An Improved Parallel Implementation of 3D DRIE Simulation on GPU

Fan Zhang, Gang Wang, Xiaoguang Liu, Jing Liu
Nankai-Baidu joint lab, College of Information Technical Science, Nankai University
Weijin Road 94, Tianjin, 300071, China
Email: zhangfan555@gmail.com, wgzwp@163.com, liuxg74@yahoo.com, jingliu@mail.nankai.edu.cn

***Abstract*—Deep reactive ion etching (DRIE) technique is a new and powerful tool in Micro-Electro-Mechanical Systems (MEMS) fabrication. A 3D DRIE simulation can help researcher understand the time-evolution of Bosch process used in DRIE. Due to the high complexity of the algorithm used in the simulation, it is necessary to develop an algorithm that can accelerate the simulation. This paper presents a parallel implementation of the 3D DRIE simulation based on GPU, built on Nvidia's Compute Unified Device Architecture (CUDA) platform. This paper also presents a fast morphological operation, which reduces the complexity of mathematical morphology operation part of the algorithm from $O(N^3)$ to $O(N^2)$. The experiment results show the parallel program on Nvidia GTX260+ GPU obtains about 70x to 75x speedup over the 4-threads parallel version on Intel Q6600 CPU.**

I. Introduction

In Micro-Electro-Mechanical Systems (MEMS) fabrication, Deep Reactive Ion Etching (DRIE) is a new and powerful tool for etching very deep trenches with nearly vertical sidewalls. The most popular silicon DRIE technique is Bosch process patented by Robert Bosch GmbH [1] in which etch and polymerization cycles alternate in an induced coupled plasma reactive ion etcher (ICP-RIE) system. Since the procedure of Bosch process is more complex than other etchings', an accurate and fast simulator is necessary to help researchers understand the time-evolution of the topography. Guangyi Sun et al. [2] has proposed a visual 3D simulation for DRIE process. But the simulation is very slow running on CPU, it is necessary to develop a parallel version which can accelerate the simulation progress using the novel GPU technology.

We make the following contributions in this paper:
1) To the best of our knowledge, there is no 3D DRIE simulation implemented on GPU, it is the first work which implemented 3D DRIE simulation on GPU.
2) We proposed the fast morphological operation to reduce the time of MO phase of the simulation.

The rest of this paper is organized as follows. Section 2 presents an overview of GPU architecture and the programming environment on which our parallel program is built. Section 3 introduces the serial algorithm used in 3D DRIE simulation. Section 4 gives a rough explanation of the fast morphological operation. Section 5 explains our implementation on the GPU. We show our results in Section 6 and conclusion in Section 7.

II. GPU Programming Architecture

In this section we discuss major micro architectural features of the latest GeForce graphics processors and Nvidia's Compute Unified Device Architecture (CUDA). A more detailed description can be found in [6,7,9,10].

A. Hardware Model

At the hardware level, the GPU is a collection of multiprocessors, with several processing elements in each. For instance, the Nvidia GeForce GTX 280 has 30 multiprocessors with 8 scalar processor (SP) in each. The many-core feature makes GPU efficiently support a very large number of threads, so the efficiency reaches the peak when massive threads are running on GPU. Each processor in the multiprocessor executes the same instruction in every cycle. Each can operate on its own data, which makes each a SIMD processor. Next we describe the programming model we use in our application.

B. Programming Model

The programming model we use is the Nvidia's Compute Unified Device Architecture (CUDA). CUDA is a programming interface to use the parallel architecture of Nvidia GPUs for general purpose computing. CUDA produces a set of library functions as extensions of the C language. A compiler generates executable code for the CUDA device. The CPU sees a CUDA device as a multi core co-processor. All memory available on the device can be accessed using CUDA with no restrictions on its representation though the access times vary for different types of memory.

For the programmer, the CUDA consists of a collection of threads running in parallel. The programmer can select the number of threads to be executed. If the number of threads is more than the warp size (32), they are time-shared internally on the multiprocessor. A collection of threads called block runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor for time-shared execution. They also divide the common resources like registers and shared memory equally among them. A single execution on a device generates a number of blocks. The collection of all blocks in a single execution is called a grid. Each thread and block is given a unique ID that can be accessed within the thread during its execution. All threads of the grid execute a single program called the kernel. We next describe the algorithm use in the 3D DRIE simulation.

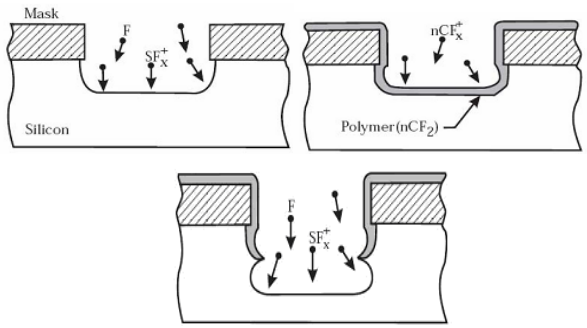

Fig. 1. Principle of DRIE

III. Algorithm of 3D DRIE simulation

A. Principle of the DRIE

The principle of the DRIE is schematically shown in Fig.1. The typical etch cycle lasts 5 to 15s, uses SF_6 to etch silicon. In the next cycle, a fluorocarbon polymer, about 10nm thick, is plasma deposited using C_4F_8 as a source gas. In the following etch cycle the energetic ions (SF_{x+}) remove the protective polymer at the bottom of trench, but the film remains relatively intact along the sidewalls. The repetitive alternation of etch and passivation steps results in high directional etch at rates between 1.5 and 4um/min, high aspect ratio up to 30:1, high selectivity to mask (75:1 to photoresist).

B. Serial Algorithm of the 3D DRIE Simulation

According to the principle of DRIE, 3D DRIE simulation includes two parts, one is etching and the other is the polymerization. Because the etching part is more complex than the polymerization process, the method and optimizations mentioned below are all about etching process.

Guangyi's DRIE model [2] is based on voxel, the silicon is treated as a set of voxels, if the value is mapped into $\{0,1\}$: the voxels assigned 1 representing opaque objects and the voxels assigned 0 representing the transparent background.

At each step of the etching process, first find all the surface voxels, where the surface voxel is defined to be the voxel whose at least one of the six adjacent voxels is 0 (transparent background). Then the program will run a shadow test function to calculate the amount of rays each surface voxels receives under the mask, these rays make up a solid angle. Fig.2 gives the illustration of shadow test and solid angle. The etching rate of each surface voxels can be calculated from a serials of formulas given in [2,3] using the solid angle. After the rate has been calculated, erase the voxels within the sphere whose center is the surface voxel and the radius is the rate, this operation is called Mathematical Morphology Operation (MO) [2]. One shadow test operation and one MO operation is called "one etching step" in the 3D DRIE simulation which consists of several steps in order to etch very deep trenches. Algorithm 1 illustrates the serial algorithm of one single step

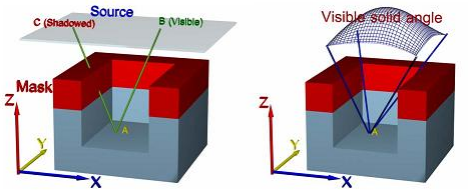

Fig. 2. Schematic illustration of shadow test and calculation of visible solid angle

of etching process. In the next section we will discuss the optimization of the MO phase.

Algorithm 1 Etch process of one step in 3D DRIE simulation

```
//Shadow test phase
for i := 1 to z step 1 do
    for j := 1 to x step 1 do
        for k := 1 to y step 1 do
            if the voxel[i,j,k] of the silicon is surface point
              then calculate the etch rate using shadow test
            fi
        end
    end
end
//MO phase
for i := 1 to z step 1 do
    for j := 1 to x step 1 do
        for k := 1 to y step 1 do
            if the voxel[i,j,k] of the silicon is surface point
              then erase the voxels within the sphere whose
                   center is voxel[i,j,k] and radius is the rate
                   calculated in the shadow test phase
            fi
        end
    end
end
```

IV. Fast Morphological Operation

We first make an improvement on the MO phase of Algorithm 1 which we call the new method fast morphological operation (FMO). Fig.3 gives the 2D schematic illustration. Different with the original morphological algorithm [3-5], it is not necessary to access every point inside the sphere when performing erasing operations since a large number of voxels overlap between two adjacent spheres. As shown in Fig.3, P_n and P_{n+1} are two adjacent morphological operation elements which are the spheres mentioned in Algorithm 1, the overlapped oblique line part only needs to be erased one time. This could be implemented as follows: when etching the sphere P_{n+1} only erase the different part of sphere P_{n+1} compared to sphere P_n.

In one step of the 3D DRIE simulation, most mathematical morphological operations take place on the same horizontal

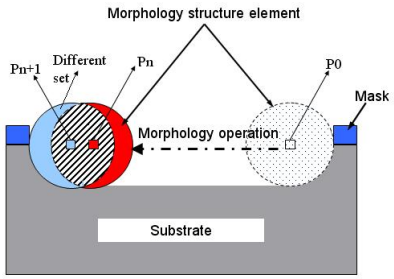

Fig. 3. Illustration of fast morphological operation

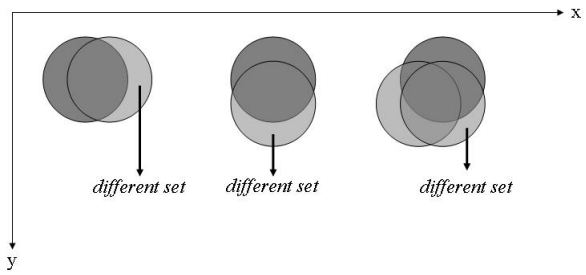

Fig. 4. three situations of fast operation

plane perpendicular to the z-axis, we only use the fast morphological operations in the following three situations shown in Fig.4. The left one, if the current point has the same etch rate as the left neighbor voxel's in x-axis, use the pre-calculated different set of that rate to erase the voxels. The middle one is same as the left one only it's along the y-axis. The right one will get more performance improvement. If the current voxels have the same rate as left and upper neighbor voxel's, only the corner different set will be used. The FMO roughly reduce the sphere erasing operation's complexity from $O(N^3)$ to $O(N^2)$.

V. GPU 3D DRIE Simulation using CUDA

A. Basic design considerations

The differences between CPU and GPU is the first design consideration: multi-core CPU only forks a few threads to get the job done, while GPU needs 1000s of threads for full efficiency. So if a task is massively parallel, it is well suitable for running on GPU. Another consideration is the global memory issue [6]. Global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp can be coalesced into a single memory transaction, where warp is a collection of threads that are scheduled for execution simultaneously on a multiprocessor. Threads in a half-warp must access the words in sequence: The k_{th} thread in the half-warp must access the k_{th} word, where word can be 32 bits, 64 bits or 128 bits. If threads not follow the above condition, the memory access is non-coalesced, and throughput is significantly reduced. Based on this point, the mapping technique for data partition in GPU programming is quite different from traditional multi-thread CPU programming, where CPU there can be block distribution, block-cyclic distribution etc. [14], these mapping techniques can not satisfy the memory coalescing requirement while on GPU assigning the continuous data to continuous threads is more suitable.

From the serial DRIE simulation algorithm, we can see that it is well adapted to run on GPU because the task is massively parallel. Because each voxel of the silicon is independent of each other, we could assign a single thread from the GPU to a voxel or more of the silicon. In other word, one thread charges of all the operations of a voxel in the volume. For the purpose of memory coalescing, continuous voxels are assigned to continuous threads, it's a typical one-dimensional cyclic distribution.

Another principle in designing the CUDA program is avoiding costly data transfer back and forth to host. The implementation uses the following kernels: Shadow test kernel, MO kernel, Polymerization kernel. After the volume data is transferred to GPU, host code invoke the shadow test kernel, the MO kernel and polymerization kernel iteratively several steps, at last transfer the volume data back to CPU for displaying, this scheme will minimize the transfer time between CPU and GPU.

B. Reduce thread divergence

To manage hundreds of threads running several different programs, the multiprocessor employs an architecture we call SIMT (single-instruction, multiple-thread). Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths. The highest performance is achieved when the threads avoid divergence and perform the same operation on their data elements.

Both the shadow test and MO kernel will detect the surface points first. If we choose every voxel of the volume as the input of kernel, and decide which voxel is the surface point for the subsequent computation, this scheme will lead to high thread divergence when etching deeply into the silicon and will lead to low performance. We need to distinguish surface points and non-surface voxels and put all those surface voxels into a new array for the next step of shadow test and MO kernel. This operation in GPU is called compaction [11]. However before it can be done, we must use a GPU building block called scan [12] to decide where each surface points should be put into the new array. In our CUDA program we use the scan operation implemented in Cudpp [13], where the algorithm used is called segmented scan [12]. The compaction operations

is in $O(N)$, there will be insignificant overhead compared to the performance gained. From the experiment result shown in Section 6, with surface points compaction, the CUDA program gets substantial speedup.

C. Other efficiency considerations

Constant Memory: Although the constant memory is very tiny, using it efficiently will get substantial performance improvement. For compute capability 1.0 [7], the constant memory is 64KB over all multiprocessors, we put the sphere data and difference sets of two adjacent spheres having same radius into the constant memory which is totally 56KB, less than the size limitation. With constant memory, the MO step gains about 2x speedup.

Threads per block: The number of threads of each block is also a factor that affects the performance, first choose threads per block as multiple of warp size (32) to avoid wasting computation on under-populated warps. Second, more threads per block will lead to better memory latency hiding, but more threads per block means that fewer register per thread, and in the experiment, we have encountered some situations that kernel invocations failed if too many registers are used. Based on experiment profiling, we always choose 256 or 128 threads per block.

Besides, template loop unrolling and CUDA built-in arithmetic operations will help CUDA program get good performance.

VI. Experiment Results

We use the following devices for testing: CPU: Intel Core 2 CPU Q6600 @ 2.40 GHz, 2GB RAM and GPU: NVIDIA GeForce GTX 260+, 768 MB global memory.

The version of CUDA is 2.2, GPU program compiler is nvcc and CPU program compiler is MS Visual Studio 2005. The masks we select are representative masks of 256 pixels width and 256 pixels length.

Fig.5 and Fig.6 give the run time of different methods, time for the GPU program does not include the data transferring time between CPU and GPU memory. The methods include:

trivial version of CUDA program on GPU(CUDA)
CUDA program with fast morphological operation(FMO) using in the MO kernel(CUDA/FMO)
CUDA program with surface volexs compaction (CUDA/Compaction)
CUDA program with fast morphological and compaction (CUDA/FMO and Compaction)
serial program with FMO on CPU(CPU)
4-threads parallel program with FMO on 4-Core CPU(4-Core CPU)

The program successively etches the silicon a reasonable number of steps, in this paper the number is 9. We can see that programs with FMO perform well on GPU. Another fact need to be noticed is the comparison between CUDA/FMO and CUDA/Compaction, for the first 2 steps, CUDA/FMO is better than CUDA/Compaction, but since the 3rd step

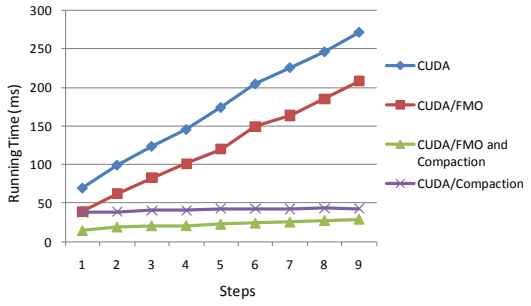

Fig. 5. Time of 9 steps with different methods - time of MO phase of each step

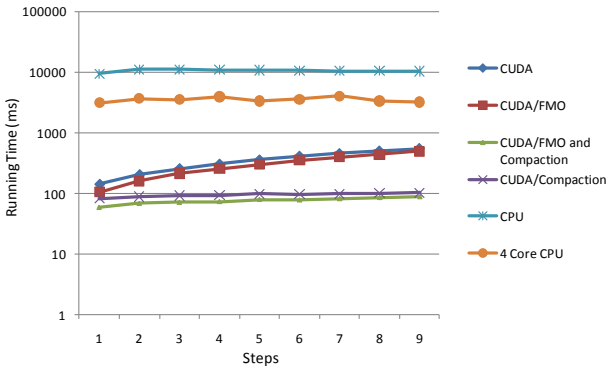

Fig. 6. Time of 9 steps with different methods - overall time of each step

CUDA/Compaction gets better. As etching deeply into the silicon, more volexs became surface volexs, more threads of FMO will become divergent threads, so Compaction method is better.

Fig.6 shows the overall time of each step including the shadow test, MO, scan and compaction kernel. Methods with FMO seems not to perform that well as shown in Fig.6 because shadow test kernel is the dominating part while MO kernel become insignificant. The time of CUDA and CUDA/FMO increases much as etching deeply into the silicon. Because, when etching deeply into the silicon, more voxels become the surface voxels, this will lead more threads to become divergent threads, and the performance will decrease. But compaction will efficiently reduce the thread divergence.

Fig.7 gives the comparison between GPU program and parallel CPU program, the GPU program includes the data transferring time between CPU and GPU memory, and the speedup is about 75x.

VII. Conclusion

In this paper we presents a parallel implementation of the 3D DRIE simulation based on GPU, built on Nvidia's CUDA platform. This paper also presents a fast morphological operation, which reduces the complexity of MO part of the algorithm from $O(N^3)$ to $O(N^2)$. The experiment results show the parallel version with fast morphological operation

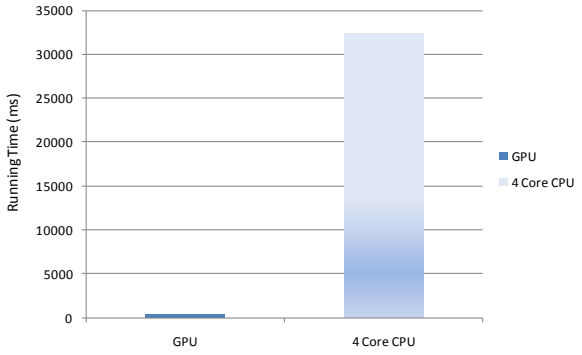

Fig. 7. Time of 9 steps with different methods - overall time of each step

and surface points compaction on Nvidia GTX260+ GPU obtains 70 to 75x speedup over the 4-threads parallel version on Intel Q6600 CPU. In the future work, we will try larger volume and use multi-GPU to do the job.

VIII. Acknowledgment

This paper is supported partly by the National High Technology Research and Development Program of China (2008AA01Z401,2009AA04Z320), NSFC of China (60674068,60903028), SRFDP of China (20070055054), and Science and Technology Development Plan of Tianjin (08JCYBJC13000,08JCZDJC22000), and we thank Guangyi Sun for providing the source code of 3D DRIE simulation.

References

[1] L. Franz and S. Andrea, "A Method of anisotropically etching silicon", *US Patent Specification 5501893, German Patent Specification DE4241045*.

[2] G. Sun, X. Zhao, H. Zhang, L .Wang, and G. Lu, "3-D Simulation of Bosch Process with Voxel-Based Method", *Proceedings of the 2nd IEEE International Conference on Nano/Micro Engineered and Molecular Systems*, Bangkok, Thailand,2007pp. 45-49, Jan 2007.

[3] R. Zhou, H. Zhang, Y. Hao, and Y. Wang, "Simulation of the bosch process with a string-cell hybrid method," *IOP J. Micromech. Microeng*, vol. 14, pp. 851-858, 2004.

[4] E. Strasser and S. Selberherr, "Algorithms and models for cellular based topography simulation". *IEEE Trans. on CAD of Integrated Circuits and Systems* 14 (9): 1104-1114, 1995

[5] G. Sun, X. Zhao, and G. Lu, "Voxel-Based Modeling and Rendering for Virtual MEMS Fabrication Process" ,*IEEE/RSJ IROS2006*Beijing, China, pp. 306311, 2006.

[6] NVIDIA "CUDA Compute Unified Device Architecture Programming Guide, V. 2.0", 06/07/2008

[7] CUDA http://developer.nvidia.com/object/cuda.html.

[8] F. Zhang, G. Wang, "An Improved Parallel Implementation of 3D DRIE Simulation on CPU", *10TH IEEE International Conference on High Performance Comuting and Communications HPCC 2008*, Dalian, Chin: pp. 891-896, Sept. 2008.

[9] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture". *Microprocessor Forum*, May 2007.

[10] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens, "Glift: Generic, efficient, random-access GPU data structures." *ACM Trans. Graph*, 25 (1): 60-99, 2006.

[11] D. Horn, "Stream reduction operations for GPGPU applications". In GPU Gems 2, Pharr M., (Ed.).AddisonWesley, ch.36, pp.573-589, Mar.2005

[12] S. Sengupta, M. Harris, Y. Zhang and J. D. Owens, "Scan primitives for GPU computing", *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2007.

[13] CUDPP http://www.gpgpu.org/developer/cudpp/

[14] A. Grama, A. Gupta, G. Karypis, V. Kumar, "Introduction to Parallel Computing (Second Edition)", Pearson Education, 2003.