An Improved Parallel MEMS Processing-Level Simulation Implementation Using Graphic Processing Unit

Yupeng Guo, Xiaoguang Liu, Gang Wang, Fan Zhang, and Xin Zhao

Nankai-Baidu Joint Lab, Inst. of Robotics and Information Automatic System
College of I.T., Nankai University, Tianjin, 300071, China
{zick_gyp,liuxg74,wgzwp,fanzhang555}@yahoo.com.cn,
xzhao@nankai.edu.cn

Abstract. Micro-Electro–Mechanical System (MEMS) is the integration of mechanical elements, sensors, actuators, and electronics on a common silicon substrate through micro fabrication technology. With MEMS technologies, micron-scale sensors and other smart products can be manufactured. Because of its micron-scale, MEMS products' structure is nearly invisible, even the designer is hard to know whether the device is well-designed and well-produced. So a visual 3D MEMS simulation implement, named ZProcess[1], was proposed in our previous work to help designers realizing and improving their designs. ZProcess shows the MEMS device's 3D model using voxel method. It's accurate, but its speed is unacceptable when the scale of voxel-data is large. In this paper, an improved parallel MEMS simulation implementation is presented to accelerate ZProcess by using GPU (Graphic Processing Unit). The experimental results show the parallel implement gets maximum 160 times speed up comparing with the sequential program.

Keywords: MEMS, Processing-level Simulation, Parallel, GPU, CUDA.

1 Introduction

While the electronics are fabricated using integrated circuit process sequences, the micromechanical components are fabricated using compatible 'micromachining' processes that selectively etch away parts of the silicon wafer or add new structural layers to form the mechanical and electromechanical devices. By modeling these 'micromachining' processes with the Mathematical Morphology Operation (MO) on voxel data, ZProcess, which is developed in our previous work [1,4,6], becomes a MEMS processing-level simulation implement. It uses voxel data to present MEMS production's 3D topography. Consideration of the production's micrometer-scale dimension (one millionth of a meter), usually we have to use 100,000,000 or much more voxels to insure the accuracy of MEMS production's topography. The problem is, running on CPU, the simulation speed will become very slow when the voxel data come to that scale. So it is necessary to develop a parallel simulation implement which can accelerate the simulation program.

C.-H. Hsu et al. (Eds.): ICA3PP 2010, Part II, LNCS 6082, pp. 289–296, 2010.

The rest of this paper is organized as follows. Section 2 introduces the basic MEMS fabrication processes and its MO model constructed by ZProcess, and the detail of the sequential algorithm. Section 3 gives the basic ideas on acceleration and the improved parallel algorithm. Section 4 shows the experimental data and the speed up we get. At last, we give our conclusion in section 5.

2 Basic MEMS Fabrication Processes and Their MO Model

One of the basic building blocks in MEMS Fabrication processes is the ability to deposit thin films of material. Usually we call it deposition processing. MEMS deposition technology can be classified in two groups, one is using chemical reaction and the other is using physical reaction. Using deposition technology we can get a thin film with the thickness between a few nanometer to about 100 micrometer. Because the surface of substrate, on which we deposit the thin film, may not be smooth, the device's surface will not be smooth also after deposition processing. For this reason, in MEMS processing-level simulation implement, we use Mathematical Morphology Operation (MO) to model the deposition processing [2]. The thickness of deposition film can be obtained through the processing parameters. We can add the voxels within the sphere whose center is the surface voxel and the radius is the thickness, just like rolling a ball on the substrate (Fig.2 shows the concept of MO).

As mentioned above, ZProcess is based on voxel method. The MEMS device is treated as a set of voxels. The value of each voxel is mapped into 0 to 255: the voxel assigned 0 representing the transparent background and the voxel assigned other value representing opaque objects and meaning the different materials. We store the voxels in a one dimensional array, the voxel's sequence number in the array is calculated as formula 1: we define Sn as the sequence number; dimX, dimY, dimZ is the dimension of the volume data respectively; and x, y, z is the voxel's coordinate.

$$S_n = Z \times \dim X \times \dim Y + y \times \dim X + x \ . \tag{1}$$

Algorithm 1 illustrates the sequential algorithm of deposition processing. Here，x, y and z is the dimension of MEMS device's voxel data.

Algorithm 1. Sequential Algorithm of Deposition Processing

```
for i := 1 to z do
  for j := 1 to y do
    for k := 1 to x do
      if the voxel[k, j, i] is a surface point
      then modify(set value to 1) the voxels within the
           sphere whose center is voxel[k, j, i] and
           radius is the thickness
```

Another basic processing is Etching. In order to form a functional MEMS structure, it is necessary to etch the thin films previously deposited or the substrate itself. Using the lithography and the mask, we can transfer the pattern we want to the material through etching processes. In program, we use mask data directly. In general, there are two kinds of etching processing: wet etching where the material is dissolved when immersed in a chemical solution; dry etch where the material is split using

reactive ions. In ZProcess, we model the wet etching processing by the same way as the deposition. The change we make is erasing the voxels instead of adding the voxels within the sphere. A limit is added in the algorithm also, the surface point which is preparing to erase must be in the etching mask. For dry etching, we vertically erase the voxels, which are surface points and in the etching mask, within the thickness.

Other basic processing, such as fabricating substrate, stripping resist, bonding and so on, can also be presented simply by operating on voxel data. In sequential program, we search the whole voxel data, set the chosen voxel to the substrate material in fabricating substrate processing or set the voxels whose value is equal to resist material to '0' in stripping resist processing.

With all these models we put forward above, we can give the 3D appearance of the MEMS device which produced by these basic processes. Fig.1 shows the simulation result of a micro-gripper, which is fabricated by totally 10 processes. In figure 1, the left is the micro-gripper's SEM photograph, the right one is the screenshot of the micro-gripper's simulation result (3D model) by MEMS Processing-level Simulation Implement. They are fabricating substrate, three deposition processes, four etching processes with different masks, bonding and stripping resist.

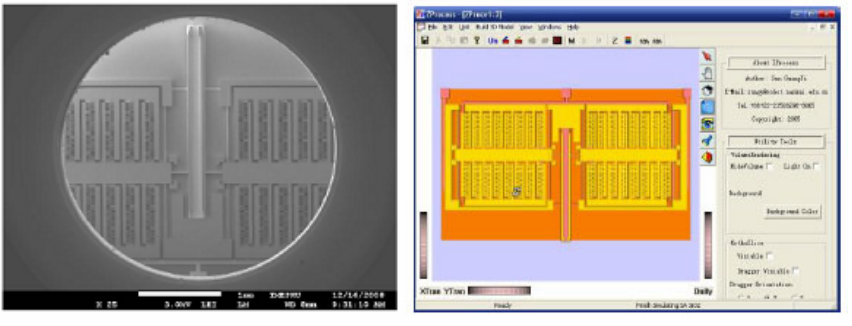

Fig. 1. The Micro-gripper's SEM Photograph and Simulation Result

3 Parallel Methods to Improve the Simulation Implement on GPU

3.1 Introduction of CUDA and the Basic Parallel Consideration

CUDA is short for NVIDIA's Compute Unified Device Architecture [7, 8, 9, 10]. It provides a programming interface to use the parallel architecture of NVIDIA's GPUs for general purpose computing. CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads. Each core has shared resources, including registers and memory. With the C language and CUDA's 'nvcc' compiler, it is convenient for developers to write CUDA program or embed it into other programs.

In the sequential simulation implement, we have to search most of voxels in volume data and set them to the right value when execute only one step of processes in simulation. When the volume data is very large, the program's speed becomes unacceptable. Unfortunately, the pattern of the etching mask is complex, if the volume data size is not large enough, the pattern in the mask will be confused when we transform the

mask's vector-graph into scalar-graph. For example, the adjacent combs with small distance in micro-gripper's structure layer may overlap. The simulation result, which is the device's 3D model, will be confused too. For this reason, usually we have to use more than 100 million voxels to represent a MEMS device's appearance. Since the operation on each voxel is not relevant, the program is well adapted to run on GPU because the task can be massively parallel. We can assign a single thread from the GPU to operate one voxel or more of the device's volume data. So we improved all the sequential processing simulation programs with parallel methods. We embed it into the original program also. Before the simulation, volume data is transferred from the host memory to device memory. After all the parallel processing simulation is completed on GPU, the volume data is transferred back to host memory for displaying.

3.2 Three Dimensional Fast Mathematical Morphological Operation(FMO)

In deposition and etching processing simulation, we can use FMO instead of MO [3]. Fig.2 shows the 2D schematic illustration. Different with original morphological algorithm [2], it is not necessary to access every voxel inside the sphere when performing erasing operations since a large number of voxels overlap between two adjacent spheres. As shown in Fig.2, P_n and P_{n+1} are two adjacent spheres. The overlapped oblique line part only needs to be erased one time. Since the MEMS device's appearance is irregular, we have to extend the FMO to the three dimensional FMO. Firstly, we calculate the voxels inside sphere with the radius which is equal to thickness. Secondly, we calculate voxels inside half shell of the sphere in positive half of x, y and z-axis with the result we get above. It is important to ensure the continuity of shell's surface. Since the voxel data is discrete, mathematical method which uses the formula for sphere surface to calculate a sphere's shell is not useful. If we want to get a half shell in the positive half of x-axis, for example, we could, for each voxel in plane y-z, search the voxel data we get in first step along the z-axis, from 'z' to '0', until get a sphere voxel. To different processing, we calculate the sphere and the half shells with different radius, and then put them into device's constant memory as templates.

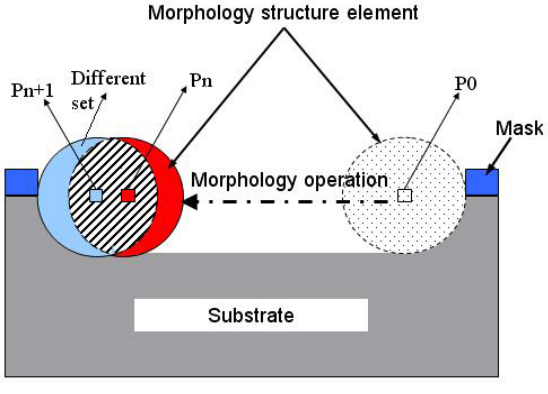

Fig. 2. Schematic Illustration of MO and FMO. In MO, we need to access every voxel in sphere. In FMO, we only need to operate the different part of the two spheres, as the schematic illustration shows.

3.3 Parallel Method for MEMS Processing Used in Simulation

For deposition processing and wet etching processing, before we use FMO to modify the volume data, we should find out the surface voxels firstly. If not, we have to put every voxel in the volume data into the deposition or etching kernel, and decide which one is the surface point in the kernel, that will lead to high thread divergence. To reduce the thread divergence, we use the operation on GPU called compaction [11]. Firstly, we design a kernel to find out all of the surface point, then we use an array which is called the surface-point array to store the surface voxel information. The array's size is same to the volume data. In this array, '1' represents the surface voxel and '0' represents non-surface voxel. Secondly, we have to compact these surface voxels. Using Cudpp [5], we scan the whole surface-point array. With that scan plan, we can calculate each element in the surface-point array, and the output data is the surface voxel's subscript in the compaction array which we want. So we define the compaction kernel as algorithm 2 shows.

Algorithm 2. Compaction Kernel

```
for each thread i
  if(surface array[i] == 1)
    compaction array[output data[i]] = i;
```

After that, we create threads block with the same size of compaction array. Since the compaction array storing the voxel's sequence number in the volume data, we can easily calculate the coordinate of each voxel by:

$$Z = S_n / (\dim X \times \dim Y) . \tag{2}$$

$$Y = (S_n - Z \times \dim X \times \dim Y) / \dim X . \tag{3}$$

$$X = S_n - Z \times \dim X \times \dim Y - Y \times \dim X . \tag{4}$$

For each surface point, we use FMO to modify the volume data. If the surface point's nearby voxel (in positive half of x-axis, y-axis, or z-axis) is a surface point too, we use the half-shell of a sphere as our FMO template. Otherwise, we use the whole sphere.

For dry etching, we find out the surface points and compact them too. Then we do not need to use FMO. We search each surface point in our etching mask data. If the surface point is in the mask, we erase the voxels within the depth.

For other processing, such as fabricating substrate and stripping resist, we define the thread block's dimension (CUDA allow developers to define three dimensional thread block) to fit the region of the volume data we want to operate. So each thread can operate one voxel. The thread's ID is just the voxel's coordinate. With the voxel's coordinate and processing parameters, we can decide to set the voxel as a part of substrate or strip it away from the device.

4 Experimental Results

We use following devices in our experiment: CPU: AMD Phenom II X4 945 (4 cores with 3.0GHz); Memory Size: 4Gb; GPU: NVIDIA Tesla 1060 with 4Gb global memory for parallel calculation (which has 240 stream processors), and NVIDIA 8600GT for displaying; Operation System: Red Hat AS 5.3; CUDA version: 2.2; GPU program compiler: nvcc; CPU program compiler: gcc version 4.2.3.

With the small volume data size, we can finish the simulation quickly. With the large data size, we can build a more accurate 3D model of the MEMS device. So in the experiment, we choose the volume data size from 3 million voxels to 200 million voxels. Table 1 shows the experimental result of Micro-gyroscope's simulation. We get the speed up from 23.0 times to 27.9 times. Here, the IO execution time means the time we used in reading mask data from disk for each etching processing. In Table 1, we can see the IO execution time occupy the most of time which parallel program used. To test the speed up of simulating MEMS processes with our parallel methods, we calculate the sequential and parallel program's runtime without IO time. The speed up which we get is from 69.3 times to 164.5 times. Comparing with the sequential program, the parallel program gets the great and stable acceleration result.

Table 1. Miro-gyroscope Simulation Experimental Result. We list sequential program's runtime and parallel program's runtime below, and then calculate the speed up we get. The timing unit is millisecond.

Micro-gyroscope Simulation Experiment	Volume Data Size(million voxels)				
Program Runtime (ms) and Speed Up	3	12	50	100	200
Sequential Runtime	1428	5627	22336	49360	89762
Parallel Runtime	62	213	828	1769	3266
Speed Up	23.0	26.4	27.0	27.9	27.5
IO Execution Time	42	168	679	1478	2630
Sequential Runtime (without IO)	1386	5459	21657	47882	87132
Parallel Runtime (without IO)	20	45	149	291	636
Speed Up (without IO)	69.3	121.3	145.3	164.5	137.0

To verify the stability of parallel algorithm, another experiment, which simulates a different MEMS device named Micro-gripper, is done. In this experiment, we choose the same volume data size as Micro-gyroscope simulation experiment. As Fig.3 shown, the simulation of Micro-gripper and Micro-gyroscope both get high speed up with different data size. Despite the different complexity of processing, we get the stable speed up. Especially, with the representative volume data size of 100 million voxels, in which we can ensure showing the MEMS device's appearance exactly with no confusion, we get more than 100 times speed up stably.

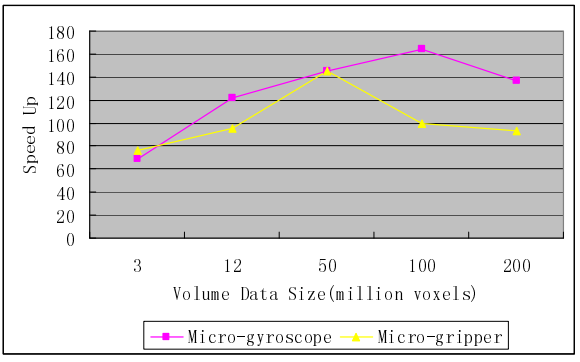

Fig. 3. The Experimental Results in Simulating Micro-gripper and Micro-gyroscope. (The x-axis is the volume data size and the y-axis is the speed up.)

5 Conclusions

The simulation to different process is the key technology to MEMS CAD software. Unfortunately, the simulation will spend much time because of its high algorithmic complexity and big data size according to real MEMS devices. In this paper, we present an improved parallel MEMS processing-level simulation implementation. By accelerating every basic MEMS processing's simulation algorithm on GPU, we get 28 times speed up in micro-gyroscope's simulation. Without IO execution time, the speed up will come to 160 times at most. We test different MEMS device, which produced with different processes and simulated by different volume size. In representative volume size, the experimental result of acceleration without IO is stable up to 100 times.

Acknowledgement. This work was supported by Program for New Century Excellent Talents in University (NCET-07-0464), National Natural Science Foundation of China (60875059), National High Technology Research and Development Program of China (2009AA04Z320), and Science and Technology Development Plan of Tianjin (08JCZDJC22000).

References

1. Sun, G., Zhao, X., Lu, G.: Voxel-Based Modeling and Rendering for Virtual MEMS Fabrication Process. In: IEEE/RSJ IROS2006, Beijing, China, pp. 306–311 (2006)
2. Sun, G., Zhao, X., Zhang, H., Wang, L., Lu, G.: 3-D Simulation of Bosch Process with Voxel-Based Method. In: Proceedings of the 2nd IEEE International Conference on Nano/Micro Engineered and Molecular Systems, Bangkok, Thailand, pp. 45–49 (2007)
3. Zhang, F., Wang, G.: An Improved Parallel Implementation of 3D DRIE Simulation on Multi-core. In: 10th IEEE International Conference on High Performance Computing and Communications HPCC 2008, Dalian, China, pp. 891–896 (2008)

4. Zhao, X., Li, Y., Zhou, Y., Ren, L., Lu, G.: Virtual Process: Concept, Problems and Implementation Framework. In: The Fourth International Conference on Control and Automation (ICCA'03), Montreal, Canada, pp. 659–663 (2003)
5. CUDPP, http://www.gpgpu.org/developer/cudpp/
6. Zhao, X., Sun, G., Ren, L., Lu, G.: On MEMS Design Automation. In: Proceedings of the 26th Chinese Control Conference, Zhangjiajie, Hunan, China, pp. 774–778 (2007)
7. NVIDIA, CUDA Compute Unified Device Architecture Programming Guide,V.2.0 (2008)
8. CUDA, http://developer.nvidia.com/object/cuda.html/
9. Nickolls, J., Buck, I.: NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum (2007)
10. Lefohn, A.E., Sengupta, S., Kniss, J., Strzodka, R., Owens, J.D.: Glift: Generic, efficient, random-access GPU data structures. ACM Trans. Graph 25(1), 60–99 (2006)
11. Horn, D.: Stream reduction operations for GPGPU applications. GPU Gems 2, 573–589 (2005)