

Implementing High-performance Intensity Model with Blur Effect on GPUs for Large-scale Star Image Simulation

Chao Li¹⁺, Yunquan Zhang², Changwen Zheng¹, Xiaohui Hu¹

¹ National Key Laboratory of Integrated Information System Technology

² Parallel Computing Laboratory

Institute of Software, Chinese Academy of Sciences, Beijing, China

{lichao09, yunquan, changwen, hxx} @iscas.ac.cn

Abstract—Intensity model with blur effects are widely employed to accurately simulate the imaging process of a star simulator used for attitude determination and guiding feedback. The model is computationally intensive and the time requirements are proportional to the number of stars in the simulation, imposing great demands of computing power for realistic uses.

This paper presents two star simulators using Graphic Processing Units (GPUs). We analyze the parallelism inherent in the intensity model and leverage a massive number of computing cores on GPU to efficiently exploit the fine-grain data parallelism. We first give a parallel simulator and discuss the performance trade-offs related to small amount of shared memory and the atomic operations on GPU. We then give the second simulator by adapting the first based on the characteristics of spatial locality with on-chip memory redesign. We analyze the balance between the kernel time and the non-kernel overhead in the two simulators and observe the inflection points in terms of two crucial model parameters. A selection table is given to choose between the two simulators. Benchmarks corresponding to the data parallelism are developed to fully evaluate the performance. The parallel simulator reports one to two orders of magnitude speedups with a maximum of $270\times$ compared to the widely-used sequential simulators and the average speedup is around 97 times. The adaptive simulator achieved up to $1.8\times$ compared with the parallel one over the inflection point. The developed code is currently used for simulating complex star images in a realistic large-scale star simulator.

Keywords- GPU computing; star image; CUDA; intensity model; blur effect

I. INTRODUCTION

Star simulator is an important aerospace simulation application, which produces real-time star imaging under any time and any attitude from the ground. The star image can be used in many devices, such as star sensor [1], an important instrument of attitude determination on satellite

that primarily uses star image for real-time attitude adjustment. In addition, a star simulator can also produce the celestial star background, widely used in space environment simulation systems. The intensity model is a crucial part of star image simulation. In order to achieve the realistic and accurate star imaging, the point spread function (PSF) has been widely adopted, which depicts the blur effect of optic system [2, 3]. Currently, star intensity simulation with blur effects takes several seconds or even minutes, far from real time, when produced by a large-scale star simulator system. The main reason is that realistic intensity simulation of a large-scale star image involves computationally onerous time-domain solution of thousands of blur effect algebraic computations, which must be solved for each star of a large-scale star catalogue in the FOV (Field of View). Since the 1980s, several simulators and software applications for the model have been developed to perform this simulation faster for large-scale systems [4, 5, 6]; the majority of these simulators have been developed for sequential architectures using languages such as Pascal, prolog or C. Their performance may be significantly improved by exploiting parallelism in star image simulation.

Nowadays, GPUs provide a compelling alternative to the traditional parallel environments such as cluster of multicore CPUs, delivering extremely high floating point performance and also a massively parallel framework for scientific applications which can exploit their specialized architecture [7, 8]. GPUs can support several thousands of concurrent threads in a massively parallel environment. Current NVIDIA GPUs, for example, contain up to 300 scalar processing elements per chip [7], they are programmed using C and CUDA, which provide a convenient way for programmers to develop GPU code. Moreover, they have a low cost compared with a multicore computer cluster.

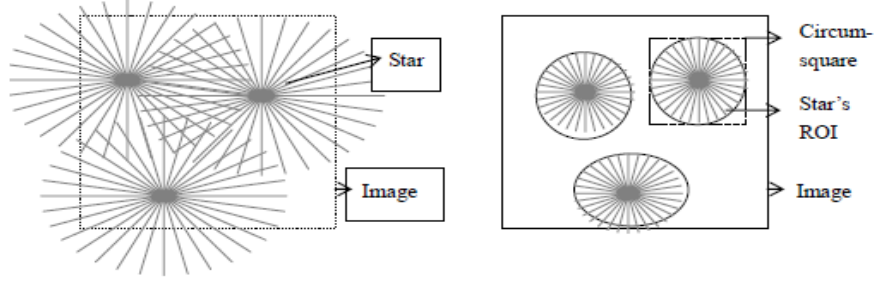


Fig 1. The effect of star brightness on image gray in different scope: in the left part of the figure, every star scatters its brightness into the whole image, even out of the bound, which is very time-consuming in calculating the distribution; in the right-side figure, the distribution scope is restricted in a square, i.e. ROI. Most of the star's energy constrained in ROI area.

In this paper, we focus on the parallelization of the model and introduce GPU-based intensity model for use in large-scale star simulation. Our work includes: a) parallelization of intensity simulation on GPUs and implementation of a parallel simulator; b) adapting parallel simulator to defined problem characteristic and GPU architectural idiosyncrasies with on-chip memory redesign; c). strategies in achieving high performance of our simulators. d). a performance balance analysis to direct the choice of two GPU simulators.

The rest of the paper is structured as follows: In Section 2 a realistic star intensity model of Gauss blur effect is defined. In Section 3 we explain the design details of the simulators. In Section 4 we show our experiments and performance analysis among the developed simulators; we further discuss the characteristic of our simulators. This paper ends with several conclusions and ideas for future work in Section 5.

II. MODEL DESCRIPTION

Star intensity model is defined to illustrate the intensity distribution of each star projecting on the space imaging device, which finally generates a star image of gray values. In this model, star can be considered as point light source for its far distance from the imaging device. The star image simulation can be considered as the response of imaging system of star points. In order to compute star image gray value, the brightness of light source must be given. In a star catalogue, the brightness of a star can be denoted by its magnitude. The brightness of a star and its magnitude can be concluded in the formula:

$$g(m) = A \times 2.512^{-m} \quad (1)$$

Where A is the proportion factor, m is the star magnitude usually ranged from 0~15, and g is the intensity of the star.

In realistic space environment, the light energy one star as a light source emits always scatters in the space domain. To simulate it, a blur effect is proposed in the space science. Specifically, the intensity spread by a star point in the image contributes the gray value of each pixel in the image. The mathematical description of star's blur effect is called point spread function (PSF). In the space camera's optical system, Gauss point spread function [2, 9] (i.e. Gauss blur effect)

can be used to accurately describe the intensity distribution rate of a star in a realistic way, the function holds the formula:

$$\mu(x, y) = \frac{1}{2\pi\delta^2} \exp\left[-\frac{(x-X)^2 + (y-Y)^2}{2\delta^2}\right] \quad (2)$$

Where δ is standard deviation reflecting the width of distribution circle, (X, Y) is the center where the intensity value of image spot is highest. The point spread distribution is obviously spatially symmetrical. $\mu(x, y)$ is the intensity contribution rate the star at (X, Y) exerts at pixel (x, y) .

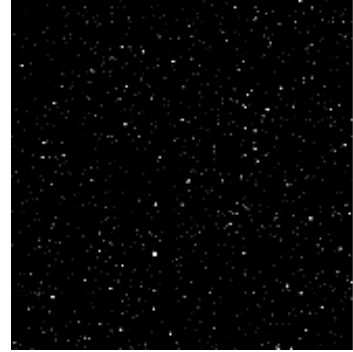


Fig 2. A segment of star image based on intensity model with blur effect

However, a pixel of the image can be determined under the process of computing all stars' intensity distribution at the pixel. In realistic large-scale simulator, the stars are in great amount and the size of star image is large, this leads to a very time-consuming simulation. To address it, the Gauss blur formula shows that the intensity distribution of a star to a certain pixel reduces drastically when the distance between the pixel and the star point expands, therefore, in computing the pixel gray value, most simulation systems generally adopt the method that the distribution scope is restricted, i.e. instead of covering the image plane, the coverage of star point's intensity distribution is imposed on a region of interest (ROI), which is a pixel circle centered by star point (see Figure. 1). The radius of the circle is relevant with optical parameters to assure good distribution effect, and empirically is set within a range from 2~20

pixels. Thus, for each star, an intensity distribution ROI is created. Only the pixels within ROI can reach the intensity distribution from the brightness of the star.

Here, the intensity distribution of a star on a pixel with image coordinate (x, y) can be computed by multiplying the star brightness and its intensity distribution rate at the pixel:

$$\phi(m, x, y) = g(m) \times \mu(x, y) \quad (3)$$

Where the pixel (x, y) is restricted into the range of a star's ROI.

Generally, by setting the Gauss blur parameters, with a given number of stars and a fixed star image size; a star image can be simulated by implementing above intensity model. Fig. 2 shows a segment of simulated star image (1024*1024) with 2252 stars projected.

III. DESIGN OF THE SIMULATOR

In this section, we will describe the simulators by implementing high-performance intensity model. Firstly, we explain the previous work we have done to prepare the development of parallel simulator on GPU. Then, we illustrate the parallel simulator that fully runs the data parallelism of intensity model, and introduce several strategies in parallelization of the model to achieve a high computing performance. Finally, we adapt the parallel simulator to defined problem characteristic of spatial locality and GPU architectural idiosyncrasies, to implement the adaptive simulator by on-chip memory redesign.

A Design of the baseline simulator: sequential simulator

As CUDA programming model is based on C language, therefore, the first recommend step of developing application in CUDA is to start from a baseline algorithm in C, where some parts of the sequential program can be convertible to be parallelized on the GPU. The sequential simulator design is straightforward, based on above intensity model. The simulation process can be divided into four stages: *Star generation*, *Star brightness computation*, *Pixel computation*, and *Output*. All of these stages are sequentially executed in this simulator, constituting a single-threaded CPU version of the model.

Firstly, *Star generation* is executed, stars in the FOV of image plane are retrieved, and each star contains a magnitude within the range of 0~15 and the coordinate in image plane. Then the simulator executes the *Star brightness computation* stage which calculates the star's brightness following the formula previously explained. The *Pixel computation* stage runs the computation of gray value of each pixel at the image sequentially, and finally the *Output* stage sends out the gray value to CPU platform to form a picture.

The input of this simulator is a star dataset, which is collected by retrieving stars that locate in the FOV of star image from star catalogue. The star obtaining process will not be discussed in this paper; detailed description of it can be found in [4]. The output pixel gray value will be written into a kind of common picture type like JPG, BMP, etc and

thus a piece of star image is rendered.

B Design of GPU simulator using CUDA: parallel simulator

1) Parallel strategy

The intensity model computes the gray value of each pixel by accumulating intensity contributions from stars within the ROI, and the number of calculations is proportional to the product of the total number of pixels and stars. Considering that a large-scale star simulator involves tens of thousands of stars and millions of image pixels, parallelization of the intensive computation is meaningful.

The attributes of the model show that the calculation of each pixel gray is independent of the calculation of other pixels. The position and magnitude information of each star are used many times in calculating its distribution to each pixel within its ROI. A good decomposition for the intensity model with blur effect is to form subproblems that calculate the gray value of each pixel. All of these subproblems can be solved in parallel with each other. It fits for a massive number of CUDA threads to solve them.

The key of arranging parallelism is to the way of decomposing the calculation work to be performed by each unit of parallel execution (a thread in CUDA). There are two alternative approaches to organize parallel execution of the model: star centric or pixel centric. In pixel centric option (in Fig.3 (a)), each thread denotes a pixel and calculates the accumulated intensity distributions to this pixel from the stars picked out. This would be a poor choice. As each thread has to identify all stars to select which ROI covers this pixel, and it will lead to many divergences in the warp execution. In CUDA, a highly divergent warp of 32 threads will be very inefficient. The star centric one uses each thread denotes a star and calculates the distribution of a star on pixels within its ROI. This will be a preferred approach, as the divergence in a warp execution is well eliminated (in Fig.3 (b)). However, this method will raise modification conflict in different threads; we must use atomic operations to prevent race conditions caused when a pixel locates in several stars' overlaid ROI. Here the overhead on atomic operation can be relieved because the possibility of ROI overlaying is relatively low, considering that stars in the image are generally scattered.

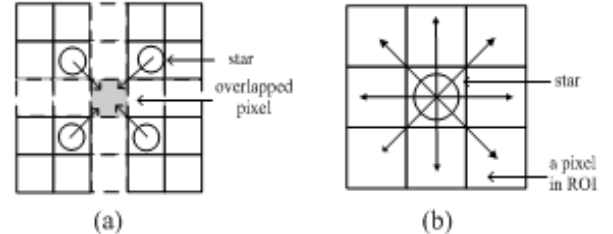


Fig 3. Two parallel computing modes: pixel centric (a) and star centric (b); (a) shows four stars' ROI overlaid at one pixel. Each star contributes its brightness to this pixel, and there are many other stars whose ROI excludes the pixel; (b) shows that a star distributes its brightness to pixels in its ROI.

2) Star-centric parallel model

As is depicted above, in star-centric model, each thread is used to compute the gray distribution of each star to a

pixel in its ROI. However, the calculation of star's distribution on different pixels is also independent and the data parallelism among pixels in a star's ROI also presents. Thus, we parallel the model by employing two levels of data parallelism to simulate the intensity model fully: parallelism among stars (the brightness of a star can be computed parallel with other stars) and parallelism among pixels inside the ROI of a star (the computation of the intensity distribution to a pixel runs independently). The parallel manner fits well into CUDA parallel architecture which capitalizes fine-grained data parallelism (Shown in Fig. 4). Specifically, we identify each star as a thread block where each thread represents a pixel in the star's ROI. Each thread block runs parallel in computing the brightness of the star it denotes, and an individual thread in the block is responsible for computing star's intensity distribution to the pixel it denotes. In the parallel model, more threads, i.e. parallel units will be generated by the two-level thread arrangement. The model runs in a full data-parallel fashion.

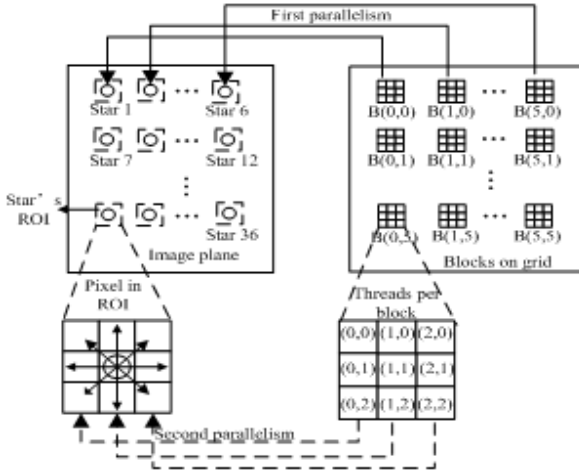


Fig 4. Parallel star-centric intensity model with Gauss blur effect: it configures a specific model: the stars num is 36 and ROI size is 3×3 ; First parallelism exists in block per grid and star per image, Second parallelism exists in thread per block and pixel per star's ROI.

3) GPU implementation

The objective of the parallel simulator is to simulate the process of gray model for star image, doing this in a parallel way whenever is possible. To do that, we use the baseline design on the four stages of the gray model computation. The first and last stage with little computation-demand is developed on the CPU, the same as baseline simulator has done; and the two computation stages are off-loaded to the GPU to be processed in parallel, developed into a GPU kernel. In our implementation, we present several performance trade-offs related to small shared memory and atomic operations on GPU to achieve high performance of the simulator.

First, we deploy the star centric parallel model in a CUDA execution manner. The two-level parallelism of the intensity model can be implemented by declaring two thread levels: *blocks* (the number of parallel blocks per

grid), and *threads* (the number of threads per block). They have different dimensions: the blocks in CUDA can be 1-dimensional, 2-dimensional, and the threads can be 1-dimensional, 2-dimensional or 3-dimensional. The dimension of each level can be tuned according to specific problem domains. For our kernel, we declare the *blocks* in a two-dimensional manner; the definition will ensure the simulation requirements of massive amount of stars in our model. The thread organization in each block, i.e. *threads* we have defined also adopts 2-dimensional fashion. The reason is that, as is shown in Fig. 5, in part of sequential version, two-level loop is used to compute the intensity distribution of a star to each pixel in its ROI; each pixel's *x* or *y* coordinate is identified by one loop iterator. The two-dimensional threads can perfectly match the two-level loop as a parallel alternative. We can get a clear overview of the thread hierarchy that is configured to parallel simulate our model from Figure 4.

```

for (i from 0 to starCount)
    mag ← starArray[i].mag;
    starPosX ← starArray[i].posX;
    starPosY ← starArray[i].posY;
    integer pixelX;
    integer pixelY;
    for (pixelY from starPosY-MARGIN to starPosY+MARGIN)
        for (pixelX from starPosX-MARGIN to starPosX+MARGIN)
            if (pixelX & pixelY locate in the range of the image)
                starBgt ← calculate star brightness;
                imagePixelArray[pixelY*img_width+pixelX]
                    +← calculate pixel gray
            end for
        end for
    end for

```

Fig 5. The loop inherent in CPU version: one-loop way in identifying the star and two-loop way is used to iterate the pixel in each star's ROI.

The model data required for GPU implementation is another important part of our simulator. The data containers for stars and pixels are firstly produced, as we can expect. In addition, we also implement indicator elements in the interface of our kernel to prevent the wrong address access of parallel threads in the execution context. Specifically, two categories of information are passed as parameters to ensure a safe data deployment. The first category is the parameters of image: the size of image checks cross-bound of star's ROI and a pointer to device memory stores the output image pixel. The second one is the star's parameters: starCount that is essential to prevent the index of thread blocks from overshooting, and a pointer to device memory that provides the information of stars on the image. When kernel runs, the data containers will be transferred into GPU memory. To exploit high-bandwidth of GPU memory subsystem, the data organization on GPU is focused. First, the star array and pixel array are loaded into the GPU global memory. As the threads in a block shared the same star

| The kernel Pseudo-code of parallel simulator | |
|--|---|
| Input: | Integer: image_width, image_height, starCount; star*starArray; |
| Output: | float* imagePixel |
| 1. | __shared__ float shareMem[3]; |
| 2. | threadX \leftarrow threadIdx.x, threadY \leftarrow threadIdx.y, /* identify thread index in a thread block*/ |
| | blockId \leftarrow blockIdx.x + blockIdx.y*gridDim.x /* identify block index in thread grid */ |
| 3. | if (blockId >= starCount) return; |
| 4. | magnitude \leftarrow starArray[blockId].mag; |
| 5. | if(threadX ==0 && threadY ==0) /* compute and store star's brightness */ |
| | { shareMem[0] \leftarrow calculate the brightness of starArray[blockId]; |
| | shareMem[1] \leftarrow starArray[blockId].posX; |
| | shareMem[2] \leftarrow starArray[blockId].posY; } |
| 6. | __syncthreads (); /* synchronize all threads in this point*/ |
| 7. | starPosX \leftarrow shareMem[1]; |
| | starPosY \leftarrow shareMem[2]; |
| | pixelX \leftarrow starPosX - MARGIN +threadX /*compute each pixel position in each star's ROI */ |
| | pixelY \leftarrow starPoxY - MARGIN +threadY /* MARGIN is the length of ROI */ |
| 8. | if (pixelX & pixelY in the range of image) |
| | { grayDistribution \leftarrow compute the star's contribution on this pixel; /* using PSF method */ |
| | atomicAdd (& imagePixel[pixelY*image_width+pixelX], grayDistribution); } |
| 9. | return imagePixel; |
| 10. | end kernel |

Fig 6. The kernel Pseudo-code of parallel simulator on CUDA: each emphasized keyword in black bold type indicates a core technology or key step in designing the kernel

information from star array, thus, when loading the stars, all threads within the same warp will access data from the same contiguous memory, enabling coalesced access. To reduce access overhead of global memory, we designate one thread per thread block to read the star information and others threads share it in low-latency shared memory. This will be described later. Then, for pixel array of image, the access behavior of threads is in the fashion of spatial locality. Each thread computes the intensity distribution and put the result into fixed global memory indicated by its x and y thread dimension, this leads to a fixed global memory access manner. We will fully take advantage of the characteristic of spatial locality in our adaptive simulator by another simulation pipeline.

The kernel of parallel simulator is executed into two consecutive steps: star brightness computation step and pixel computation step. The first step calculates the brightness of stars, which will be used in computing the gray distribution of the stars at the second step. They are working together to calculate the pixels value of star image. Fig. 6 illustrates the Pseudo-code of kernel.

At the first step, a star's magnitude and coordinate are accessed by the index of block, i.e. blockId, and the brightness of the star is computed parallel in block level. The brightness result and coordinate of each star will be read by all threads in this thread block. This is because the

star contributes brightness to every pixel in its ROI, and each pixel's position in ROI is determined by the star's position and the thread index together. Furthermore, the values of the star magnitude and position are not modified during the brightness computation. This means that the brightness value of each star can be computed ahead and then together with star's coordinate, both are stored in the on-chip shared memory, which will be available to every thread in the thread block. To implement it (See the Fig. 7), we identify the first thread in the thread block to compute the brightness value of the star and store it in shared memory (Step.5 in Fig. 6), and the threads in a thread block should be synchronized in case certain thread may read the empty shared memory before it is written (Step.6 in Fig. 6). By deploying the share memory usage among threads in a block, the global memory access frequency will be reduced from all threads to one thread per block, and the brightness computation is performed only once per block. In GPU, one shared memory call costs 1~4 clock cycles while a global memory access need 400~600 clock cycles of latency. Thus, this strategy will effectively enhance the performance of the simulator in both memory access and computation.

At the second step, the gray value of the pixel that each thread represents is calculated. The star brightness and star position are used and accessed by reading share memory. The star position is read two times (Step.7-8 in Fig.6). To

decrease access overhead, each thread first reads the star position in share memory into local registers, and then accesses the local registers for it (Step.7 in Fig. 6). This will relieve the bank collision of share memory generated by different threads accessing it simultaneously. After reading star coordinate from local registers, each thread calculates the pixel coordinate according to its two-dimensional index. Following the step, the thread will execute the intensity distribution calculation of the star to the pixel, and modify the gray of the pixel by adding the intensity distribution (Step.8 in Fig. 6). Here, it is noted that the ROI of different stars within a short distance is likely to overlap, and the operation of pixels in overlapped region will invoke the write-collision once different threads change a pixel's gray value simultaneously. To solve it, we employ the atomic method by putting an atomic add operation on the pixel. This enables parallel threads to safely make concurrent modification to the shared data (Step.8 in Fig. 6). However, the emergence of latency is caused by queuing for the same memory modification. In our model, the stars in simulation are distributed relatively scatterly and this translates into a relatively small number of threads that are competing for simultaneous modification to the same memory address.

Beyond the execution of the kernel, we need to transfer the image pixel array from GPU global memory into CPU memory. This will certainly bring in transmission overhead. Similarly, before the execution of kernel, memory transfer between host and device also needs. The transmission overhead, though inevitable for such hybrid system, should be eliminated as low as possible by applying some CUDA transmission optimization strategy, which has been described a lot in [10].

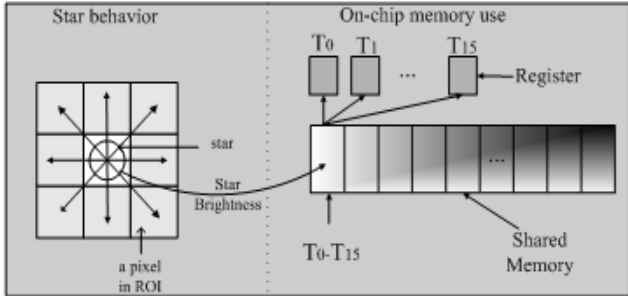


Fig 7. The on-chip memory use of parallel simulator: (1) threads in a block shares the same star brightness. (2). Registers are used to replace share memory in frequent memory calls for each thread.

C Adapting parallel simulator to the specific problem characteristic: adaptive simulator

Usually, a star simulator will be labeled with a star magnitude range which indicates its detecting ability of star in celestial environment. For star map simulation, it means a determined range of star brightness. A fixed-length array can be used to store the star brightness of different star magnitudes. Similarly, the size of ROI representing the optical performance is also fixed for a star simulator. Thus, we can compute a star's intensity distribution in its ROI and store it in a two-dimensional matrix. With a fixed star magnitude and side of ROI, we can build a

three-dimensional lookup table which contains each magnitude of a star and its intensity distribution matrix. A pre-built lookup table ahead of the kernel execution can shift part computation task from the kernel method into the memory access of the table. As we can expect, there might exist a performance balance between the shift process. This manner decreases the kernel execution time meanwhile increasing the non-kernel overhead in building and memory accessing of the lookup table. And this issue will be carefully studied in our performance analysis in Section 4. The building process of lookup table is shown in Fig. 8. We store the search tables in texture memory of GPU. There are two reasons. First, the thread access of the table has the character of 2D spatial locality, as we have illustrated in Figure. 4. This makes use of one advantage of texture memory access manner which capitalizes 2D locality, enabling a higher access bandwidth in such condition. Second, the texture memory has the texture (L2) cache, which will speed up the access when the same star data in lookup table has been accessed several times.

In implementation of the simulator, the parallel strategy and star-centric model is applied in the same way as the parallel simulator does; but for the kernel method, the computation of star brightness and distribution of star on its ROI will be replaced by accessing the search table in texture memory. Then, the content of shared memory kernel method is also changed by storing star magnitude instead.

Our source code for both CPU and GPU versions are freely available to download from a web page [11], and the terms of use are included in the code packet.

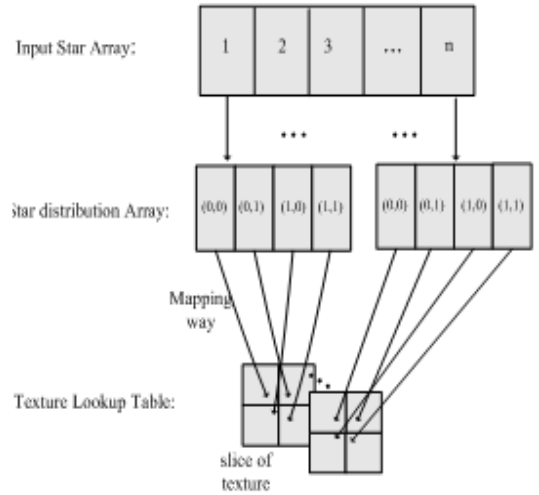


Fig 8. The process of building lookup table.

IV. PERFORMANCE ANALYSIS AND DISCUSSION

In this section, we analyze the performance of the three simulators presented above: the sequential simulator developed in C++, the parallel simulator and the adaptive simulator both on GPU using CUDA. The GPU used for the experiments is a NVIDIA GPU GTX480 which has 480 execution SPs and 1.5 GB of device memory, plugged in a computer server with an Intel core i7 @2.80GHz CPU and

3.5 GB of RAM. Although the CPU has eight cores, to accurately control the execution of sequential simulator and to have a clear comparison, we only employ one of the eight cores to run the sequential code on CPU. The CUDA programming version we adopt is version 3.2.

We developed two benchmarks (called test1, test2 respectively) to analyze the performance behavior of our simulators in two ways: increasing the numbers of stars simulated on the image (and so the number of thread blocks in grid increases) and increasing the side length of ROI (and so number of threads per thread blocks increases). In the experiment, these stars are the simulated data which have been generated randomly. The star information at image plane generates in such format file by configuring the two parameters: the magnitude of the star, the 2-dimensional coordinate in image plane. As is mentioned in Section 3, the number of thread blocks is equal to the number of stars in the star image; and the number of threads per block corresponds to the size of star's ROI with a two-dimensional shape.

A Benchmark test 1

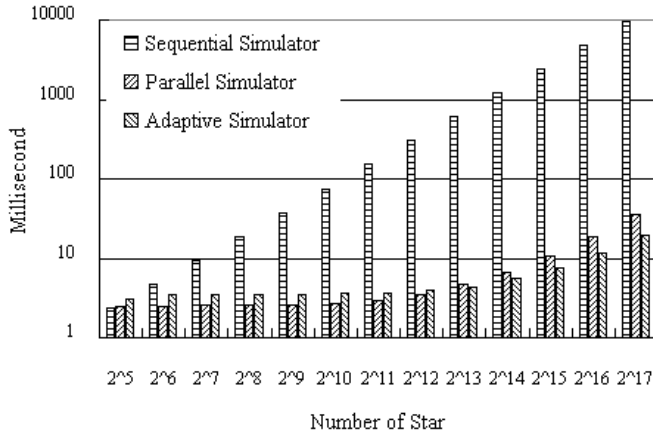


Fig 9. Simulation performance for sequential, parallel, adaptive simulators: test1

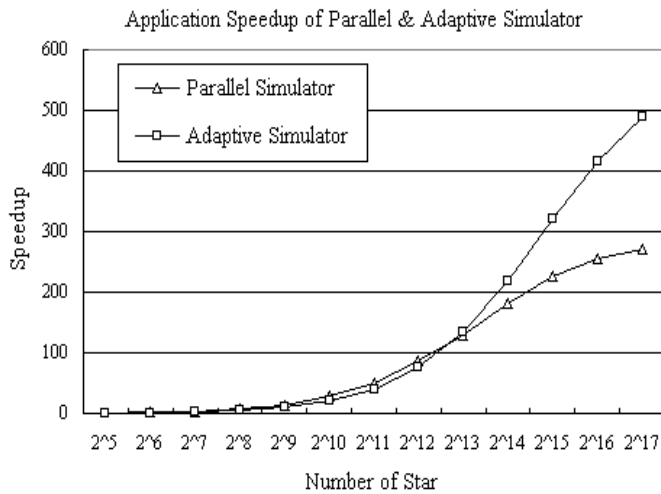


Fig 10. Speedup of parallel simulator, adaptive simulator to sequential simulator: test1

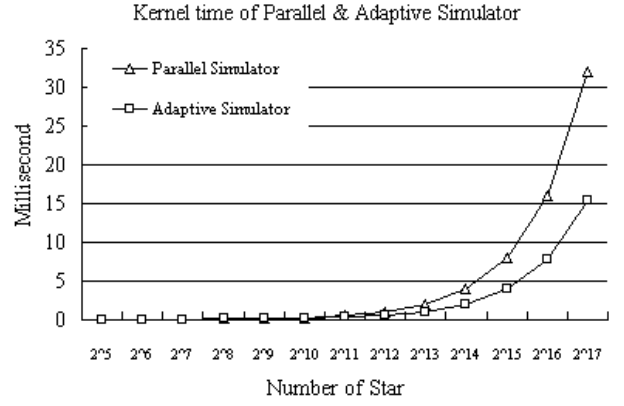


Fig 11. Kernel time in parallel & adaptive simulator: test1

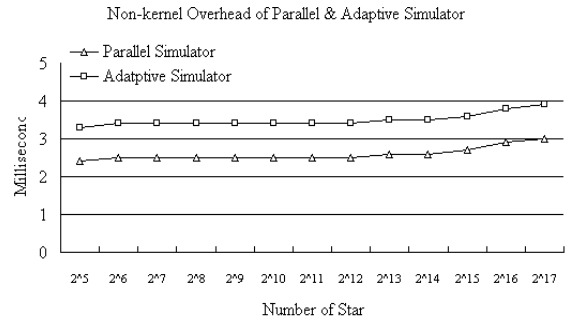


Fig 12. Non-kernel time in parallel & adaptive simulator: test1

Fig.9 shows the experimental performance of the simulators for test 1, the benchmark of test 1 increases the number of stars in the star image until reaching the configuration of 2^{17} , the number of simulated stars is constrained by the available memory of the simulator. The size of star's ROI is fixed to 10×10 , which means 100 threads per block, and it has also fixed the image size to 1024×1024 .

The behavior of three simulators, as shown in Fig. 9, is compared. With the number of stars increase, the execution time of simulator 1 increases linearly in a fast-ascending manner while the time consumption of the two GPU simulators rises slowly. When the number of threads is low, the performance of GPU codes is not advantageous. Due to the low data parallelism, we cannot fully take advantage of the massive computing resources available on the GPU. However, as long as the number of stars increases, the data parallelism of the simulation increases quickly; GPU computation cores are less idle, and this translates into the better performance of parallel GPU code. Fig. 10 shows the application speedup of the two GPU simulators compared to sequential one. The parallel simulator achieves a better speedup than the adaptive simulator at the early stage; with the increasing stars, the adaptive simulator catches the parallel one and overtakes it much when stars reaches 2^{17} . The speedup updating condition is determined by the dynamic time variation of the simulators' execution parts. Fig.11 and Fig.12 shows the breakdown of the two GPU simulators. When the number of stars is less than 2^{13} , the

TABLE I. THE BREAKDOWN OF NON-KERNEL PART FOR ADAPTIVE SIMULATOR: TEST1

| Stars Time(ms) | 2 ⁵ | 2 ⁶ | 2 ⁷ | 2 ⁸ | 2 ⁹ | 2 ¹⁰ | 2 ¹¹ | 2 ¹² | 2 ¹³ | 2 ¹⁴ | 2 ¹⁵ | 2 ¹⁶ | 2 ¹⁷ |
|---------------------------|----------------|----------------|----------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| CPU-GPU Transmission | 2.43 | 2.45 | 2.52 | 2.51 | 2.50 | 2.51 | 2.50 | 2.51 | 2.61 | 2.67 | 2.71 | 2.89 | 3.01 |
| Lookup Table Build | 0.70 | 0.71 | 0.71 | 0.72 | 0.71 | 0.71 | 0.70 | 0.71 | 0.72 | 0.71 | 0.72 | 0.71 | 0.71 |
| Texture Memory Binding | 0.21 | 0.20 | 0.21 | 0.21 | 0.20 | 0.21 | 0.21 | 0.20 | 0.21 | 0.20 | 0.21 | 0.22 | 0.22 |

kernel execution time of simulators increases little, remaining a small value, and the non-kernel overhead takes up most part of application time. As the adaptive simulator needs to build the lookup table in the texture memory, this leads to more non-kernel overhead in adaptive simulator than that in the parallel one, and it translates into the lower speedup of adaptive simulator at the early stage. In table 1, we can see the detailed breakdown of non-kernel overhead for adaptive simulator. The time of lookup table building and texture binding varies a little due to the constant side of ROI, but the two overhead constitutes the disadvantage part of adaptive simulator in non-kernel simulator. Based on the breakdown of each simulator, we can further analyze the variation condition of two GPU simulator's behavior. As the number of stars increases, the kernel time rises in a rocket way compared to its non-kernel overhead (See Fig. 11). The parallel simulator costs much more time than the adaptive one in kernel execution due to its more computing operators. For parallel simulator, the time advantage in non-kernel part (Fig. 12) at the early stage can't catch the increasing time disadvantage in kernel execution when the number of star continues to increase. This translates into the higher application overhead for parallel simulator than the adaptive one in later stage. In Table 2, we show the peak flop count of two GPU simulators' kernel execution (num of stars is 2¹⁷). The adaptive simulator has a lower GFlops due to the less computing operators and more memory call. Though the adopted GPU chip has a theoretic peak GFlops of 168, considering the frequent memory calls and kernel context overhead, the achieved arithmetic float speed of the two simulators is good. In terms of application-level throughput, the implementation of parallel simulator can process 9.507 billion float computations on pixel per second.

TABLE II. THE EXECUTION GFLOPS : TEST1

| Number of Star | Parallel Simulator (GFLOPS) | Adaptive Simulator (GFLOPS) |
|-------------------|--------------------------------|--------------------------------|
| 2 ¹⁷ | 95.07 | 93.8 |

B Benchmark test 2

The benchmark of test 2 increases the side length of ROI until reaching a configuration of 32×32. The area of ROI equals the number of threads per block, so the side length of ROI is constrained by CUDA's computation capacity (GTX480 is 2.0), indicating that the maximum of threads per block is 1024. The number of stars in the simulation is fixed to 8192, which means 8192 blocks per grid. The simulated image size is 1024×1024.

Fig. 13 shows the experimental performance of the three simulators for test 2. The cost of sequential simulator

increases in a linear way, which is expected. The two GPU simulators have a similar application cost of the simulation. The application speedup of GPU simulators is showed in Fig. 14. With the early increase in the side of ROI, the parallel simulator has a minor time advantage compared to the adaptive one due to the extra overhead of building lookup table in adaptive simulator. However, when the side of ROI reaches 10, the performance condition of the two simulators has changed. The adaptive simulator begins to overtake the parallel one in speedup. This behavior variation can be explained by the time breakdown of each GPU simulator. Fig.15 shows the kernel time and non-kernel overhead of the two simulators. When the side of ROI is small, the non-kernel overhead has occupied the most share of application time for both simulators. The parallel simulator has a lower cost on non-kernel part of the two, and this translates into its speedup advantage at this stage. However, with the increasing side of ROI, the time share of different parts in application time is changing. The kernel execution percentage is rising up with a fast drop of non-kernel time share for both simulators.

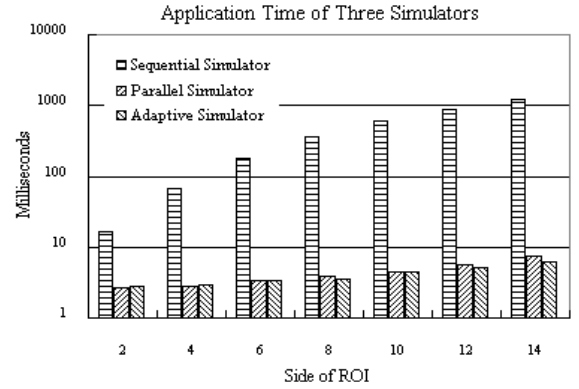


Fig 13. The overall performance for sequential, parallel, adaptive simulators: test2

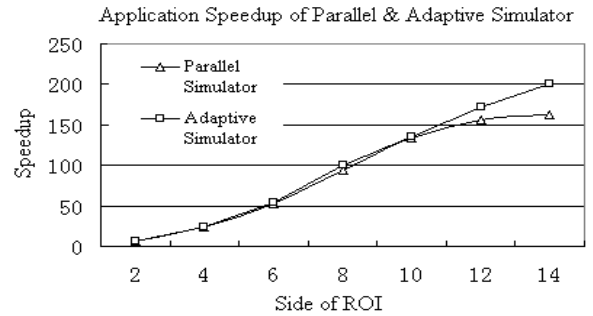


Fig 14. Speedup of GPU simulators to sequential simulator: test2

Fig. 16 shows the variation of non-kernel time percentage in application time. The percentage of non-kernel overhead in parallel simulator drops faster because the kernel execution time has gone up much more quickly than that of the adaptive one. With the trend going until the side of ROI reaches 10, the inflection point emerges. The faster-increasing kernel time of parallel simulator has directly enabled the exchange of speedup advantage between two GPU simulators. In test2, when side of ROI reaches 14, we report up to a speedup of $163\times$ between the parallel and sequential simulators. For adaptive and sequential simulators, we achieve a speedup of nearly $200\times$.

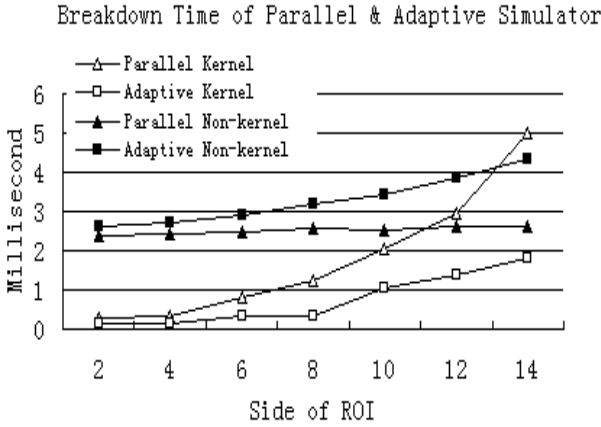


Fig 15. Breakdown of parallel simulator, adaptive simulator: test2

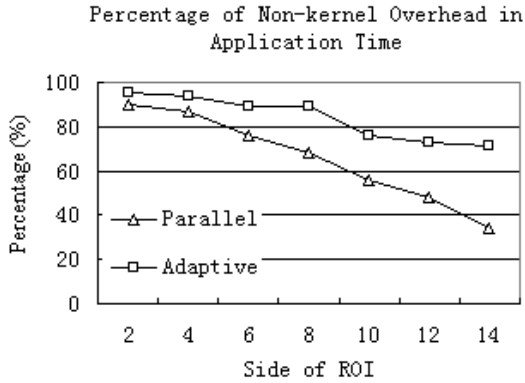


Fig 16. Percentage of non-kernel overhead for parallel simulator, adaptive simulator: test2

C Inflection point observation

The adaptive simulator has shifted the execution of star distribution with fixed star magnitude range from kernel into texture memory access by creating a lookup table in texture memory, and the parallel simulator has take the lead on performance when computation scale is not very large due to its advantage in non-kernel overhead compared to the adaptive one. However, the adaptive simulator overtakes the parallel one when computation scale becomes larger in terms of a large star number or side of ROI. As is shown in Figure. 10, in test 1, the side of ROI is fixed as 10, the inflection point comes when number of stars reach 2^{13} ; In

test 2 (see Figure. 14), the number of stars is fixed as 8192 (2^{13}), the inflection point comes when side of ROI meets 10. The two tests accord perfectly in the value of two model parameters at the inflection point, which should be achieved, or else, there must be mistakes in either simulator. From the standpoint of an end user, the inflection point is meaningful as it can direct you in selecting the best simulator that fits the star image simulation. The simulator selection criteria for different model parameters is showed in Table. 3. Obviously, a fixed parameter with the other tunable one translates into different selections of GPU simulators.

TABLE III. THE GPU SIMULATOR SELECTION

| Turning Point Simulator Choice | Number of Star(2^{13}) | Size of ROI (10) |
|-----------------------------------|----------------------------|------------------|
| Parallel Simulator | = | < |
| Parallel Simulator | < | = |
| Adaptive Simulator | = | > |
| Adaptive Simulator | > | = |

D Discussion

Based on the developed simulators, we have evaluated the performance of each simulator's behavior. To fully understand our simulators, here we'd like to discuss the limitation of our simulators in simulation scale and factors that affects performance. In our parallel strategy for intensity model, we adopt the star-centric simulation way. Correspondingly, the thread arrangement of kernel is designed to run the two-dimensional data parallelism of the model in GPU inherent thread fashion. The threads in a thread block are assigned to map the pixels in ROI. We know that the thread block has a maximum of 1024 threads, and this translates into the limitation on the size of ROI. The parallel simulator limits the scale of ROI, i.e. the size should be under 16. In most cases, the simulator is applied well, but the limit should be noted when using the parallel simulator. Besides, the adaptive simulator is dependent on the lookup table in texture memory. The texture memory is a limited on-chip resource even though latest GPU provides a bit more available texture memory size. Thus, we should first determine the size of lookup table to assure that it can be successfully bound into the GPU texture memory. As the maximum size of ROI is determined, we can calculate the maximum star magnitude range that the simulator can simulate with the fixed size of texture memory in a specific version of GPU chip. When building the lookup table, we run it in CPU platform instead of GPU kernel, due to the small execution overhead and little data parallelism. It is also necessary to remark that the non-kernel overhead occupies much in overall application time, and when the number of threads is low, the percentage is more notable. Therefore, when the star image is in a very small-scale (num of stars : $0\sim 2^7$), the sequential simulator on CPU can be a competent choice with a relatively promising performance, as this case can not fully take advantage of GPU computing power but with an extra pay for

communication overhead in such hybrid system.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have designed and analyzed three simulators for simulating intensity model in large-scale star imaging. The first simulator performs sequentially the computation of intensity model on the CPU. The other two simulators (parallel simulator, adaptive simulator) have been developed on the GPU. The parallel simulator fully simulates the data parallelism of the intensity model with several optimized parallel strategies. The adaptive simulator adapts the parallel simulator to the GPU architecture idiosyncrasies and problem characteristic.

Doing this, we report up to 270× of speedup between parallel simulator and sequential simulator, and up to 1.8× between two GPU simulators. In this work, we show two results. On one hand, our experimental results demonstrate that GPUs are good platforms to simulate star image due to the highly data parallelism presented in the model. On the other hand, if the parallel simulation behaviors are redesigned by using on-chip textured memory to be adapted to the GPU architecture idiosyncrasies and characteristic of spatial locality, the performance of the simulations can also be improved. However, there exists a balance between the non-kernel overhead and kernel execution in GPU simulators. We provide the performance inflection point of the two GPU simulators to direct the selection between them.

We also mention that our simulator presented has limitation on the available resources on the GPU (device memory, PCI-E between GPU and CPU) in two ways: the texture memory size limits the scale of lookup table in adaptive simulator, and thread number per thread block on GPU limits the size of ROI in star image simulation. This can be improved with the development of GPU general computing; the current capacity of our simulator can support the existing model requirement in realistic star image simulation. Now the clusters of GPUs provide a higher massively parallel environment [12, 13]. Our future work will focus on scaling our simulators to multiple GPUs in order to obtain better performance and also more memory space for our simulation.

ACKNOWLEDGMENT

Authors acknowledge the financial support by a grant from the National High Technology Research and Development Program of China (863 Program) (No. 2009AA01Z303). We totally express our thankfulness to Professor Chen Ding in University of Rochester who helps this paper a lot. We also thank Jose M. Cecilia for his insightful work [8] that gave this article some inspirations.

REFERENCES

- [1] Carl Christian LieBe, Star Trackers for Attitude Determination, IEEE AES Systems Magazine: 10-16, June 1995.
- [2] George V. Poropat, Effect of system point spread function, apparent size, and detector instantaneous field of view on the infrared image contrast of small objects, Optical Engineering: 2598-2607, October 1993.
- [3] Liebe. C.C. , Accuracy performance of star tracker-atutorial, IEEE Transactions on Aerospace and Electronic Systems: 587-597, April 2002.
- [4] Hye-Young KIM, John L. JUNKINS, Self-organizing guide star selection algorithm for star trackers: thinning method, In Proceedings of 2002 IEEE Aerospace Conference, pages 2275-2283, March 2002.
- [5] Shaodi Zhang, Honghai Sun, Yanjie Wang, Xiaomeng Jia, Hao Chen. Design of High Precision Star Image Locating Method Used In Star Sensor Technology. In Proceedings of 2010 International Conference on Computer, Mechatronics Control and Electronic Engineering, pages 411-414, May 2010.
- [6] Yang Yan-de Wang Jiang-yun Zhu Yu-tong. High-Precision Simulation of Star Map Using Forward Ray Tracing Method. In proceeding of ninth International Conference on Electronic Measurement & Instruments, pages 541-544, August 2009.
- [7] John Nickolls, Michael Garland, Scott Le Grand, etc. parallel computing experiences with CUDA, IEEE Micro: 13-25, July 2008.
- [8] Jose M. Cecilia b, Jose M. Garcia b, Gines D. Guerrerob, etc. Simulating a P system based efficient solution to SAT by using GPUs. The Journal of Logic and Algebraic Programming: 317-325, March 2010.
- [9] Liu Tai-yang, Wang Shi-cheng, Liu Xing-miao. Attitude Information Deduction Based on Single Frame of Blurred Star Image. In Proceedings of 2nd International Conference on Future Computer and Communication, pages 642-646, May 2010.
- [10] NVIDIA, NVIDIA CUDA Programming Guide 3.2, 2010.
- [11] http://gucas.academia.edu/chaoli/Teaching/25421/CPU_AND_GPU_CODE_on_Intensity_model_with_blur_effect
- [12] Y. Chen, E. Li, J. Li, and Y. Zhang. Accelerating video feature extractions in cbvir on multi-core systems, Intel Technology Journal, vol. 11, no. 04, November 2007. [Online]. Available: http://www.nvidia.com/object/cuda_get.html
- [13] Mamadou. D, Chrysostomos. N, Johnman. K. Large-Scale Semantic Concept Detection on Manycore Platforms for Multimedia Mining. In proceedings of 2011 IEEE International Parallel & Distributed Processing Symposium, pages 384-393, May 2011.