could provide a valuable approach for implementing a surface atom list in combination to our octree management, possibly improving even more the efficiency of the calculations while keeping the current reduced level of memory usage.

The computational times and speedup values presented in figures 7 and 8 depend on the particular details of the GPU card architecture used for the simulations. In this study we have chosen a mid-price graphics card whose memory and computational power specifications fell grossly in the middle of the range at the time of acquisition. If superior (inferior) GPU card models are used, larger (smaller) speedups will be obtained. The results presented in this study give a flavor of the significant performance boost that can be obtained by using this type of affordable many-core architecture and the proposed adaptation of the octree data structure.

The term Many Core Architecture (MCA) has been recently coined by consumer-driven, computer companies, such as Nvidia and Intel, in order to refer to processors composed of arrays with many computational units (or cores) and various memory hierarchies (shared or not). Although Graphical Processor Units (GPUs) are at this moment the most extended implementation of this architecture, new processors conceived for graphics-independent, massively parallel processing are gradually entering the market and are expected to become popular in the close future. As an example, the Nvidia Tesla c1060 is a many-core processor exclusively dedicated to algorithm processing, although the inner architecture is similar to that of modern GPUs. This particular card has been classified as an MCA because it cannot be directly used for graphical representations. From this perspective, every reference to the term 'GPU' in the present study can be understood as a reference to the more general class of 'MCA' processors.

During the different stages of this study we performed various modifications of the GPU wet etching algorithms in an effort to optimize several aspects of the GPU global memory management. This includes memory coalescing, the use of shared memory for reutilizing data, the storage of data in different ways to improve texture fetching performance, etc. Although the parallel algorithms can be optimized in many ways, the particular details and modifications do not affect the CPU-based octree implementation, which is very robust and was kept unchanged. Based on the results shown on Table 2 and Figure 7, the present study strongly indicates that the proposed dynamic structure management using an octree is an effective procedure for increasing the efficiency of parallel computations that are focused on evolving surfaces.

## 5. Conclusions

The availability and constant development of new Many core Architectures (MCA) in recent years enables the exploration of novel implementation approaches for existing problems. As an affordable example of the new architectures, Graphics Processing Units (GPUs) stand out as an attractive option for various scientific applications. In this study, we consider the simulation of evolving dynamic surfaces that can be efficiently stored using octree data structures and accurately propagated in time using intrinsically parallel methods, such as cellular automata. An octree subdivides the topologically-cubic computational domain into increasingly smaller octants, stopping the subdivision at the octants that do not contain any surface points, thus resulting in a detailed partition of the space nearby the surface while the bulk regions at both sides of the interface are described by empty octants whose size becomes gradually larger as one moves away from the surface. The main focus of the study is on the determination of an optimized implementation for the octree management in a parallel computing environment.

We conclude that the dynamic surface management and the model calculations must be decoupled as follows, by: (i) storing all the surface data and performing all the model calculations in the GPU, and (ii) storing the underlying octree structure and performing all the octree management tasks in the CPU. For this purpose, the GPU memory is considered as an array of equally sized Memory Clusters (MCs), and each MC is used to store one supercell, *i.e.* the minimum space subdivision of the underlying crystal or computational grid. Conversation between the GPU and the CPU is obtained by regarding the CPU octree leaf nodes as pointers to the GPU memory clusters. In the event that future GPU generations are able to efficiently manage an octree structure, the programer can port the fragment of code from the CPU to the GPU, leaving the rest of the program unchanged. In this case, our proposed memory management algorithm for a dynamic surface will still be valid.

The particular size of the MCs is chosen according to: (a) the particular features of the underlying physical model –*e.g.* in our case each MC must contain a crystallographic unit cell, or a multiple of it– and (b) a trade-off between (b.1) reducing the number of unnecessary processed bulk points by reducing the size of the supercell, and (b.2) preventing excessive data transfers between the GPU and the CPU, as well as excessive octree-management calculations in the CPU by increasing the size of the supercell.

In order to improve the parallel processing efficiency, the data in the GPU memory should be stored as contiguously as possible (proximity in memory space). This allows grouping the GPU memory clusters so that each group can be efficiently processed by the same thread block during execution. This leads to dispersion of the data in real space –with neighboring supercells being stored in distant GPU clusters. As a result, the use of a constantly updated, GPU-stored, look-up table that links the real positions and the memory locations is recommended. This minimizes the impact of multiple access to the neighbor points required by the underlying physical model. In addition, a stack data structure must be used as a repository of free GPU memory locations so that the CPU can assign them to new space regions during the creation of new octree nodes.

Considering wet etching of silicon as a computationally challenging process of wide interest in microengineering applications, we tested the efficiency of the previous guidelines for the propagation of the etch front using a Graphics Processing Unit (GPU) as a cost-efficient parallel architecture. The results show that the decoupling of the model calculations and the surface management can significantly improve the performance of parallel algorithms for the simulation of dynamic surfaces. In addition, the proposed approach shows good scalability, making the simulation of complex surfaces affordable. As a result of this study, the GPU implementation not only enables the realization of accurate anisotropic wet etching simulations in faster times but also makes possible the simulation of the same systems with significantly larger resolutions.

Although the proposed GPU-CPU implementation has been tested only on Nvidia GPUs in this study, the main guidelines in Section 2 are very general and should be valid for other MCAs. The most important feature that makes the presented approach useful is the lack of an efficient global thread exclusion mechanism. GPUs are a clear example, because the exclusion mechanism is based on atomic global memory access, which is very slow and contrary to the GPU programming philosophy. We strongly believe that other parallel hardware architectures can also benefit from the presented approach.

Similarly, we believe that the implementation aspects are not limited to cellular automata models and are probably valid for other computational techniques, such as the finite element, finite difference or level-set methods. As an example, state-of-the-art level-set simulations using 3D arrays to store the complete computational domain in the GPU memory are likely to benefit substantially from the use of octree data structures as a replacement for the 3D arrays. In this context, CPU-GPU management procedures similar to the one presented in this study can probably reduce significantly the amount of GPU memory used in the simulations and markedly improve the overall computational speed.

## Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions. This work has been supported by Programa de Becas de Excelencia de la Universidad Politécnica de Valencia (PAID-09-09), MEXT Grant in Aid Research (Kakenhi: Silicon etching (A) 19201026), and the Global COE program of Japan (GCOE, Wakate JSPS Young Scientist Fund).

## References

[1] K. Yee, IEEE Trans. Antenn. Propag. 14 (1966) 302.
[2] J. D. Anderson, Computational Fluid Dynamics : The Basics with Applications (McGraw Hill, 1995)
[3] T. Suntola, Handbook of Crystal Growth 3, Thin Films and Epitaxy, Part B: Growth Mechanisms and Dynamics Ch. 14 (Elsevier Science Publishers B.V., 1994)
[4] R. L. Puurunen, J. Appl. Phys. 97 (2005) 121.
[5] R. A. Wind, M. A. Hines, Surf. Sci. 460 (2000) 21.
[6] T. J. Hubbard, E. Antonsson, Sensor Mater. 9 (1997) 437.
[7] Z. Zhou, Q. Huang, W. Li, C. Zhu, J. Phys.: Conf. Ser. 34 (2006) 674.
[8] T. R. Halfhill, Microprocessor Report Newsletter (Published: 2008/01/28) (www.mdronline.com/mpr/mpr_index.html, visited 14/12/2009).
[9] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips, Proc. of IEEE 96 (2008) 879.
[10] D. Libes, Comput. Graph. Mag. 15 (1991) 383.
[11] M. A. Gosálvez, Y. Xing, K. Sato, R. M. Nieminen, J. Micromec. Microeng. 18 (2008) 055029.
[12] J. Strain, J. of Comp. Phys. 151 (1999) 616.
[13] F. Losasso, F. Gibou, R. Fedkiw, ACM Trans. Graph., 23 (2004) 457.
[14] Y. Xing, M. A. Gosálvez, K. Sato, New J. Phys. 9 (2007) 436.
[15] M. A. Gosálvez, Y. Xing, K. Sato, J. microelectromech. syst. 17 (2008) 410.
[16] M. A. Gosálvez, Y. Xing, K. Sato, R. M. Nieminen, Sensor. Actuat. A 155 (2009) 98.
[17] S. Gobron, F. Devillard, B. Heit, Mach. Vis. Appl. 18 (2007) 331.
[18] D. Sholl, J. A. Steckel, Wiley-Interscience, 2009.
[19] M. A. Gosálvez, K. Sato, A. S. Foster, R. M. Nieminen, H. Tanaka, J. Micromech. Microeng. 17 (2007) S1.
[20] C. Merveille, Sensor. Actuat. A 60 (1997) 244.
[21] L. Qiu, S. Hein, E. Obermeier, A. Schubert, Sensor Actuat. A 54 (1996) 547.
[22] J. D. Martinez, P. Blondy, A. Pothier, D. Bouyge, A. Crunteanu, M. Chatras, Microwave Conf., 2007. European, (2007) 439.
[23] Nvidia CUDA home page, http://nvidia.com/cuda.
[24] S. Ryoo, C. L. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. W. Hwu, Proc. of PPoPP (2008) 73.
[25] VisualTAPAS simulator home page, http://tfy.tkk.fi/˜mag/VisualTAPAS/Home.html.
[26] SIMODE collection of examples (c) 2001, gesselschaft fur mikroelektronikanwendung chemnitz mbh.
[27] G. Schröpfer, M. de Labachelerie, S. Ballandras, P. Blind, J. Micromech. Microeng. 8 (1998) 77.
[28] J. E. Cates, A. E. Lefohn, R. T. Whitaker, Med. Im. Anal. 8 (2004) 217.
[29] P. Sypek, A. Dziekonski, M. Mrozowski, Magnetics, IEEE Trans. 45 (2009) 1324.
[30] D. Komatitsch, D. Micha, G. Erlebacher, J. Parallel Dist. Comp. 69 (2009) 451.
[31] F. M. Jr., T. Szakaly, R. Meszaros, I. Lagzi, Comput. Phys. Commun. 181 (2009) 105.
[32] P. Martinsen, J. Blaschke, R. Knnemeyer, R. Jordan, Comput. Phys. Commun. 179 (2009) 1983.
[33] W. Liu, B. Schmidt, G. Voss, W. Mller-Wittig, Comput. Phys. Commun. 179 (2008) 634.
[34] A. E. Lefohn ,J. M. Kniss ,C. D. Hansen ,R. T. Whitaker , IEEE Trans. on Vis. Comp. Graph. 10 (2004) 422.
[35] W. K. Jeong, J. Bayer, M. Hadwiger, A. Vazquez, H. Pfister, R. T. Whitaker, IEEE Trans. Vis. Comput. Graph. 15 (2009) 1505.
[36] M. Roberts, M. C. Sousa, J. R. Mitchell, Proc. of High Perf. Graphics (2010) 123